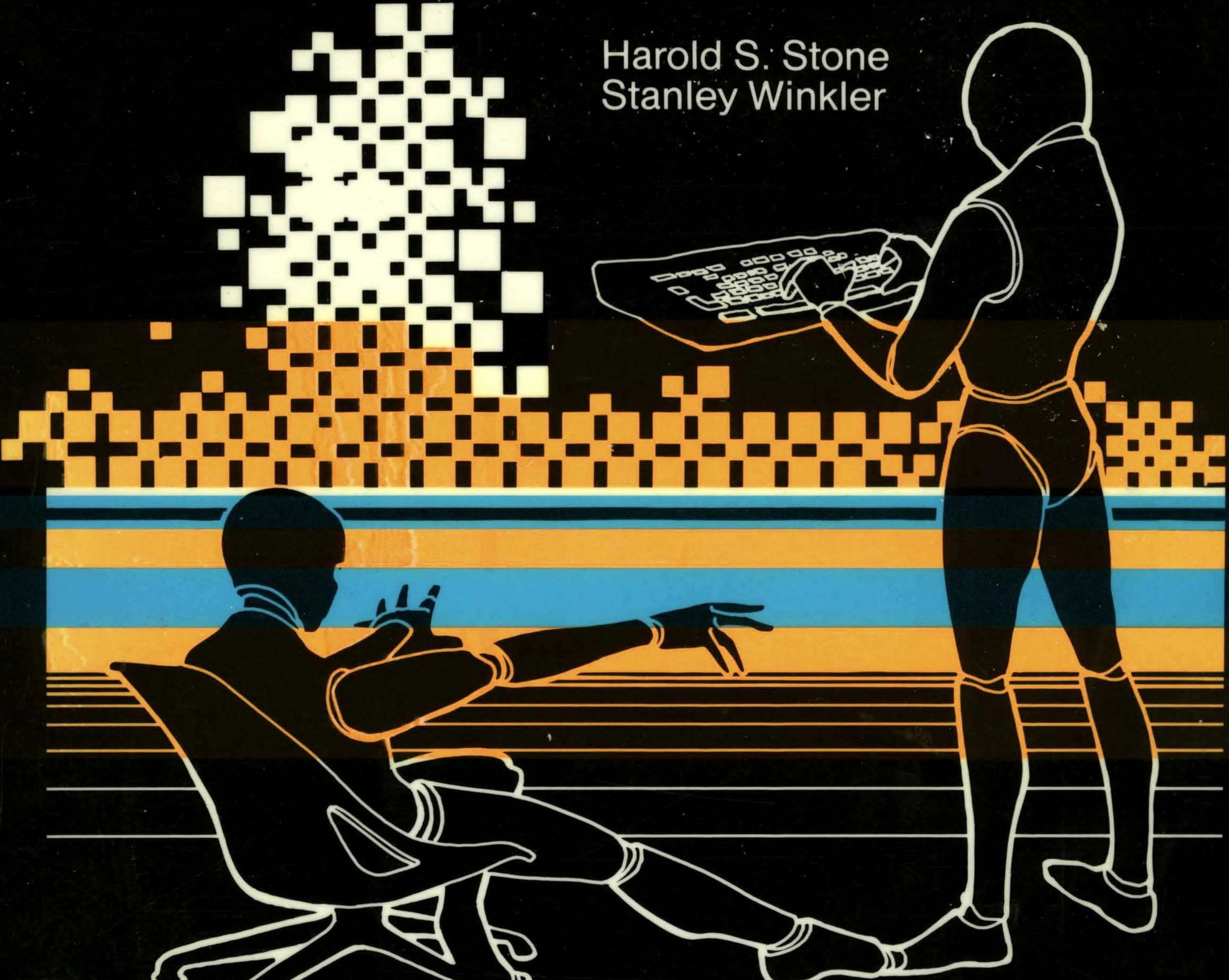
1986 Proceedings

FALL JOINT COMPUTER CONFERENCE

November 2-6, 1986—INFOMART®—Dallas, Texas
Sponsored by **ACM and Computer Society of the IEEE**



Harold S. Stone
Stanley Winkler

Computer Society Order Number 743
Library of Congress Number 86-81582
IEEE Catalog Number 86CH2345-7
ACM Order Number 401860
ISBN 0-8186-0743-2

THE COMPUTER SOCIETY OF THE IEEE
Association for Computing Machinery
IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE COMPUTER SOCIETY PRESS

1986 Proceedings
FALL JOINT COMPUTER CONFERENCE

November 2-6, 1986—INFOMART®—Dallas, Texas
Sponsored by **ACM and Computer Society of the IEEE**

Harold S. Stone, Proceedings Editor and Program Chairman
Stanley Winkler, Conference Chairman

Computer Society Order Number 743
Library of Congress Number 86-81582
IEEE Catalog Number 86CH2345-7
ACM Order Number 401860
ISBN 0-8186-0743-2

THE COMPUTER SOCIETY OF THE IEEE
Association for Computing Machinery
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS INC
COMPUTER SOCIETY PRESS

The papers appearing in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change, in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, IEEE Computer Society Press, or the Institute of Electrical and Electronics Engineers, Inc.

Published by IEEE Computer Society Press
1730 Massachusetts Avenue, N.W.
Washington, D.C. 20036-1903

COVER DESIGNED BY JACK I. BALLESTERO

IEEE Computer Society Order Number 743
Library of Congress Number 86-81582
IEEE Catalog Number 86CH2345-7
ACM Order Number 401860
ISBN 0-8186-0743-2 (paper)
ISBN 0-8186-4743-4 (microfiche)
ISBN 0-8186-8743-6 (case)

Prices (1986) ACM or IEEE Members: $60.00 prepaid
All others: $120 prepaid

Additional copies of the 1986 Proceedings may be ordered prepaid from:

ACM
Order Department
Post Office Box 64145
Baltimore, MD 21264

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

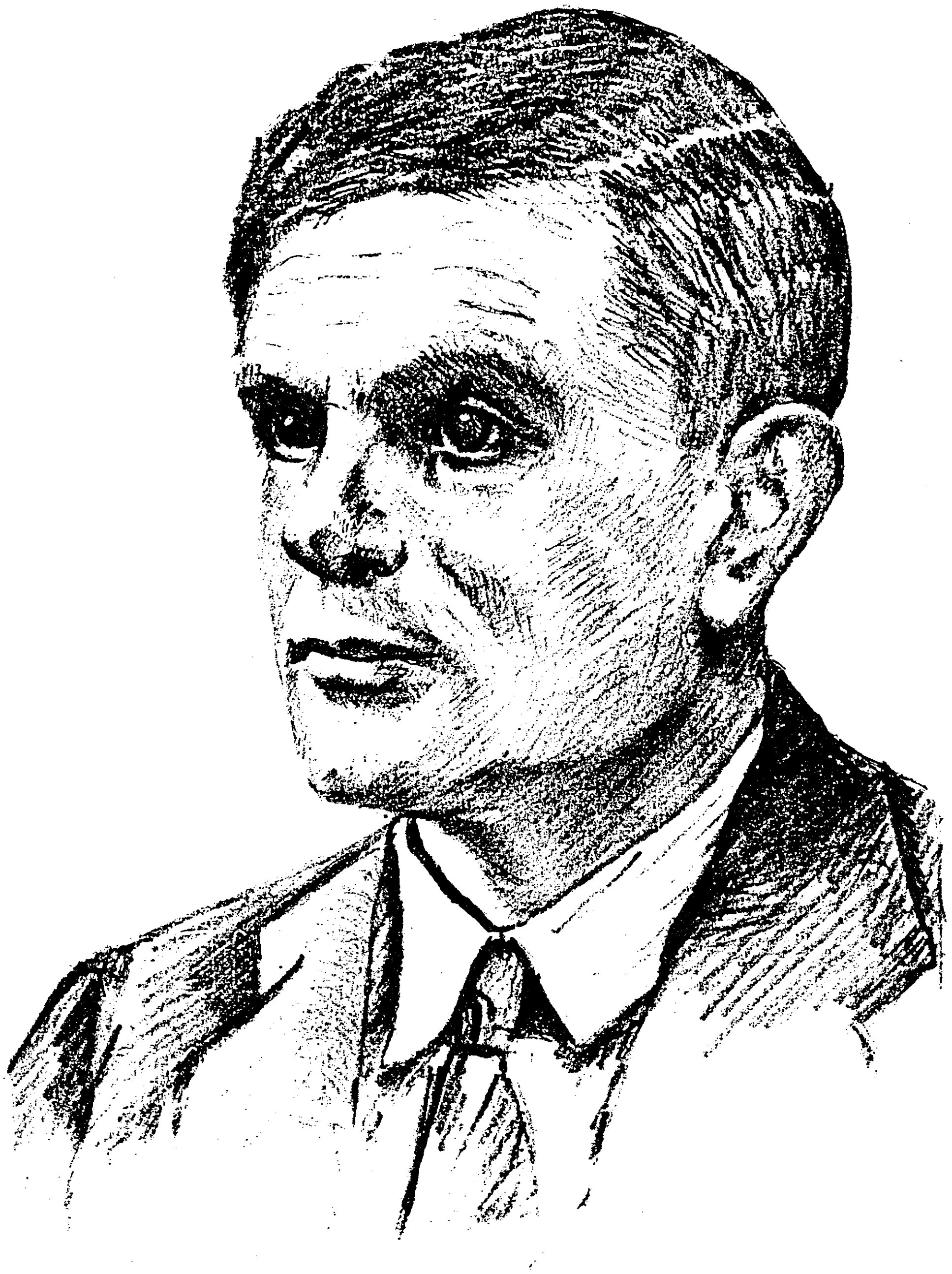
Computer Society of the IEEE
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

Computer Society of the IEEE
Ave. de la Tanche
1160 Brussels, Belgium

acm Association for Computing Machinery

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC
IEEE

THE COMPUTER SOCIETY OF THE IEEE



This 1986 edition of the proceedings of the Fall Joint Computer Conference, published on the 50th anniversary of the writing of the paper

"On Computable Numbers, with an Application to the Entscheidungs Problem,"

is dedicated to the memory of

ALAN M. TURING

who wrote:

"My contention is that machines can be constructed which will simulate the behaviour of the human mind very closely. They will make mistakes at times, and at times they may make new and very interesting statements, and on the whole the output of them will be worth attention to the same sort of extent as the output of a human mind."

How close are we to his vision today?

Conference Committee 1986 Fall Joint Computer Conference

Stanley Winkler
Conference Chairman

Harold S. Stone
Program

David C. Wood
Finance

Harry M. Kepner
Operations

Carla Elfeld
Samuel Fleming
M. Alex Harkins
Deanna Kirchoff
Charlotte Lin
Peter Maverick
George Weinreich

Dennis J. Frailey
Registration and Conference Advisor

Bruce Anderson
Publications

La Joyce Doran

William Lively
Resources

Rosetta L. Winkler
Conference Secretary

Joan W. Golden

Toni Shetler
Professional Development

Karen Duncan
Helen Takacs
Judi Paulos
Kermit Paulos

Beth Stinnette
Executive Education Coordinator

David M. Hyatt
Industrial Liaison

A.T. Landberg

Alex A.J. Hoffman
Society Liaison

Karen Duncan
Conference Management

Thomas A. D'Auria
Special Events

Adrian J. Basili
Technical Advisor

Tim Durkin
Exhibits

Cynthia Cegelski
Lynda Rosenthal

FJCC Steering Committee

Dick B. Simmons
Chairman

James H. Aylor

Roy L. Russo
President, Computer Society of the IEEE

William Habingreither
Wendy Chin

Seymour J. Wolfson
Past Chairman

James Iverson

Paul W. Abrahams
President, Association for Computing Machinery

James Adams
Pegotty Cooper

Program Committee

1986 Fall Joint Computer Conference

Harold S. Stone, **Program Chairman**
Maureen Ferraro, **Executive Program Coordinator**

Lionel Baldwin
Laszlo Belady
Domenico Ferrari
Norman Gibbs
C. Lee Giles
William Howden
Kai Hwang
Laurel Kaleda
Elaine Kant
John Kender
David Kincaid
Kenneth Kolence
Jerome Kurtzberg
Stephen Lavenberg
Tony Marsland
John Meyer
Cleve Moler
Ryoichi Mori
Kenji Naemura
Anil Nigam
Arthur Parry
Richard Paul
James Peterson
Paul Purdom
David Rine
Daniel Siewiorek
Jack Stankovic
John White
Robert Wilensky
Michael Willett
Michael Wozny

Referees

1986 Fall Joint Computer Conference

Abraham, J.A.
Adams, S.
Agarwal, R.
Agarwal, V.K.
Aida, H.
Aikens, J.
Allen, P.
Alonso, R.
Ammann, A.
Babaoglu, O.
Bard, Y.
Benjamin, J.
Bennett, B.T.
Berghel, H.
Birman, K.
Bose, P.
Brandenberg, J.
Brown, J.W.
Carey, M.
Carlsson, G.E.
Carter, W.C.
Chen, C.Y.
Cheng, V.
Chikayama, T.
Cook, J.
Cooper, E.
Copeland, G.
Cox, G.
Dana, C.
Dandapani, R.
David, R.
De Jong, K.
Driscoll, J.R.
Duchamp, D.
Eager, D.L.
Ezzat, A.
Freitas, R.
Ferrari, D.
Friedman, D.
Friedman, M.
Furthgott, D.
Gagliardi, R.
Gangopadhyay, D.
Garcia-Molina, H.
Gerasch, T.
Gilbert, R.
Goosen, H.
Goto, A.
Graham, D.
Hallmark, G.
Hansen, W.J.
Harandi, M.T.
Hassner, D.
Hayes, J.P.
Hellerstein, J.L.
Hillyer, B.
Homan, T.
Houghton, R.
Hutz, S.
Hwang, K.T.
Ibrahim, H.
Ihara, H.
Ishiguro, M.
Ishizuka, M.
Iyer, B.R.
Jha, N.
Johnson, M.S.
Kaneda, Y.
Kawaoka, T.
Kender, J.
King, R.
Kjell, B.
Kobayashi, S.
Kokubu, A.
Koo, R.
Korth, H.
Krishna, M.
Kung, H.T.
Kuper, G.
Kurose, J.
Landwehr, C.E.
Lee, I.
McCall, T.
McCluskey, E.J.
Malek, M.
Malhotra, A.
Masson, G.M.
Matsumoto, Y.
Melamed, A.
Mitra, D.
Mirchandaney, R.
Mok, A.
Moler, C.
Molina, H.G.
Morita, Y.
Mourad, S.
Mullarkey, P.
Muraoka, Y.
Murthy, S.
Nataraj, K.S.
Newman, T.
Neumann, P.G.
Ng, Y.W.
Nicola, V.
Nitta, K.
Norris, E.
Okada, Y.
Okugawa, S.
Omori, K.
O'Neill, D.M.
Patel, J.
Preparata, F.P.
Ramamritham, K.
Rashid, R.
Reddy, S.M.
Reeker, L.
Robinson, J.
Rosenberg, A.L.
Sabbah, D.
Sakai, K.
Sanders, W.H.
Sarin, S.
Sauer, C.H.
Scott, D.
Sekino, A.
Seth, S.C.
Sevick, K.C.
Sha, L.
Shibayama, E.
Shin, K.G.
Siewiorek, D.P.
Silberschatz, A.
Singh, A.
Smith, E.
Stankovic, J.
Stephenson, P.
Stewart, B.
Stone, H.S.
Sturgis, H.
Suwa, M.
Sylvester, J.
Tamaki, H.
Tanaka, H.
Tanaka, Y.
Toda, M.
Tokoro, M.
Tokuda, H.
Tomita, S.
Townsley, D.
Tripathi, S.
Tsujii, J.
Turn, R.
Valduriez, P.
Varman, P.J.
Vernon, M.K.
Waddle, V.
Wagner, K.
Wah, B.
Wang, H.
Wang, P.
Whang, K.Y.
White, J.R.
Willett, M.
Williamson, C.
Wolberg, G.
Wong, M.
Wozny, M.
Yemini, Y.
Yokota, M.
Yonezawa, A.
Young, H.
Yuba, T.
Zahorjan, J.
Zicari, R.

Table of Contents

EDUCATION ARENA

TRACK ED–1: New Technology in Education. Track Chair: Lionel Baldwin, President, National Technological University

Session 1 - Education by Satellite. Session Chair: Lionel Baldwin, National Technological University (1-B)

Session 2 - Computers in Education. Session Chair: Arthur Melmed, Consultant (4-B)

SOFTWARE SYSTEMS ARENA

TRACK SS-1: Software Engineering. Track Chair: Laszlo Belady, MCC

Session 1 - Object-Oriented Software. Session Co-Chairs: Clarence Ellis and Ted Biggertsaff, MCC (2-C)

SUPERCOMPUTING ARENA

TRACK SC-1: Parallel Computation. Track Chair: Kai Hwang, University of Southern California

Session 1 - Parallel Processing for AI. Session Chair: Kai Hwang, University of Southern California (1-D)

Session 2 - Parallel Algorithms for Supercomputing. Session Chair: Benjamin Wah, University of Illinois at Urbana-Champaign (2-D)

TRACK SC-3: Multiprocessors. Track Chair: Daniel Siewiorek, Carnegie Mellon University

Session 1 - Multiprocessors I. Session Chair: Zary Segall, Carnegie Mellon University (4-D)

Session 2 - Multiprocessors 2. Session Chair: Pat McGehearty, Microelectronics and Computer Technology Corporation (5-D)

Session 3 - High-Speed Techniques. Session Chair: William Brantley, IBM T. J. Watson Research Center (6-D)

TRACK SC-4: Optical Computing. Track Chair: C. Lee Giles, AFOSR/NE

Session 1 - Optical Computers. Session Chair: John Caulfield, University of Alabama (1-A)

Session 2 - New Directions in Optical Computing. Session Chair: C. Lee Giles, AFOSR/NE (2-A)

Session 3 - Optical Interconnections for Computing. Session Chair: John Neff, DARPA/DSO (3-A)

TRACK SC-5: Networks. Track Chair: Michael Willett, IBM Corporation

Session 1 - Implementing a Token-Ring Local Area Network, Session Chair: Michael Willett, IBM Corporation (4-A))

ALGORITHMS ARENA

TRACK AL-1: Artificial Intelligence Algorithms. Track Chair: Tony A. Marsland, University of Alberta

Session 1 - Computer Chess Techniques. Session Chair: Tony A. Marsland, University of Alberta (3-H)

TRACK AL-2: Numerical Methods. Track Chair: David R. Kincaid, University of Texas at Austin

Session 1 - Vector and Parallel Algorithms. Session Chair: David R. Kincaid, University of Texas at Austin (7-D)

Session 2 - Finite Differences, Finite Elements, and Grid Generation — A Tutorial. Session Chair: Linda J. Hayes, University of Texas at Austin(8-D)

TRACK AL-3: General Algorithms. Track Chair: Paul Purdom, Indiana University

Session 1 - Searching. Session Chair: Cynthia Brown, Northeastern University (7-I)

Session 2 - Data Structures. Session Chair: Michael Loui, University of Illinois (8-I)

TRACK MM-2: The State of the Art of Capacity Management in MVS Systems. Track Chair: Kenneth Kolence, Kolence Associates

Session 1 - Capacity Management 1. Session Chair: Kenneth Kolence, Kolence Associates (3-G)

Session 2 - Capacity Management 2. Session Chair: Kenneth Kolence, Kolence Associates (4-G)

COMPUTER DESIGN ARENA

TRACK CD-1: Fault-Tolerant Computing. Track Chair: John Meyer, University of Michigan

Session 2 - Evaluation. Session Chair: Kishor S. Trivedi, Duke University (5-E)

Session 3 - Testing. Session Chair: Edward J. McCluskey, Stanford University (6-E)

INTERNATIONAL DEVELOPMENT ARENA

TRACK ID-1: Computer Developments in Japan. Track Chair: Ryoichi Mori, University of Tsukuba, Track Secretariat: Mr. Kenji Naemura, NTT Electrical Communications Laboratories

Session 1 - Fifth Generation Computers I: Language Architecture. Session Chair: Koichi Furukawa, ICOT (4-F)

Session 2 - Fifth Generation Computers II: Applications. Session Chair: Koichi Furukawa, ICOT (5-F)

Session 3 - Advanced Microcomputer Developments. Session Chair: Iwao Morishita, University of Tokyo (6-F)

Session 4 - Supercomputing Systems. Session Chair: Yoshikuni Okada, Electrotechnical Laboratory (7-F)

Session 5 - Interworking Systems. Session Chair: Kenji Naemura, NTT Electrical Communications Laboratories (8-F)

OPERATING SYSTEMS AND DATA BASES ARENA

TRACK OSDB-1: Operating Systems. Track Chair: James Peterson, MCC

Session 1 - Applications of Petri-Nets. Session Chair: Paul Reynolds, University of Virginia (7-H)

Session 2 - Security and Protection in Computer Systems. Session Chair: James Peterson, MCC (8-H)

TRACK OSDB-2: Distributed Operating-Systems. Track Chair: Jack Stankovic, Carnegie Mellon University

Session 1 - Distributed-Operating Systems. Session Chair: Jack Stankovic, Carnegie Mellon University (5-H)

Session 2 - Distributed Databases. Session Chair: Hector Garcia-Molina, Princeton University (6-H)

TRACK OSDB-3: Data Bases. Track Chair: Anil Nigam, IBM T. J. Watson Research Center

Session 1 - Data Bases. Session Chair: Anil Nigam, IBM T. J. Watson Research Center (1-H)

The 1986 Fall Joint Computer Conference—
A Conference for the Profession

Stanley Winkler
Conference Chairman

A professional, technical conference is the term we have used to describe the 1986 Fall Joint Computer Conference (FJCC '86). It was our aim to prepare the finest conference of the decade. Sponsored by two great professional computer associations, the Association for Computing Machinery (ACM) and the Computer Society of the IEEE, the FJCC was conceived as a conference-for-all-members. Specifically, the FJCC combines two meetings of the Societies customarily held in the fall season: The ACM National Conference and the COMPCON Fall meeting of the Computer Society of the IEEE. The ACM Council and the Computer Society Governing Board will meet during FJCC '86, as will various Boards and Committees, following the tradition of conducting Society business during the annual Fall Meeting.

Professional, technical conferences in the United States have a long and honorable history that can be traced back to the Mechanics Institutes of the 1800s. These Institute meetings, made necessary by the rapid technical advances of the Industrial Revolution, allowed professionals in the field of engineering to gather together and share problems and experiences. As the workforce became more specialized, it was no longer adequate for industry to rely on the transfer of technical knowledge from father to son or from master to apprentice. Today, in the computer field, professional conferences remain an important, if not the most important, means of exchange of ideas, information, and knowledge within the profession. While classroom education can provide the basis for entry into the profession, the essential continuing education is best acquired by interaction with one's peers. This interaction among peers is at once the function and the *raison d'être* of professional, technical conferences.

Of course, times have changed, and our computing profession has changed with it. Thirty-five years ago at the first joint meeting, in Philadelphia, the topics discussed encompassed most of the field as it was known then. That completeness would be impossible to accomplish today. Nonetheless, it seemed important to try, during FJCC '86, to provide a broad-based review of the most significant topics that confront the computing profession and industry. The salient feature of the 1986 Fall Joint Computer Conference is its broad-based nature. Equally important is the fact that, broad-based though it is, FJCC '86 presents full in-depth discussions of the topics selected. The technical program for FJCC '86, so ably developed under the leadership of Harold S. Stone, is a cornucopia of technical delights prepared by experts for their fellow practitioners of the art of computing.

In his introduction to the FJCC '86 technical program (in this volume), Harold S. Stone refers to FJCC '86 as "a new beginning." Since by definition all beginnings are new, one cannot quarrel with this phrase. I think, however, that the French saying "the more things change, the more they remain the same" also applies. I had the pleasant and comfortable feeling, on first seeing the finished FJCC '86 technical program, that the program is a return to the best traditions of the past. It is, indeed, a thoroughly modern program, broad-based and in full detail. It displays, without compromise, the current state-of-the-art. A casual look at the Conference-at-a-Glance confirms this assertion. But the technical program of the 1986 Fall Joint Computer Conference does more than provide a static snapshot of the world of computing today. It describes and represents the directions that the leaders of the computing profession are taking.

This illumination of the directions that the computing profession is taking is the most important function of an FJCC. Significantly, this illumination is not a prediction of the future gained by gazing into a crystal ball, or by the reading of tea leaves. It is, in fact, a self-fulfilling prophecy. The participants in FJCC '86, the speakers, panelists, discussants, and attendees, are not just talking about professional leadership in the 1990s—they are making it happen. They can make it happen because "they" are the leaders, the top, key people in the profession and in the industry. The content of

FJCC '86 in its breadth and depth is outstanding—probably exceeding that of any conference held in the last decade. This unprecedented breadth and depth is achieved through the nine conferences that are held simultaneously during FJCC '86.

Each of these nine conferences is a front-line, major event of its own. The nine conferences are: Artificial Intelligence, Supercomputing, Software Systems, Algorithms, Modeling and Measurement, Computer Design, Computer Developments in Japan, Operating Systems and Data Bases, and Education.

The fact that these nine conferences are going on at the same time is a significant added dimension to the FJCC '86 experience. Not only can FJCC participants meet the experts in their own specialty, but they can interact with the leaders in other specialties. This provides an enrichment for the individual, and, at the same time, is very good for the profession. The interpersonal communication among specialists in various fields of computer science and engineering stimulates thinking and creativity. This "cross-cultural" communication induces the propagation of ideas and concepts from one field into another, adding robustness and vibrancy to our profession.

This Proceedings provides a permanent record of the technical papers presented at the Conference. As such, it is a valuable addition to the shelves of our personal and organizational libraries. It does not, however, capture the other dynamic and exciting aspects of FJCC '86; absent are the discussions following each presentation of a technical paper, the poster sessions where last minute ideas are put forth, and the conversations in the hallways and lounges during coffee breaks and after sessions. Also not reflected in the Proceedings are the special events such as the world class Chess Tournaments—the 17th North American Computer Chess Championship and the 6th World Microcomputer Chess Championship.

The FJCC was designed to provide a complete educational experience. Complementing the technical program, Toni Shetler and her committee arranged an unparalleled Professional Education Program (PEP) and a very interesting Exhibitor Technical Forum. The Professional Education Program, which took place during the first two days of the Conference, gave attendees the opportunity to learn new skills and to sharpen old ones. There were one- and two-day courses, many of the hands-on variety. The Exhibitor Technical Forums provided the opportunity for vendors to discuss and explain the technology imbedded in their principal products. All of these activities were part of the effort to attain the objective of FJCC '86: to expand the professional horizons and capabilities of the conference attendees.

It was my privilege to chair a Conference Committee of capable, dedicated individuals, who contributed their time and effort in the service of the profession by creating the FJCC '86. They are listed elsewhere in this volume. To each of these friends I want to say, "Thank you." There were many others who helped us and to all of them I want to express my sincere appreciation. And finally, I want to thank the participants because, in the final analysis, it is they who are the Conference.

The 1986 Fall Joint Computer Conference—A New Beginning

Harold S. Stone
Program Chairman

The Conference Role

The 1986 Fall Joint Computer Conference is a conference of the future and of the past. The future is embodied by the conference theme — Exploring the Knowledge-Based Society — and the past by its popular predecessor conference of the 1960s. In looking forward, the conference offers technical papers, panels, and tutorials to cover the topical areas that form the technology base for the next decade. Such areas as Artificial Intelligence, supercomputers, design automation, computer graphics, and networks are among the topics of special attention. In looking backward to the FJCC's of 20 years ago, the vast changes in the discipline are evident, which underscores the importance of providing state-of-the-art information in subject areas destined to form the core of the field in coming years.

Consider the change in the hardware technology from the 1960s to the 1980s. The supercomputers of the 1960s are the micros and minis of the 1980s. Who would believe then that such computing power would be available on the desk tops of virtually every researcher, analyst, programmer, and student. The workhorse of a typical scientific installation in 1960 had 128K of 10 microsecond memory. A low-cost microprocessor of 1986 has 256K of 150 nanosecond memory. A supercomputer of 1965 had 2M bytes of 1 microsecond memory. A typical workstation of 1986 has 4M bytes of 120 nanosecond memory. The dramatic improvements in cost and performance made possible through VLSI technology were truly unimaginable in the days of the former FJCC.

The impact of such changes on the computing field are not yet fully assimilated, but the trends are clear. In days of expensive hardware, software was viewed to be inexpensive, more or less by default. Where hardware was too costly to commit to a special-purpose job, or where the functional requirements were too vague to lock into hardware, the often-used solution was to build hardware to do approximately what was desired, and to leave the final tailoring to software. That is, constrain the major cost by freezing the hardware at some stage, and fill in the remainder of the implementation with "inexpensive" software.

All too often this approach had costly surprises in store for the system developers. With hardware frozen, the only freedom available to make such systems work was in software development, enhancements, and maintenance. Software costs climbed continually during the life of systems since software costs never ceased. In long-lived systems, software costs eventually dwarfed the hardware investment. Moreover, the inherent flexibility attributed to software became a myth, as changes to existing large-scale software became substantially more difficult and eventually impossible to implement. On the other hand, hardware became far more flexible, as each new generation of computers was succeeded by faster and less expensive generations, each upward compatible with its predecessors. The relative flexibility of hardware and software as viewed in the 1960s had turned upside down by the 1970s.

To improve the software situation, substantial efforts in high-level languages increased programmer productivity, but, productivity as measured in lines of code, failed to attain high multiples that was once viewed necessary to prevent the massive cost of software production from swamping the industry. Who would have predicted how this view would change after widespread introduction of the microprocessor? With millions of potential users instead of hundreds, the sales price of software could be kept relatively low per user, in spite of high costs for development. Moreover, as new microprocessors were introduced, it was totally impractical to rewrite a new software base for each new offering. Survivability of the microprocessor was largely tied to the ability to move a complete software base to the microprocessor, and this in turn created the market for portable software. With relatively little effort, it is possible today to move a complete operating system plus compilers, editors, and supporting tools to a totally new microprocessor with a unique instruction set.

Instead of writing new code that reinvents old software, the field has developed techniques to reuse software that does the job. The net result is that most software has become inexpensive on a per-user basis. Expensive software still exists, however, where user communities are small, and where techniques are embodied in software for the first time.

Having reflected on the changes in computing technology from when the FJCC was at its former height to the present time when the FJCC has been reborn, can we conceive of the changes that will take place in the next several years? For example, what will happen to programming as a profession? Will it be a profession that supports a population that seems to grow exponentially? Or will there be a limit to that growth that holds the population constant? Or will the numbers collapse? All three of these models are possible, depending on how the field places computer power in the hands of the user. The exponential growth models the growth in the number of computers, and is a model in which each computer system requires individual programming. The constant-population model is a model in which a fixed pool of people is able to supply a growing population of machines, and is probably an accurate reflection of the industry in the next few years as portable software becomes more widely used in place of specially tailored software. The last model in which fewer programmers are able to supply a growing pool of machines is one in which a relatively few "super" systems created by highly skilled programming teams account for large fractions of software use. The remaining software can be supplied by a much smaller pool of programmers under the first or second models.

The potential for this last model clearly exists today. If the model eventually becomes reality, the industry will be far different from the one we know. Are we ready for that event?

The 1986 FJCC is a conference where we can examine the recent trends, hear the projections, discuss the possibilities with the experts in each field, and then prepare for the future. The key to the FJCC is the technical focus. We must be an informed profession, and we must look forward in technology.

The growth of the computer profession has brought diversity, and the diversity has splintered the profession a hundred ways into the SIGs and TCs that form the technical leadership of the disciplines within the profession. The diversity has created journals, newsletters, conferences, and workshops with single themes directed to the experts in the various areas. Each of these activities has had its positive benefits within the narrow focus area, but the single-focus activities cannot provide for advances that require the synthesis of ideas from multiple disciplines. The FJCC is, by design, a multiple-focus conference. Its purpose is to bring together the experts across a range of disciplines so that the mix of ideas can provide impetus for new projects attaining new plateaus that are not readily achievable within any one discipline.

Consider, for example, four different ways of representing information. Individually, we might have information represented as

- text,
- voice,
- graphics, or
- data base.

For each representation we can build a discipline that deals with that representation exclusively, and we create such disciplines as publishing, telephony, computer graphics, and on-line information utilities. Now reconsider the four representations, and consider what happens when you join any two together. A whole new discipline is created. If you join text and voice, you obtain the voice-operated typewriter, voice-data communications networks, or spoken output from written text. If you combine graphics and text, you create computer publishing, intelligent copiers, and electronic encyclopedias. But there is no need to focus on just the four data representations. Pick any collection of special-interest areas and consider what new challenges can be formed by combining any two or three areas. This gives an inkling of what can happen when you bring together active thinkers from a variety of areas and let the pot boil. This is the FJCC.

Conference Management

How was the technical conference put together? With a new conference we have no history from last year, no experienced program committee to draw upon, and no expectant audience ready to submit materials to the annual gathering of the clan. This conference was mounted as thirty tracks in the major areas of computer science and engineering. Each track chair had the charge to create a track that best illuminated the area, whether through solicited refereed papers, panels, or through invited papers from recognized innovators. The conference call also produced papers in abundance, and these were distributed to the tracks for refereeing according to the subject matter of the papers. In all, 248 submissions were processed for the conference. Of these, approximately 150 were unsolicited and went through the conference refereeing processing. Roughly one third of these papers were accepted for the proceedings. The remainder of the papers in the proceedings were generated by the collective energy of the track chairs. These were treated either as invited papers not subjected to an external review or as solicited papers that were reviewed and possibly modified prior to publication in the proceedings. A number of solicited papers were dropped from consideration after the reviewing process, but the exact numbers of such rejections is not known because such papers were not coordinated centrally.

With a large fraction of solicited papers at this conference, there became a potential for abuse of the refereeing system. To assure high technical quality, it is essential that papers receive fair, independent assessments by competent reviewers. Invited papers were not refereed, so the basis for invitation had to be on the basis of past performance. Chairs of sessions have been selected for their contributions, and they themselves are candidates for invited papers in their own sessions. However, for ethical reasons, no person at this conference was permitted to accept a paper in which that person or a close colleague was among the authors. In each such instance in this proceedings, the paper underwent independent review and was accepted by a party other than the session chair.

The success of the program is due entirely to the efforts of the program committee members (listed on page vii). Finally, I thank each of the following for their contributions:

Sheldon Akers
Lionel V. Baldwin
Laszlo Belady
Ted Biggerstaff
Barry Boehm
Pradip Bose
Cynthia Brown
William Brantley
Luis Felipe Cabrera
John Caulfield
Lori Clarke
Paul Cohen
Kenneth De Jong
Louis Doctor
Clarence Ellis
Richard Fairley
Domenico Ferrari
Henry Fuchs
Koichi Furukawa
Hector Garcia-Molina
Norman Gibbs
C. Lee Giles
Linda J. Hayes
Philip J. Hayes
Alex Hoffman
William G. Hooper
William Howden
Kai Hwang
Laurel Kaleda
Elaine Kant
John Kender
Peter Kessler
David R. Kincaid
Chandra M. R. Kintala
Ken Kolence
Jerome Kurtzberg
Stephen Lavenberg
Tomas Lozano-Perez
Michael Loui

T. A. Marsland
Nancy Martin
Edward J. McCluskey
Pat McGehearty
Arthur S. Melmed
John Meyer
Cleve Moler
Robert Morgan
Ryoichi Mori
Iwao Morishita
Kenji Naemura
John Neff
Anil Nigam
Yoshikuni Okada
Arthur Parry
Richard Paul
James Peterson
Dhiraj K. Pradhan
Paul Purdom
Paul Reynolds
Harriett Rigas
David Rine
Larry Ruzzo
Zary Segall
Daniel Siewiorek
S. E. Smith
Jack Stankovic
Russell Taylor
Timothy N. Trick
Kishor Trivedi
Wing Toy
Andries van Dam
Benjamin Wah
Richard L. Wexelblat
John R. White
Robert Wilensky
Michael Willett
Michael Wozny

In a list this long, the nature of the contributions varies widely across the list. All contributions have been important, and each party noted above deserves their share of credit in the success of the FJCC. However, some contributions deserve special mention. I greatly appreciate the work of Ryoichi Mori and Kenji Naemura for producing the papers in the International Developments Arena. Les Belady, Jim Peterson, Jerry Kurtzberg, John Meyer, David Rine, Steve Lavenberg, Jack Stankovic, and Ken Kolence each produced top quality tracks through their resourcefulness and continued efforts. Maureen Ferraro, Executive Program Coordinator, provided the glue that held the team and the participants together. As the conference grew from embryo into infancy and then maturity, Maureen was there with the detailed work to guide the development. Tracking the papers and referees, managing the proceedings, and making personal contacts to assure timely responses were typical of the many tasks she tackled. The magnitude of the job could exceed the capacity of many computers I have known, and only occasionally challenged but never exceeded her capacity to get the job done.

Finally, we come to the referees—anonymous to the authors—recognized here in the proceedings. We gratefully acknowledge the role played by the referees (listed on page vii). No conference can succeed without the wisdom of careful reviews to assure the quality and accuracy of the published material.

Conference at a Glance

PLENARY SESSION
Industrial Keynoter:H.Ross Perot
Electronic Data Systems
Tuesday, 8: 30-9: 30

	Grand Ballroom B Room A	Grand Ballroom D Room B	Grand Ballroom A Room C	Governor's Lecture Hall Room D	Senator's Lecture Hall Room E	Sapphire Room Room F	Topaz Room Room G	Thornton Room Room H	Grand Ballroom E Room I
1 Tues. 04 Nov. 10: 00 - 12: 00	SC4-1 Caulfield: Optical Computers	ED1-1 Baldwin: Education by Satellite	CI1-1 Hoffman: Legal Professional Concerns	SC1-1 Hwang: Parallel Processing for AI	CD2-1 Akers: VLSI Design Automation		MM1-1 Lavenberg: Performance Studies	OSDB3-1 Nigam: Databases	AI5-1 Wilensky: User Interfaces
2 1: 30 - 3: 30	SC4-2 Giles: Optical Computing Directions	SS4-1 Van Dam: Hypertext	SS1-1 Ellis: Object-Oriented Software	SC1-2 Wah: Parallel Algorithms	CD2-2 Trick: VLSI Research in Academia	ED2-1 Fairley: Software Engineering Education	MM1-2 Lavenberg: Performance-Modeling Methods	AI1-1 Martin: Design Issues and Practice	AI5-2 Granger: Natural Language Processing
3 3: 45 - 5: 15	SC4-3 Neff: Optical Inter-Connections	MM1-3 Lavenberg: Performance-Modeling Workstations	SS1-2 Belady: Software-Design Modes	SC2-1 Moler: Hypercube Computers	CD2-3 Pradhan: VLSI Fault-Tolerant Goals	ED2-2 Smith: Corporate Software Engineering	MM2-1 Kolence: Capacity-Management 1	AL1-1 Marsland: Computer Chess Techniques	AI5-3 Hayes: Natural Language Panel

PLENARY SESSION
Keynoter: Kenneth Wilson, Nobel Laureate
Cornell University, Dept. of Physics

Keynoter: C. Gordon Bell
National Science Foundation
Wednesday, 8: 30-9: 30

	Room A	Room B	Room C	Room D	Room E	Room F	Room G	Room H	Room I
4 Wed. 05 Nov. 10: 00 - 12: 00	SC5-1 Willett: Token-Ring Local Area Networks	ED1-2 Melmed: Computers in Education	CI1-2 Kaleda: Computer Standards	SC3-1 Segall: Multi-Processors 1	CD1-1 Toy: Fault-Tolerant Applications	ID1-1 Furukawa: Fifth-Generation Computers 1	MM2-2 Kolence: Capacity Management 2	SS4-2 Boehm: Software Development Environment	AI2-1 Kender: Computer Vision 1
5 1: 30 - 3: 30	SC5-2 Willett: Token-ring Networks	SS4-3 Kintala: Integrated Programming Environments	AI4-1 Rine: Engineering Rule-Based Systems	SC3-2 McGehearty: Multi-Processors 2	CD1-2 Trivedi: Reliability Evaluation	ID1-2 Furukawa: Fifth-Generation Computers 2	MM2-3 Ferrari: Insularity of Performance Evaluation	OSDB2-1 Stankovic: Distributed Operating Systems	AI2-2 Kender: Computer Vision 2
6 3: 45 - 5: 15	SC5-3 Hooper: Integration of Voice and Data	SS4-4 Kessler: Issues in Code Generation	AI4-2 Wexelblat: Knowledge Engineering	SC3-3 Brantley: High-Speed Techniques	CD1-3 McCluskey: Testing	ID1-3 Morishita: Micro-Computer Developments		OSDB2-2 Garcia: Distributed Databases	AI3-1 Kanade: Robot Perception

PLENARY SESSION
Turing Lecture
1986 ACM Turing Award Winner
Thursday, 8: 30-9: 30

	Room A	Room B	Room C	Room D	Room E	Room F	Room G	Room H	Room I
7 Thurs. 06 Nov. 10: 00 - 12: 00	SS3-1 Clarke: Problems in Program Testing	SS4-5 White: Programming Languages	AI4-3 Cabrera: UNIX: Wave of the Past?	AL2-1 Kincaid: Vector and Parallel Algorithms	CD2-4 Bose: Expert Systems Design/Test	ID1-4 Okada: Super Computing Systems	AI3-2 Lozano-Perez Task-Level Robot Programming	OSDB1-1 Reynolds: Application of Petri-Nets	AL3-1 Brown: Searching
8 1: 30 - 3: 30	SC5-4 Kumar: Network Management	CD2-5 Hsia: Design Languages	AI4-3 Kant: Rule-Based: Models and Applications	AL2-2 Hayes: Finite-Element Methods	CD3-1 Doctor: Computer Geometry	ID1-5 Naemura: Inter-Working Systems	AI3-3 Taylor: Real-Time Robot Programming	OSDB1-2 Peterson: Security/ Protection Systems	AL3-2 Loui: Data Structures
9 3: 45 - 5: 15		SS1-3 Morgan: Application of ADA	AI4-4 Bottegal: Logic Programming		CD3-2 Wozny: Computer Graphics Standards				AL3-3 Ruzzo: Optimization Techniques

EDUCATION ARENA

New Technology in Education

TRACK CHAIR: Dr. Lionel Baldwin
National Technological University

Software Engineering Education

TRACK CHAIR: Prof. Norman Gibbs
Carnegie Mellon University

AMCEE PROGRAMMING FOR COMPUTER PROFESSIONALS

John T. Fitch
Associate Director

Association for Media-based Continuing Education for Engineers

ABSTRACT

The Association for Media-based Continuing Education for Engineers (AMCEE) is a consortium of 33 engineering universities which provides off-campus education via television and videotape. Clients are engineers, industrial scientists, and technical managers in business, industry, and government. AMCEE operates a satellite delivery system, offering six hours a day, five days a week of non-credit continuing education courses. The bulk of the courses, however, are delivered on videocassettes, accompanied by study guides and textbooks. The majority of these "short courses" are in computer and communication related subjects.

INTRODUCTION

In a session on "Technical Education by Satellite," it seems appropriate to talk specifically about continuing education for computer professionals, because the AMCEE programs aimed at that audience have been among its most successful. But first it might be worth a brief digression to explain AMCEE itself and its mission.

AMCEE is an acronym for the Association for Media-based Continuing Education for Engineers. It is a non-profit, tax exempt consortium of, at present, thirty-three engineering universities. What these schools have in common is programs of off-campus graduate and/or continuing education using television and/or videotape. Most of the members offer master's degree programs and continuing education short courses to practicing engineers, industrial scientists, and technical managers who take their coursework at the job site rather than on campus. The medium that connects the campus classroom with the industrial site or government laboratory is either "live" television using Instructional Television Fixed Service (ITFS) microwave channels, or a set of courier-delivered videocassettes. Most of these materials are in what is commonly called "candid classroom" format, i.e. the courses are broadcast or videotaped as they are being taught to a class of on-campus students. Cameras are fixed to the walls and ceiling of the classroom, and the equipment is often operated by students. Thus, the production costs are marginal.

Bringing the classroom to the student offers convenience and flexibility, as well as significant cost- and time-effectiveness. And all without sacrificing academic quality. Several studies have shown that off-campus students do as well or better than their on-campus counterparts taking the same course.

With a view toward providing better quality materials to "increase the national effectiveness of continuing education for engineers," twelve universities joined together in 1976 to form AMCEE. Its headquarters were placed on the campus of one of its members, the Georgia Institute of Technology in Atlanta. The idea was that an association of schools would make it economically feasible to develop studio produced videocassettes and collateral printed materials for a national rather than a regional clientele. Thus, 1986 is the tenth anniversary of the consortium which has since nearly tripled in size in terms of membership -- and increased by an order of magnitude its services to business, industry, and government. Table 1 is a list of the present AMCEE membership.

Table 1 - Members of AMCEE

Auburn University
Colorado State University
Georgia Institute of Technology
GMI Engineering and Management Institute
Illinois Institute of Technology
Iowa State University
Massachusetts Institute of Technology
Michigan Technological University
North Carolina State University
Northeastern University
Oklahoma State University
Polytechnic University
Purdue University
Southern Methodist University
Stanford University
University of Alaska
University of Arizona
University of Florida
University of Idaho
University of Illinois at Urbana-Champaign
University of Kentucky
University of Maryland
University of Massachusetts
University of Michigan
University of Minnesota
University of South Carolina
University of Southern California
University of Washington

During these first ten years, the medium of delivery has been the videocassette, there being no economical national equivalent of the local, live ITFS. And videocassettes still offer scheduling flexibility that prompts many clients to opt for them, even when live television is also available. Participants who miss a session because of travel or pressing business can catch up with their colleagues by watching the videocassettes on their own.

Today, with over 90 per cent of the engineering universities who offer media-based off-campus education as members, AMCEE publishes an annual catalog listing some 500 video courses in 16 engineering and science disciplines from its 33 members. These disciplines cover all the traditional ones from aeronautical engineering to mathematics as well as a number of interdisciplinary and management subjects. During the last fiscal year, AMCEE logged over 1500 orders from some 850 clients and reached an estimated 22,000 individual participants.

SATELLITE DELIVERY

By 1985 the economics of satellite television had made it feasible for AMCEE to re-evaluate the possibility of live television delivery for its programs. At the same time, a subset of AMCEE member universities formed a sister organization, the National Technological University (NTU) to offer an accredited master's degree program on a national basis. The two organizations agreed to share a transponder on a recently launched satellite in the "Ku" band. Unlike the more widely used "C" band, used by the cable and movie channel companies, the Ku band is used primarily for business communications. Several AMCEE clients were already equipped to receive satellite programs in this higher frequency part of the spectrum. Furthermore, because of the higher power of the particular satellite selected (G-Star I), it was possible to split the bandwidth and power of a single transponder and still provide reasonably good signals to carefully specified and installed receivers -- thus further improving the economics of this new delivery mode.

In September 1985, the two organizations inaugurated the "AMCEE/NTU Satellite Network," NTU providing candid classroom courses for credit towards a master's degree, AMCEE providing non-credit short courses for continuing education. Currently, there are five origination sites where "earth stations" are located that can transmit classes up to the satellite. These are located at Colorado State University; the Universities of Massachusetts, Maryland, Maryland, and South Carolina; and the Georgia Institute of Technology. AMCEE, because it is located at Georgia Tech, relies heavily on the "up-link" there. Each weekday from 11:00 a.m. to 5:00 p.m. Eastern time, AMCEE broadcasts a variety of short courses and seminars, a total of 1500 hours of instruction per year. Many of the courses are prerecorded on videocassette, but some of them are live with participant interaction via telephone with the instructor. These are on engineering and technical management topics, with the most popular being those having to do with computers and communications.

Organizations receiving the telecourses install their own "down-links" and pay for the service through a network registration fee and individual course registration fees. Under license, they can videotape transmissions for delayed use to provide them with the scheduling flexibility available through videocassette delivery. In general, the pricing is similar for the two modes of delivery, though the startup cost for participation in the satellite network clearly makes it more expensive, initially. So, one might ask why a company might choose satellite over videocassette delivery. There is more than one answer. For one thing, it is simply a lot easier to walk into a room and turn on the TV than it is to plan far enough ahead to order a set of videocassettes, take delivery of them, keep them safely, and return them after they've been used. And when the program is a "live" event, there is the added value -- gratification, even excitement, if you will -- in being able to call up the instructor and ask a question that applies to your own particular situation. As of this writing (Spring 1986) there were approximately 60 down-link receiving sites scattered across the country.

But to concentrate on the difference in cost between videotape and television delivery systems obscures a more important point: the dominant cost of education is the participating engineer's time. Anything, then, that can be done to conserve that time, by, for example, eliminating the need to commute to a college campus undoubtedly outweighs the higher cost of media-based delivery.

We have experienced minimal start-up problems with the network, most having to do with the novel split transponder and the necessity for careful purchase, installation, and maintenance of down-link equipment. Training directors regularly call the AMCEE office with their lists of registrants for a variety of courses. AMCEE, in turn, coordinates the shipment of printed materials -- study guides and/or textbooks -- to the receiving sites (usually!) in time for the start of each course.

SPECIAL PROGRAMS

Occasionally (currently about once every other month) AMCEE opens up its satellite network and transmits a short seminar or longer telecourse on a C-band transponder as well as on its Ku-band network. This means that any organization with access to a C-band down-link can receive the programs, and even if they do not have access to such equipment, there are many areas around the country where participants can come to an AMCEE member campus to watch the program. Furthermore, the Hewlett-Packard Corporation has very generously opened many of its plants and offices to outsiders who wish to participate. These special "open network" broadcasts receive far wider promotion than is given the rest of the schedule. Instead of just being listed in the AMCEE publications, the **Monitor** and the **Uplink**, separate brochures are published for each program and mailed to several thousand prospective participants.

These special broadcasts are usually reserved for live events such as an April 1986 program on "Computer Communications and Networking: A Technology Forecast" or a combination of videotape and live broadcasts such as a June 1986 program on "Microcomputer Software for Project Management." The former -- the all "live" programs -- are usually videotaped as they occur and, if the quality of the programs is high and the content likely to have a reasonable shelf life, these videotapes are then advertised for rental and sale. Although these cassettes do not have the advantage of the telephone interaction, they can, nevertheless, be useful to those not able to watch the satellite broadcast.

The latter type of program -- the combination of videotape and live -- is a more frequently followed model. Here, a set of videotaped lectures, often made just prior to the broadcast, are used as the backbone of the presentation. We find that, by videotaping the lectures, the instructor is subject to less pressure, errors can be corrected, and there is time between lectures to collect thoughts and materials. However, the instructor remains in the studio after the taping is completed (or returns at a later date) for the satellite broadcast. Then, after each videotape is run, the instructor appears "live" on camera and takes telephone questions from the participants. The schedule is arranged so that any time not used for questioning serves as a brief intermission before the next videotaped lecture.

We are still developing this mode of operations -- the whole concept of dual-band satellite programs is still relatively new for us -- but the scheme that appears to be emerging is one in which we produce the videotapes at a television studio at the University of Maryland (where we produce most of the AMCEE videotaped courses) and then up-link the actual broadcast from one of the "candid classrooms" across campus at the engineering school, which has a Ku-band earth station, or we take the tapes to the University of South Carolina (which has, in place, both Ku-band and C-band earth stations). Exotic as all this may sound when compared with offering a continuing education program at a local college campus, it is not, marginally, a very costly operation (because the studios and up-links are already "there"). Therefore, it does not require a very large audience to break even (on the order of a few hundred people).

COMPUTERS AND COMMUNICATIONS

As indicated earlier, among the most popular courses delivered over the satellite network are those having to with computers and/or communications. (This is also true to a lesser extent for our videocassette distribution, but the videocassette audience is greater and more diversified; hence a program on metallurgy might do well on tape but fail on the network where the clients are still primarily high-tech companies with a heavy concentration in the computer field.) Table 2 lists the programs offered thus far on the network on computer-related subjects.

Table 2 - Computer Related Courses on the AMCEE/NTU Satellite Network

Applied Kalman Filtering
Communication Networks
Computer Communications
Computer Communications & Networking
DDN & DOD Protocol Standards
Distributed Processor Communication Arch.
Distributed Telecommunication Networks
Effective Use of Small Computers
Fortran 77
Fundamentals of Data Communications
Gallium Arsenide Integrated Circuits
IEEE 802: Local Network Standards
Integrated Services Digital Network
Interactive Computer Graphics
Kalman Filtering
Lisp at Work, Parts 1, 2, & 3
Local Area Networks
Local Network Technology & Selection
Microcomputer Software for Project Management
Microprocessor Interfacing
Packet Switching Networks
Pascal, Part 1
Principles of Modern Software Engineering
Relational Database
Robotics: A Tutorial in Four Parts
16-Bit Microprocessor Programming
Software Management for Small Computers
Software Project Management
Telecommunications & The Computer
Vector Processors & Mini Computers

Neglecting individual lectures -- usually on management skills topics, these 30 courses represent approximately half of all the courses offered on the network during the period from early September through mid May, 1986. In other words, all other disciplines made up the other fifty percent.

CONCLUSIONS

AMCEE currently broadcasts its non-credit courses six hours a day, five days a week on the share AMCEE/NTU satellite network in the Ku band, for a total commitment of 1500 hours a year. In addition, AMCEE offers six or more "special" events each year on both the Ku-band network and on a C-band transponder. These special programs include both three or four hour seminars as well as two and three day short courses. Those tentatively scheduled through June of 1987 are:

	1986
November	Computer Organization & Architecture

1987	
January	Database Management Software for Personal Computers
March	Office Automation
May	Computer & Network Security
June	Microcomputer Software for Project Management: An Update

For these "specials," it is clear that the subject matter is completely devoted to computer related subjects. Should this mode continue to be as successful as it has proved thus far, AMCEE will undoubtedly increase the frequency of these open-circuit transmissions. At the same time, the Ku-band network continues to grow as more corporate sponsors and more sites are added, with the expectation that it, too, will cross over into the black during 1987.

NTU COMPUTER ENGINEERING PROGRAM

Frederic J. Mowle, David G. Meyer, Philip H. Swain

School of Electrical Engineering
Purdue University
West Lafayette, IN 47906

Abstract

Live teleconference from Purdue University describing the Computer Engineering Program offered by the National Technological University. Areas to be covered include the degree program requirements, comments by a course instructor, and comments by a university administrative contact person. A live question and answer session is planned.

Background Information

The National Technological University (NTU) was established in Colorado as a nonprofit corporation in 1984. The academic programs offered by NTU draw upon approved course offerings from the 21 participating universities, all of which are members of the Association for Media-Based Continuing Education for Engineers (AMCEE). Although NTU's charter specifically prohibits offering baccalaureate or doctoral degrees, NTU offers selected undergraduate classes from participating universities to assure appropriate foundation for master's level coursework. NTU uses advanced educational and telecommunications technology to deliver instructional programs to graduate engineers and technical professionals at their employment locations. Each NTU site is operated by a sponsoring organization.

Academic Organization

The National Technological University relies upon a faculty consisting of consultants selected from the faculty of each participating institution. These faculty consultants are organized in discipline groups to form Graduate Faculties, typically with one representative in each discipline from each participating institution. At the present time, NTU offers Master of Science degrees in five disciplines: Computer Engineering, Computer Science, Electrical Engineering, Engineering Management, and Manufacturing Systems Engineering. Three standing Committees support each of the various Graduate Faculties. The Curriculum Committee in each discipline develops study programs and reviews all courses submitted by the participating universities. The Admissions and Academic Standards Committee for each Graduate Faculty sets the policies governing admission and criteria for students to continue as active degree candidates. The Staffing Committee in each discipline monitors activities of faculty consultants to assure that the proper faculty functions are performed.

Participating Universities

At the present time, the following universities are cooperating in the various degree programs offered by the National Technological University. The course suffix assigned each university is used to aid in the identification of course offerings.

University	Course Suffix
Boston University	V
Colorado State University	H
Georgia Institute of Technology	J
Illinois Institute of Technology	K
Iowa State University	U
Michigan Technological University	I
North Carolina State University	P
Northeastern University	F
Oklahoma State University	O
Purdue University	M
Southern Methodist University	N
University of Alaska	G
University of Arizona	E
University of Florida	R
University of Idaho	S
University of Kentucky	L
University of Maryland	B
University of Massachusetts	A
University of Minnesota	C
University of Missouri-Rolla	T
University of South Carolina	D

Method of Delivery

The National Technological University has its administrative offices on the campus of Colorado State University in Fort Collins, Colorado. However, the faculty are located on the campuses of the participating universities and the students are located at their work sites nationwide. Instructional programs are delivered by the faculty from the home campuses to the students through telecommunication technology.

The communication links facilitate student advising, faculty conferences, and special programming.

CH2345-7/86/0000/0005$01.00 © 1986 IEEE

Briefly, the NTU distribution system is satellite-based, using a satellite operating in the 12/14GHz (Ku) band. A series of satellite uplink stations located at participating universities has been installed, and television receive-only terminals are located at each organizational site of participating graduate students. The space segment is provided over existing Ku-band domestic communications satellites. The technical operation of the network is controlled from a central headquarters known as the NTU Network Control Center, where schedules are prepared, satellite channels are monitored for technical quality, and return communications (from student to instructor) are coordinated.

To make optimal financial use of satellite transponder time and the realities of the working student's class time, many course transmissions are recorded at the student's site on videotape for use at the convenience of the student. Teleconferencing and electronic mail, using one of the packet-switched networks, are the primary means of interaction between students and instructors.

Computer Engineering

Program Description. The National Technological University Master of Science Degree Program in Computer Engineering provides the means for engineers with a Bachelor of Science in Electrical Engineering, Computer Engineering, or Computer Science to complete the requirements for Master of Science. Applicants are considered only if sponsored by their employing or affiliated organizations. Applicants for admission to this program must submit Graduate Record Examination (GRE) scores. GRE examinations need only include the aptitude test (morning). Students may submit the advanced test in Engineering or Computer Science (afternoon) if they desire. Students must also provide two or three letters of recommendation. Letters from their supervisor as well as a professor, if the applicant has been out of school for less than four years, are required. The additional reference is the student's choice. The Curriculum Committee designed the approved curriculum around the model developed and published by the IEEE Computer Society (1977) and the ACM (1977). However, The National Technological University Master of Science Degree Program in Computer Engineering has the distinguishing characteristic of required "breadth" courses outside the field of computer engineering *per se.*

The Master of Science Degree Program in Computer Engineering consists of 30 semester credits (or the equivalent quarter credits) distributed through three broad categories of courses: Core, Depth and Breadth Courses. In addition, all successful candidates for the Master's Degree must participate in a noncredit seminar. The curriculum features substantial student choice in all three categories of courses, thereby enabling the students to tailor their programs of study to meet their specific needs and fulfill their particular aspirations, all within a coherent framework assuring academic excellence and state-of-the-art preparation. The Breadth Courses expose the students to a spectrum of topics. In this way, the University insures that students become aware of important and emerging areas that might otherwise be overlooked. The NTU curriculum in Computer Engineering remains open-ended with regard to advanced courses in order to encourage the students to take advantage of recently evolved courses concentrating on the latest developments in the field.

Completion of the curriculum requires approximately one and one-half years of full-time, graduate study. Students enrolled through The National Technological University, whose work schedules prevent full-time study, should expect to fulfill the requirements in five years by registering for at least two three-credit courses each academic year.

Academic Advising. Sound and responsive academic advising constitutes an integral part of every program of study offered by The National Technological University. The Admissions and Academic Standards Committee of the Computer Engineering Graduate Faculty assigns to each admitted student an academic advisor, who is a regular faculty member drawn from one of the participating universities and who contributes to the Master of Science Degree Program in Computer Engineering. The academic advisor assists the student to reach informed decisions about the program of study, including course selections. In addition, the academic advisor must approve all petitions for exceptions to the prescribed program of study. Communication between the student and the advisor occurs, in most instances, by telephone, although other media — including regular mail, electronic mail, and personal contact — are also available, depending upon the circumstances in each instance.

Curricular Requirements. The National Technological University offers courses in Computer Engineering and related fields at the "mezzanine" and graduate levels. A mezzanine course is defined as one appropriate for undergraduate students with senior standing or for entering graduate students. However, *only* graduate courses will count toward fulfillment of the Depth Requirements in the Master of Science Degree Program in Computer Engineering. Further, candidates for the degree can count no more than 12 credits earned in mezzanine courses to fulfill the requirements for the Master of Science Degree.

Each student should expect to enroll in ten or more courses with a minimum of three Core Courses, four Depth Courses, two Breadth Courses, one Elective and a noncredit seminar. A total of 30 credits are necessary for graduation. The required curriculum consists of five parts.

Core Requirements

Each student must complete at least eight credits of required Core Courses, with at least one course in each identified area. Core courses are divided into three general areas:

1. Software Systems
2. Computer Architecture
3. Algorithms and Data Structures

Depth Requirements

Each student must complete at least four additional courses consisting of two courses from each of two areas listed below, and all courses taken to fulfill the Depth Requirements must be graduate courses. Depth courses provide instruction on the most advanced and current topics in seven distinct areas, three of which form the Core Requirements described above:

1. Software Systems
2. Computer Architecture
3. Algorithms and Data Structures
4. Digital Design
5. Graphics
6. Intelligent Systems
7. Mathematics and Computational Methods

With the advice of the academic advisor, the student should plan a program of study that assures appropriate depth in at least two areas.

Breadth Requirements

Each student must complete at least six credits in Breadth Courses. Breadth Courses focus on fields that relate to or support the study of Computer Engineering, including:

1. Business Applications
2. Computer-Aided Design/ Computer-Aided Manufacturing
3. Communications
4. Control and Robotics
5. Electrophysics
6. Mathematics
7. Signal Processing
8. Theory of Computing

With the assistance of an academic advisor, the student should plan an integrated program of study that assures breadth overlaying the depth achieved in Computer Engineering.

Elective Requirements

Each student must complete at least three additional elective credits consisting of a Core, Depth, or Breadth Course to bring the total credits to 30.

Seminar Requirements

Each student must complete one noncredit seminar offered by The National Technological University.

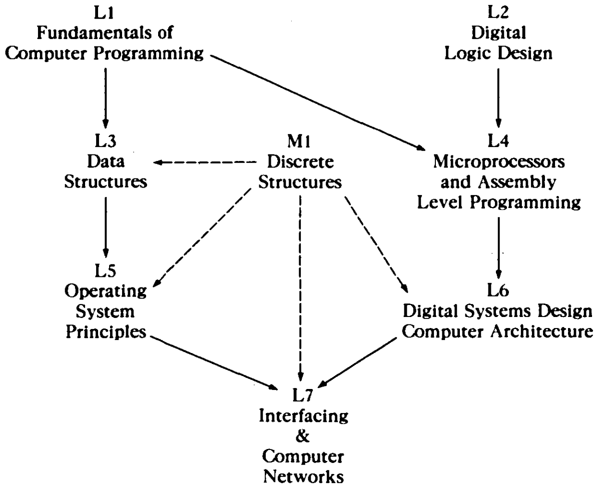
Thesis

The National Technological University offers a nonthesis Master of Science Degree Program in Computer Engineering. However, when desirable and appropriate, as determined by the student in consultation with an academic advisor, a thesis, with a maximum of six credits, can be substituted for the Elective Course and one of the Depth Courses.

Undergraduate Bridging Courses

Applications of computers are pervasive today, affecting the work of most engineers and technical professionals. It is natural, therefore, for people with very diverse technical backgrounds to seek additional education in computing. For that reason, NTU faculty have identified the undergraduate prerequisites which are necessary background for entering graduate study in computer engineering and computer science. Eight undergraduate bridging courses have been identified. Detailed outlines of these courses can be found in the National Technological University Bulletin.

Flow Chart of Undergraduate Computer Science and Engineering Program



L1	Fundamentals of Computer Programming
L2	Digital Logic Design
L3	Data Structures
L4	Microprocessors and Assembly Level Programming
L5	Operating System Principles
L6	Digital System Design - Computer Architecture

L7	Interfacing and Computer Networks
M1	Discrete Structures

The bridging courses are not available for graduate credit.

Courses of Instruction

The National Technological University categorizes the courses of instruction in accordance with the Core, Depth, and Breadth Requirements within the curriculum in Computer Engineering and arrays the courses according to the list that follows. The courses are arrayed by subject matter areas as identified by two-letter prefixes which serve to identify specific subject matter areas (e.g., SS refers to Software Systems, AC to Architecture and Computer Design, DD to Digital Design, *et cetera*). Courses are numbered as they fall within each appropriate subsection. The suffixes following course numbers refer to section offerings and identify the institution offering the individual sections of the course.

Core Courses

Software Systems

SS 10—19	Systems Programming
SS 20—29	Programming Languages

Architecture and Computer Design

AC 30—39	Computer Architecture
AC 40—49	VLSI
AC 50—59	Embedded Computer Systems

Algorithms and Data Structures

AD 60—69	Data Structures
AD 70—79	Analysis of Algorithms

Depth Courses

Advanced Digital Design

DD 10—19	Advanced VLSI Design
DD 20—29	Reliable Computation
DD 30—39	Computer Arithmetic
DD 40—49	High Speed Computation
DD 50—59	Data Communications Systems
DD 60—69	Digital Hardware Design

Advanced Computer Architecture

CA 10—19	Computer Architecture/ Operating Systems
CA 20—29	Distributed Computer Systems
CA 30—39	I/O and Memory Systems Architecture

Systems Programming

SP 10—19	Advanced Techniques in Translator Design
SP 20—29	Advanced Operating Systems
SP 30—39	Data-Base Systems
SP 40—49	Modeling and Performance Evaluation

Mathematics and Computational Methods

CM 10—19	Numerical Analysis
CM 20—29	Computational Methods for Linear Algebra
CM 30—39	Partial Differential Equations and Numerical Techniques for Solving Them
CM 40—49	Stochastic Queuing Theory and Statistical Analysis
CM 50—59	Automata Theory

Advanced Software Techniques

AS 10—19	General Methods for Artificial Intelligence
AS 20—29	Knowledge-Based Systems
AS 30—39	Robotics
AS 40—49	Computer Graphics
AS 50—59	Computer Networks
AS 60—69	Computer Vision
AS 70—79	Programming Languages for AI

Breadth Courses

Electrophysics

EP 10—19	Lasers
EP 20—29	Microelectronics
EP 30—39	Electronic Systems
EP 40—49	Optics
EP 50—59	Field Theory
EP 60—69	Solid State Devices

Communication and Control

CC 10—19	Digital Control Theory
CC 20—29	Digital Communication Theory
CC 30—39	Coding Theory
CC 40—49	Statistical Communications Theory
CC 50—59	Information Theory
CC 60—69	Speech Processing
CC 70—79	Image Processing

Operations Research

OR 10—19	Linear Programming and Its Applications
OR 20—29	Algorithms for Combinatorial Optimization

Business Applications

BA 10—19	Management Information Systems
BA 20—29	Financial and Decision Analysis Techniques

Mathematics

MA 10—19	Discrete Structures
MA 20—29	Combinatorial Analysis
MA 30—39	Stochastic Processes, Queuing Theory, and Statistical Analysis
MA 40—49	Advanced Calculus

Example Programs of Study

The following examples illustrate the flexibility available to students wishing to specialize in specific areas of computer engineering.

M.S. In Computer Engineering
Software Engineering Emphasis

Core		Credits
SS 15-C	Software Engineering I	2.7*
AC 30-A	Advanced Computer Architecture I	3
AD 70-A	Algorithms and Data Structures	3
Depth		
AD 60-C	Introduction to Data Structures	2.7
AD 61-C	Advanced Data Structures	2.7
SP 30-C	Distributed Data Base Systems	3
SP 15-B	Theory of Programming Languages	3
SP 20-C	Introduction to Operating Systems	2.7
Breadth		
CC 30-F	Error Correcting Codes	3
MA 30-A	Probability and Random Processes	3
Elective		
SS 20-A	Programming Languages	3
		31.8

* Fractional credits due to conversion of quarter hours to semester hours.

Computer Architecture Emphasis

Core		Credits
SS 20-A	Programming Languages	3
AC 30-A	Advanced Computer Architecture I	3
AD 60-B	Data Structures	3
Depth		
AC 31-A	Advanced Computer Architecture II	3
AC 32-A	Testing and Diagnosis of Digital Systems	3
DD 60-G	Advanced Digital Hardware Design	3
DD 50-A	Computer Communications Networks	3
Breadth		
CC 20-F	Digital Signal Processing	3
MA 30-A	Probability and Random Processes	3
Elective		
CA 10-H	Microprogramming	3
		30

Acknowledgements

The computer engineering program was planned and developed under the direction of Harold S. Stone, aided by the members of the Computer Science and Engineering Graduate Faculty.

S. K. Chang
Electrical Engineering Dept.
Illinois Institute of Technology

Paul Cohen
COINS Graduate Research Center
University of Massachusetts

John Dickinson
Computer Science Dept.
University of Idaho

Frederic J. Mowle
(Chairman, Computer Engineering
Curriculum Committee)
School of Electrical Engineering
Purdue University

Robert A. Mueller
Dept. of Computer Science
Colorado State University

Troy Nagle
Electrical Engineering Dept.
North Carolina State University

Gerald Peterson
Dept. of Electrical and
Computer Engineering
University of Arizona

Robert Pettus
(Chairman, Admissions and Academic
Standards Committee)
Dept. of Electrical and
Computer Engineering
University of South Carolina

Randy Reininger
Electrical and Computer Engineering
Oklahoma State University

Sartaj Sahni
(Chairman, Computer Science
Curriculum Committee)
Department of Computer Science
University of Minnesota

Charles Silio
(Chairman, Staffing Committee)
Dept. of Electrical Engineering
University of Maryland

Terry Smay
Dept. of EE/CPR E
Iowa State University

David B. Spell
Electrical Engineering Dept.
University of Alaska

Keith Stanek
Dept. of Electrical Engineering
Michigan Technological University

John Staudhammer
Electrical Engineering and Computer
and Information Sciences
University of Florida

John Wakerly
Consultant
Mountain View, CA

Special thanks for the success of the program is due to the numerous course instructors, advisors, site coordinators, administrative contact persons, and the staff of NTU who make the program work.

Satellite Presentation

The formal presentation for this session will be divided into three areas. Professor Frederic J. Mowle, Chairman of the Computer Engineering Curriculum Committee, will present an overview of the Computer Engineering program. Professor David G. Meyer, course instructor for CA 30-M, Advanced Microprocessors and System Design Components, will discuss the NTU program from the viewpoint of a course instructor. Professor Philip H. Swain, Director of Continuing Engineering Education at Purdue University, will discuss the NTU program from the viewpoint of a University Administrative Contact Person.

Additional Information

Additional information on the various NTU programs can be obtained from:

National Technological University
P. O. Box 700
Fort Collins, CO 80522
(303) 491-6092

THE NTU COMPUTER SCIENCE PROGRAM

SARTAJ SAHNI

UNIVERSITY OF MINNESOTA

ABSTRACT

This paper describes the Computer Science Master's program of the National Technological University, Fort Collins, Colorado.

1. NTU

The National Technological University (NTU) was established as a non-profit private educational corporation in January 1984 in Colorado. Its exclusive mission is to serve the educational needs of graduate engineers and to award master's degrees in engineering related disciplines. Its charter prohibits the award of baccalaureate and doctoral degrees. Currently, NTU offers master's degrees in: Computer Science, Computer Engineering, Electrical Engineering, Engineering Management, and Manufacturing Systems Engineering.

NTU was created with a unique mission: provide graduate engineers a quality master's program in engineering that can be successfully completed by enrollees without leaving their place of employment. It differs from its parent organization AMCEE (Association for Media-Based Continuing Education for Engineers) primarily in that AMCEE does not grant degrees. Through NTU, it is possible for practicing engineers to obtain a master's degree no matter how distant they may be from the university. Further, their progress towards such a degree is not adversely affected by a transfer within their company or between employers that are NTU sponsors. Since all students enrolled in NTU programs must be sponsored by their employers, NTU students changing employers must ensure that their new employer will sponsor them.

In order to best serve its mission, NTU has identified four primary goals [1]: "The discovery, dissemination, application, and preservation of knowledge within designated fields of study, with special emphasis upon those engineering disciplines critical to the continued development and implementation of technology appropriate to an information knowledge-intensive society."

NTU has set itself the following seven objectives [1]:

(1) **The Instructional Objectives**
The University encourages and fosters the development of innovative pedagogical and technological methods to enhance learning and achievement.

(2) **The Creativity Objectives**
The University encourages and fosters the development of creativity among the faculty and students as the most dynamic response to a rapidly changing technological society. In an information, knowledge-intensive age, people must have the developed capacity to glimpse the future as it unfolds and to act to shape it. The National Technological University research seminar is directed to this end.

(3) **The Student Relations Objectives**
The University stresses the responsibility of the faculty to the professional as well as instructional needs of the students and the responsibility of the students for their own individual growth and development.

(4) **The Human Resources Objectives**
The University selects as participating faculty only those persons with reputations for outstanding performance as teachers and scholars, and offers selected programs to assist these faculty members in their own professional development.

(5) **The Support Objectives**
The University provides the facilities and services essential to the fulfillment of the institutional mission. Moreover, the University strives for the refinement and development of technological means to enhance the quality of the facilities and services.

(6) **The Organization and Administration Objectives**
The University maintains a supportive organization and administrative structure that rests firmly upon participatory management and academic programs.

(7) **The Evaluation Objectives**
The University maintains a process of continuous evaluation of programs and services to ensure progress toward the achievement of institutional goals.

There are three distinct aspects to NTU:

(1) **NTU Administrative Services**
These are responsible for policy, admissions, maintenance of records (including grades), award of degress, etc. The NTU office in Fort Collins, Colarode provides these services.

(2) **Participating Universities**
These are schools from which NTU obtains the courses that it provides its students. All courses offred by NTU are regulary offered to graduate students at the participating universities. These courses are typically offered in specially equipped classrooms at the participating university campus. The lectures including any discussion and question/answer sessions are either broadcast live over satellite to sites at which NTU students are located or are taped and then viewed with time delay at these sites. In the case of live satellite broadcast, there is provision for a talk back channel. So, it possible for NTU students to ask questions in real time.

CH2345-7/86/0000/0011$01.00 © 1986 IEEE

(3) **Sponsoring Organizations**
These are the entities that are permitted to enroll their employees in NTU programs. Each sponsoring organization must maintain suitable equipment at each of its instructional sites for the receipt and viewing of the lectures. To receive courses over satellite, a down link is needed.

The universities that are participating in the NTU program are:

(1) Boston University
(2) Colorado State University
(3) Georgia Institute of Technology
(4) Illinois Institute of Technology
(5) Iowa State University
(6) Michigan Technological University
(7) North Carolina State University
(8) Northeastern University
(9) Oklahoma State University
(10) Purdue University
(11) Southern Methodist University
(12) University of Alaska
(13) University of Arizona
(14) University of Florida
(15) University of Idaho
(16) University of Kentucky
(17) University of Maryland
(18) University of Massachusetts
(19) University of Minnesota
(20) University of Missouri-Rolla
(21) University of South Carolina

The sponsoring organizations are divided into two categories: corporate subscribers and major site subscribers. The corporate subscribers are:

(1) AT&T
(2) Digital Equipment Corp.
(3) Eastman Kodak Co.
(4) General Electric Co.
(5) CTE Spacenet Corp.
(6) Hewlett-Packard Co.
(7) Intel Corp.
(8) IBM Corp.
(9) Motorola
(10) NCR Corp.
(11) Tektronix, Inc.

The major site subscribers are:

(12) ALCOA
(13) General Dynamics Corp.
(14) General Instruments
(15) Honeywell, Inc.
(16) Magnavox Co.
(17) RCA Corp.
(18) Sandia National Labs.

A sponsoring organization may have several sponsoring sites. There are approximately 75 sponsoring sites at present.

In the remainder of this paper, I shall provide an overview of the NTU master's program in Computer Science. This overview is not intended to serve as a substitute for the NTU Bulletin [1]. Persons interested in knowing the exact admission and graduation reuirements should read this bulletin carefully.

2. ADMISSION REQUIREMENTS

To be admitted to the NTU master's degree in Computer Science program, the applicant must have successfully completed a Bachelor of Science Degree in Computer Science or a related field. All applicants are also required to perform satisfactorily in the aptitude test of the Graduate Record Examination (GRE). In addition, applicants may be granted provisional admission to enable them to make-up deficiencies prior to obtaining regular graduate admission.

3. THE COMPUTER SCIENCE CURRICULUM

The Master of Science Degree Program in Computer Science consists of 30 semester credits (or the equivalent quarter credits). These must be appropriately distributed over the three broad categories of courses: Core, Depth, and Breadth. In addition, participation in a seminar is required.

CORE COURSES

The general categories for the core courses are:

(1) Algorithms and data structures
(2) Software systems
(3) Computer architecture
(4) Mathematics and theory of computation
(5) Artificial intelligence

DEPTH COURSES

The categories for the depth courses are listed below. Together with each such category, a sampling of some of the topics included in the category is provided.

(1) **Computer Software**
- Structure of higher level languages
- Translators
- Operating systems
- Database systems
- Software engineering
- Programming and algorithms for supercomputers
- Computer networks
- Computer graphics
- Security and protection mechanisms

(2) **Computer Architecture**
- Computer architecture operating systems
- Distributed computer systems
- I/O and memory systems architecture
- Supercomputer architecture
- CAD tools for VLSI design

(3) **Mathematics & Theory of Computing**
- Numerical analysis
- Methods for supercomputing
- Design & Analysis of algorithms
- Modeling and performance evaluation
- Automata theory
- Formal languages
- Complexity theory

(4) **Artificial Intelligence**

- General methods for artificial intelligence
- Knowledge based systems
- Knowledge representation
- Natural language processing
- Computational epistomology
- Logic programming
- Robotics
- Computer vision
- Speech recognition
- Pattern recognition

BREADTH COURSES

The general categories and a sampling of example topics within these categories are summarized below:

(1) **Operations Research**

- Linear Programming
- Integer Programming
- Nonlinear Programming
- Queueing Theory
- Dynamic Programming
- Stochastic Methods
- Combinatorial Optimization

(2) **Electrical Engineering**

- VLSI Technology & Design
- Coding & Information Theory
- Computer Hardware Design
- Computer Communications

(3) **Business Applications**

- Management Information Systems
- Computer Methods In Management

(4) **Mathematics**

- Combinatorics
- Graph Theory
- Statistics
- Probability
- Recursion Function Theory
- Logic

NTU is able to provide a very rich selection of courses in each of the above categories. This stems from the fact that NTU gets its courses from its participating universities. These universites together have a combined engineering faculty in excess of 2,800. The number of different computer science and related courses taught at these universities together is far larger than the number of such courses taught at any one of these universities. In fact, it is safe to say that NTU's course offering, in computer science, is richer than that of any single university in the country.

4. CREDIT DISTRIBUTION REQUIREMENT

The required 30 NTU credits should satisfy the following distribution:

(1) At least one course from each of the five core categories.

(2) At least three courses from any one of the depth depth categories.

(3) At least one course from a breadth category.

To meet the needs of students whose Bachelor's degree is not in Computer Science, NTU offers three bridging courses:

(1) Computer Programming

(2) Discrete Structures

(3) Data Structures and Algorithms

Credits earned for bridging courses cannot be applied towards the 30 credits required for a master's degree.

5. 1985-86 ENROLLMENT FIGURES

While NTU first started offering courses in Fall semester 1984, the Computer Science program became available only in Fall 1985. The enrollment figures for the 85/86 academic year are summarized in Table 1. The entries in this table are total course enrollment counts. Thus the sum of the number of students enrolled for credit in each course in the Fall semester was 232. The actual number of students taking courses for credit in this semester may be slightly less as it is possible for one student to take several courses in a given semester. The number of different courses made available during the academic year exceeded 130. A little over 40 of these were actually subscribed to by NTU students.

	Fall	Winter & Spring	Total
Credit	232	200	432
Audit	92	105	197
Total	324	305	629

Table 1 1985/86 enrollment figures

In addition, 16 courses were made available for the summer 1986 session. Enrollment figures for this session were not available at the time this paper was written. As is evident, the Computer Science program is off to an excellent start. Enrollments are expected to climb significantly in the next few years.

6. MEDIA

NTU is committed to the use of state of the art technology to provide the best delivery of its courses. Many of the courses are broadcast live over satellite. The satellite link prvides for a color transmission of the video. A two way audio channel is also used. This permits live interaction with remote students. The remaining courses are viewed with a time delay. These get to the viewing sites either via satellite transmission during off peak hours or via video tapes express mailed to the viewing sites. Live tutorial sessions are organized for courses that are not broadcast live. This gives students an opportunity for live interaction with the instructor and/or the teaching assistants.

The lectures may be viewed by students at a time convenient to them. This is true even in the case of live broadcasts as viewing sites have video taping facilities and taping authorization. This flexibility is essential to the success of the NTU program as all its students are full time employees. Their primary responsibility is is to their employers. With NTU, students need not organize their work schedules around their class activities. Rather, the class activities are made to fit around the work schedule.

Interaction with the instructor and teaching assistants may take the form of telephone conversations or communication via AT&T mail or the US postal service. Initial data indicates that only about 10% of the students are currently using AT&T mail. The use of this powerful electronic communications system needs to be encouraged as a convenient way to ask and answer questions that are of interest to all students enrolled in a given course. Further, this could provide a fast inexpensive mechanism to get assigments to and from remote sites. The delay currently involved in doing this poses a serious problem. NTU is currently exploring the possibility of adding an analog signal to the end of its video signals. This analog signal will encode classroom handouts. Thus, these will become available at the same time as the lectures.

7. SUMMARY

NTU has begun a new era of educational delivery. It promises to bring quality education leading to a graduate degree into the work place. By combining the educational resources of the country's leading institutions, NTU is able to provide curricula that is richer than found at any one of these institutions. In fact, the services of NTU could also be benificially used by participating (as well as nonparticipating) universities to enrich their own offerings.

It is anticipated that by the end of this decade, NTU will become one of the top producers of M.S. degrees in engineering.

8. ACKNOWLEDGEMENTS

I am grateful to Dr Lionel Baldwin, President, NTU for providing some of the data used in this paper and also for authorizing the use of material from the NTU Bulletin [1].

9. REFERENCES

[1] The National Technological University Bulletin, 1986-1987 Academic Programs, NTU, Fort Collins, Colorado

WORKSTATIONS AT CARNEGIE MELLON

Bruce Arne Sherwood

CDEC, Carnegie Mellon, Pittsburgh PA 15213

ABSTRACT

Carnegie Mellon University intends to deploy advanced-function workstations for students, faculty, and staff. The basic goals and the organizational structures created to attain those goals are described. Several hundred workstations are currently deployed. There is an overview of the Andrew and CMU Tutor software environments.

In 1982 Carnegie Mellon University (CMU) established the goal of giving all students ready access to the wealth of information made possible by personal computers, and providing the kind of computational environment that would enhance learning. At the same time, faculty would use these computers for research and teaching, and the university itself would benefit, both in improved administrative functions and more rapid communications.

Some other universities have adopted the goal of providing every student with a personal computer, but CMU's plan is unusual in emphasizing advanced "workstations" which are much more powerful than typical personal or home computers[1]. While these workstations are rather expensive today, the rapid pace of computer evolution insures that these tools will be inexpensive and widely available in the late eighties. Because it takes significant lead time to integrate any computer system into university life, Carnegie Mellon started early to deploy and exploit machines which will constitute the next generation of personal computers.

Another important component of the Carnegie Mellon plan is to link these workstations together, giving students, faculty, and staff immediate access to up-to-the-minute programs and data. Fiber-optics links between all major campus buildings already exist. By the fall of 1986, IBM token-ring networks will be installed inside the buildings to distribute information to offices, classrooms, and to student residences.

ORGANIZATIONAL STRUCTURE

When the project was conceived, a contract between Carnegie Mellon and IBM established the Information Technology Center (ITC) to develop system software to support large-scale deployment of workstations. ITC, led by its director and computer scientist James Morris, has focused on two major aspects of the challenge: (1) a file system to support several thousand networked workstations, and (2) a friendly, graphics-oriented operating framework for users of workstations. Representatives from IBM work closely with professional system designers, CMU graduate students, and consultants from the Department of Computer Science, doing the basic research and development necessary to create this large-scale system. While much remains to be done, the system is already operational and heavily used. The various system-software components built by ITC are collectively called the Andrew system[2,3], honoring Andrew Carnegie and Andrew Mellon.

The university also set up the Center for Design of Educational Computing (CDEC) to help create educational applications. The cognitive psychologist Jill Larkin was the first director of CDEC, and the new director is the philosopher Preston Covey. CDEC consults with faculty, provides seed money for educational projects, develops software tools to assist educational programming, and runs a seminar series and newsletter. CDEC serves as a liaison between faculty and ITC, helping both groups define and refine the computing environment necessary for education.

The environment for developing educational computer applications is enhanced by CMU's strong Department of Psychology with its special strength in cognitive psychology, the study of thinking. The expertise on learning from the Department of Psychology, combined with the technical expertise of members of the Department of Computer Science, promises to provide exceptional new opportunities for students to learn and for faculty to explore new ways to teach. Already, unusual "artificially intelligent" tutoring programs exist which will be increasingly useful as Carnegie Mellon installs workstations capable of handling these complex programs.

The nature of the new generation of powerful, sophisticated computers has made it possible for ITC to develop system software which operates on a variety of different computers. This will greatly reduce the current severe problems of obsolesence and of incompatibility which plague educational computing. To encourage the kind of sharing that is so important in an undertaking of

CH2345-7/86/0000/0015$01.00 © 1986 IEEE

this size, CDEC works closely with the Inter-University Consortium for Educational Computing (ICEC), headed by Ken Friend, who is stationed at Carnegie Mellon. The Consortium includes major universities which are strong in the computing field, such as Brown, Cornell, Michigan, MIT, and Stanford. For balance of viewpoints and needs, ICEC also includes some smaller schools such as Vassar and Mills. Through collaboration among these universities and colleges, the Consortium hopes to convince computer vendors that it is worthwhile to develop compatible systems. If exceptional educational software developed at one institution could be readily adapted to the needs of others, the total equipment market would be much larger than it is now.

Older organizations are playing new roles. The user services section of the university computing center now offers short courses on how to use and program the workstations. The university library in association with CDEC has created a software library with acquisition and cataloging functions.

All of these organizations report to William Arms, vice president for academic services. John P. Crecine, senior vice president for academic affairs, has played a central role in formulating policies and plans concerning the workstation project.

DEPLOYMENT HISTORY

In January 1985 fifty workstations were placed around the campus to give faculty an early look at the system and to start developing educational applications. In January 1986 the first public cluster of advanced workstations was made available for shared student use. This cluster contained twenty-four IBM RT PC's. By fall 1986 there are expected to be several hundred workstations on campus, with about a hundred of them in public clusters. At present, even with quantity educational discounts workstations cost close to ten thousand dollars, which precludes individual student ownership. However, by the late eighties prices should drop considerably, enabling students to buy any of several brands of compatible advanced workstations in the campus computer store.

Although some students were involved as programmers from the beginning of the project, other kinds of student use were limited due to lack of public machines. In the spring of 1986 other students could begin to take advantage of the new resources[4]. For example, undergraduate architecture students used a powerful structural design package to synthesize and visualize buildings, and students in a graduate physics course on non-linear dynamics used a sophisticated function plotter.

THE SOFTWARE ENVIRONMENT

The machines deployed at CMU (currently including IBM RT PC's, DEC Vaxstation 2's, and Sun workstations) all use the Berkeley Unix 4.2 operating system, which is a key to compatibility of applications. The Andrew file system software (called Vice) intercepts Unix file references and converts these references into calls to file servers to which the workstations are attached. Each file server handles about thirty workstations, and the file servers are connected to each other by a high-speed backbone link. As a result, a user can log in to any workstation anywhere on campus and file access is completely transparent. Whole files are transferred over the network and cached on a local hard disk. The workstation receives a message from a file server if the server has a newer version of a previously cached file. While there exist many implementations of networked personal computers, Vice is unusual in being designed from the beginning to be expandable to several thousand workstations while providing good security for data.

The Andrew programmer and user interfaces (called Virtue) consist of a window manager, a text-processing subroutine library, a database subroutine library, and applications such as editors, mail, and bulletin boards. The Andrew window manager will soon be replaced by the X window manager built by MIT, in order to enhance compatibility above that provided by the underlying Unix. The text-processing subroutine library, called the Base Editor, provides powerful display and manipulation of text, including italics and centering, scroll bars, and real-time justification within a rectangle. Because it is a subroutine library, not a text editor, it is easy to invoke a sophisticated text editor inside one's own program. It also has a marker facility which will keep track of a part of a file and inform the surrounding program when a marked section has been changed. The Base Editor library will soon be replaced by a new "Multi-Object Environment" which has the important enhancement of permitting "insets" to be contained in documents. These insets can be equations, graphs, tables, etc. Cutting and pasting text containing these non-text items will be as easy as current text operations.

CMU TUTOR

While the system software is written in C, this language is not well suited for applications programming by non-expert programmers. To give such people the ability to create complex interactive graphics programs which exploit the rich capabilities of the workstation environment, Bruce and Judith Sherwood have created a programming environment called CMU Tutor[5,6,7,8]. While patterned on the MicroTutor language developed in the PLATO project at the University of Illinois, CMU Tutor has enhancements for the workstation environment, including pop-up menus, variable window size, mouse support, etc. Thanks to the Base Editor keeping track of which routines have been changed and therefore need recompilation, CMU Tutor operates as an incremental compiler, which combines the revision speed of an interpreted language with the execution speed of a compiled language. A tightly integrated graphics editor generates source code from a display created using the mouse. An on-line help facility not only describes the language features in detail but provides sample routines which can be run immediately thanks to the multi-window and incremental-compiler environment.

During the summer of 1986, the Carnegie Foundation of New York funded four one-week workshops on advanced-function workstations and CMU Tutor for faculty and support staff at universities in the Interuniversity Consortium for Educational Computing. The workshops were held at Carnegie Mellon, and in less than one week many workshop participants created significant interactive graphics programs written in CMU Tutor. IBM funded an additional week of workshops for people from some non-ICEC universities.

Work is underway to be able to execute CMU Tutor programs on popular micros, including the IBM PC and the Macintosh. A limited authoring capability will also be available, although the workstation environment is a more productive one for the creation of new materials.

REFERENCES

[1] Crecine, J. P. The next generation of personal computers. Science **231**, 935-943 (Feb. 28, 1986).

[2] Morris, J. H., Satyanarayanan M., Conner, M. H., Howard, J. H., Rosenthal, D. S. H., and Smith, F. D. Andrew: a distributed personal computing environment. Communications of the ACM **29**, 184-201 (March 1986).

[3] Trowbridge, D. Using Andrew for development of educational applications. Proceedings of the IBM Academic Information Systems University AEP Conference, Alexandria, Virginia, 85-89 (June 1985)

[4] Trowbridge, D. A sampler of educational software at CMU. Proceedings of the National Educational Computing Conference, San Diego, 135-142 (June 1986).

[5] Sherwood, B. A. An integrated authoring environment. Proceedings of the IBM Academic Information Systems University AEP Conference, Alexandria, Virginia, 29-35 (June 1985). Here it is explained that CMU Tutor gets its name from being implemented in C, with MU being the Greek letter for Micro.

[6] Sherwood, B. A., and Sherwood, J. N. CMU Tutor: An integrated programming environment for advanced-function workstations. Proceedings of the IBM Academic Information Systems University AEP Conference, San Diego (April 1986).

[7] Sherwood, B. A., and Sherwood, J. N. *The CMU Tutor Language, Preliminary Edition.* Stipes Publishing Company, 10 Chester Street, Champaign, Illinois 61820 (1986).

[8] Sherwood, J. N. *CMU Tutor Reference Manual.* Carnegie Mellon University internal report (1986). This is a printed version of the on-line reference manual.

INTELLIGENT TUTORING SYSTEMS FOR PROFESSIONALS

ALAN M. LESGOLD

Learning Research and Development Center
University of Pittsburgh
Pittsburgh, Pennsylvania 15260

Abstract

Tools are becoming available for developing intelligent tutoring systems to teach professional and technical jobs. One basic tool is a method for analyzing jobs that involve considerable problem solving, such as electronics troubleshooting. A second tool is a combined device simulation and hypertext design capability. Combined with recent developments in cognitive instructional science and artificial intelligence, these tools promise affordable and efficient computer-based tutors that can speed up on-the-job learning.

Introduction

So far, most of the intelligent tutoring systems that have been built have aimed at relatively simple educational goals. However, it is now possible to build systems that can address the very job-specific, complex tasks that professional and technical workers face in our rapidly changing society. While many approaches seem worthy of investigation, I favor job-situated, coached practice as the basic approach.

In such an approach, a series of problem forms are developed that afford opportunities to exercise the various bits of knowledge one wants to teach. Trainees work through these problems in a cognitively real simulation environment, receiving coaching as necessary. Decisions about which variants of which problems to present and how to coach are made intelligently on the basis of aspects of the trainee's performance. This means that the following capabilities, among others, are needed to build such a tutoring system: (a) a curriculum goal structure to motivate the choice of problem forms and variants; (b) a cognitively-real simulation of the work environment, within which problems can be posed; (c) adequate knowledge about how people at different levels of skill think and act as they attempt to solve the problems that will be posed. In the sections that follow, two tools are described that contribute to developing these components. A task analysis methodology is described that can drive the development of the curriculum, as is a tool for building the simulated work environment.

Effective Problem Space Descriptions: A Tool for Cognitive Task Analysis

In order to decide what to teach, it is necessary to do task analyses of the jobs that trained professionals are asked to perform. Because the jobs a professional does are largely thinking jobs, cognitive task analyses are required — we need to know how professionals think their way through problems. In our own work, we have found it useful to begin by identifying an expert who has to contend with the problems produced by inadequate training. Such a person can be helpful in identifying classes of problems that the target group of professionals should be able to solve but often are not. We then work with the expert to develop an effective problem space for each such problem. That is, we ask the expert to describe how he would solve the problem and then probe deeply for alternative solution methods that either other experts or trainees might conceivably take.

We then use a differential empirical strategy to verify the expert's analyses. We take other experts, new professionals who are doing well, and new professionals who are having difficulties and ask them all to solve the problems for which we have preliminary effective problem spaces developed. The general differences between fast-track and slower new professionals are used to develop a set of goals for the tutor, its curriculum. The specific differences in problem solution paths are used as a basis for the part of the tutor that coaches problem solving performance. So, a trainee is constantly solving problems that are known to separate faster and slower newcomers to the profession, and the tutor is prepared to coach, assist, and demonstrate the steps that newcomers are likely to find difficult.

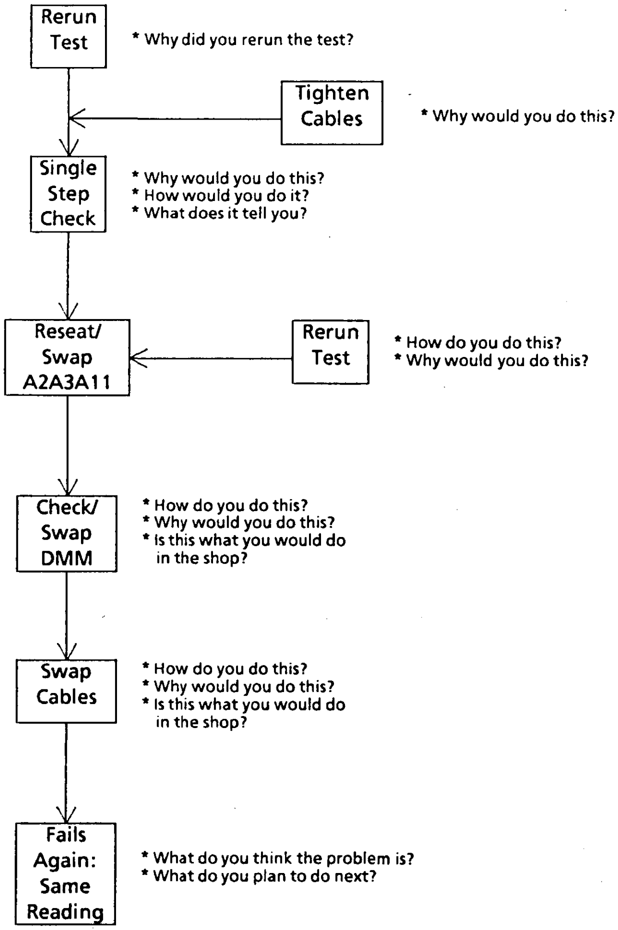
A few specifics

The approach my colleagues and I have used can be made clearer by looking at an example. One of our projects is to develop a tutor for a particular Air force specialty in which manual (non-automated) test equipment is used to diagnose and repair faults in navigational equipment for a particular airplane, the F-15. To carry out this project, we needed to find out what was difficult about this job, which involves massive amounts of electronic circuitry. Our breakthrough came not so much from our psychological expertise but rather from interactions of three of us (the author, Debra Logan, and Susanne Lajoie) who had substantial cognitive psychological training with an electronics expert (Gary Eggan) who had extensive experience watching novice troubleshooting performances. He pointed out that it was quite possible to specify the entire effective problem space, even for very complex troubleshooting problems. That is, he could almost completely specify all of the steps that an expert would take as well as all of the steps that any novice was at all likely to take in solving even very complex troubleshooting problems. In this case, the task was to find the source of a failure in a test station that contained perhaps 40 cubic feet of printed circuit boards, cables, and connectors, but various specific aspects of the job situation constrained the task sufficiently so that the effective problem space could be mapped out.

This then created the possibility that we could specify in advance a set of probe questions that would get us the information we wanted about technicians' planning and other metacognitive activity in the troubleshooting task. For what is probably the most complex troubleshooting task we have ever seen, there are perhaps 55 to 60 different nodes in the problem space, and we have specific metacognitive probe questions for perhaps 45. Figure 1 provides an example of a small piece of the problem space and the questions we have developed for it.

An examination of the questions in the Figure reveals that some are aimed at very specific knowledge (e.g., *How would you do this?*), while others help elaborate the trainee's plan for troubleshooting (consider *Why would you do this?* or *What do you plan to do next?*). Combined with information about the order in which the trainee worked in different parts of the problem space, this probe information permits reconstruction of the trainee's plan for finding the fault in the circuit and even provides some information about the points along the way at which different aspects of the planning occurred. In fact, we went a step further and also asked a number of specific questions about how critical components work and what their purpose is. Finally, when a trainee was headed well away from a reasonable solution path, we would, at preplanned points, redirect his efforts back to more fertile ground.

Figure 1
Examples of Problem Space Probes



After reviewing the records of performance in our tasks, we developed six scales on which we scored each airman. Each of these scales could be further subdivided into subscales to permit more detailed and task-specific issues to be addressed. The six scales were titled *plans, hypotheses, device and system understanding, errors, methods and skills,* and *systematicity.* Table 1 gives two examples for each scale of the items for which points could be earned (in the error scale, more points means more errors and thus is a lower score).

Table 1
Examples of Items Counted in each Subscale

Plans
- Extend and test a card.
- Trace through the schematic of an individual card.

Hypotheses

- There is a short caused by a broken wire or a bad connection.
- The ground is missing from the relay.

Device and system understanding

- Understanding and use of the external control panel.
- Understanding of grounds and voltage levels in the test station.

Errors

- Misinterpreting/misreading the program code, called FAPA, for a test that the test station carries out under computer control.
- Getting pin numbers for a test wrong.

Methods and skills

- Schematic understanding: Ability to interpret diagrams of relays, contacts, coils.
- Ability to run confidence check programs.

Systematicity

- The subject returns to a point where he knew what was going on when a dead end is encountered.
- The path from the power source is checked.

Plans was a count of the number of plans mentioned by the subject during his problem solving efforts. Any time that the subject entered a new part of the problem space, we prompted for a plan, but the lower skill subjects, especially, often did not have one. That is, they more or less randomly acted until a plan or hypothesis came to mind. A count was kept of the number of hypotheses offered by subjects at various points in their work. Again, subjects were prompted for hypotheses at the predetermined boundary points between regions of the problem space. The high-skill newcomers entertained more hypotheses, which is what we would expect given that even they are at intermediate skill levels. True experts were be expected to have a more constrained set of probable hypotheses.

The *device and system understanding* scale was based upon specific questions that were put to the subjects after they had performed the troubleshooting tasks. We asked a fixed set of questions about each of the components of the test station that played a role in the problems we had posed. These questions probed for knowledge about how the component worked, what role it played in the test station, what its general purpose in electronic systems was, and what it looked like.

The *errors* scale was simply a count of the number of incorrect steps taken by the airman in trying to troubleshoot the system.

The *methods and skills* measure tallied which of the procedures needed to carry out the troubleshooting of the test station were successfully demonstrated by the subject.

Finally, the *systematicity* measure consisted of a set of relatively broad criteria gauging the extent to which troubleshooting proceeded in a systematic manner rather than haphazardly or without a sense of goal structure.

This, then, provided a first, relatively global view of the skills that separated high and low skill first-term airmen. We found out quite a bit from these analyses. High skill airmen differed on most of the measures, except for *Plans*. Looking a bit more closely, the higher-skill airmen seemed not to be less inclined to engage in metacognitive activity; rather, their planning was more general, less conditioned to the specific point in a specific problem solution at which they found themselves. This suggested to us that it might be necessary for us to teach very much in the context of the skill domain. It wasn't that the high-expectation airmen were more likely to use general planning strategies. Rather, they were only better at concrete, domain-specific variants of those planning methods. The missing knowledge to be taught to less-promising airmen was domain-specific planning, plus some specific skill components, such as tracing schematics and using meters for active-circuit measurement. Thus, we had a basis for specifying our curriculum.

However, the knowledge coming from the six scales, or even from slightly more specific tallies, was insufficient to drive the specification of our problem types. To do that, we needed to be able to look at additional qualitative aspects of job-domain performance. An impressive example of how to do this was provided by Christopher Roth, an LRDC student working at HumRRO in Alexandria, Virginia. He took our methods and applied them to a slight variant of the domain we studied. He also used the same domain expert (Gary Eggan). However, he spent more of the total effort comparing the specific problem-space paths of experts and novices. The next few figures illustrate the yield of this work.

Roth worked with Eggan to develop a collection of diagnosis problems that were characteristic of real problems faced in the work environment and that were likely to be solved correctly by the best technicians and not by many of the less-skilled first-term airmen. People with experience in handling the tasks on which many airmen have difficulty tend to have a good sense of the kinds of problems that are in this class (problems that are representative of the domain, solvable by the better technicians, and not solvable by a substantial number of workers), and Eggan's advice has proven itself empirically. Figure 2 illustrates the effective problem space that Eggan helped Roth construct for the problem. It was felt that the bulk of problem solving activity would fall within this space, for both experts and less-skilled

Figure 2
Effective Problem Space

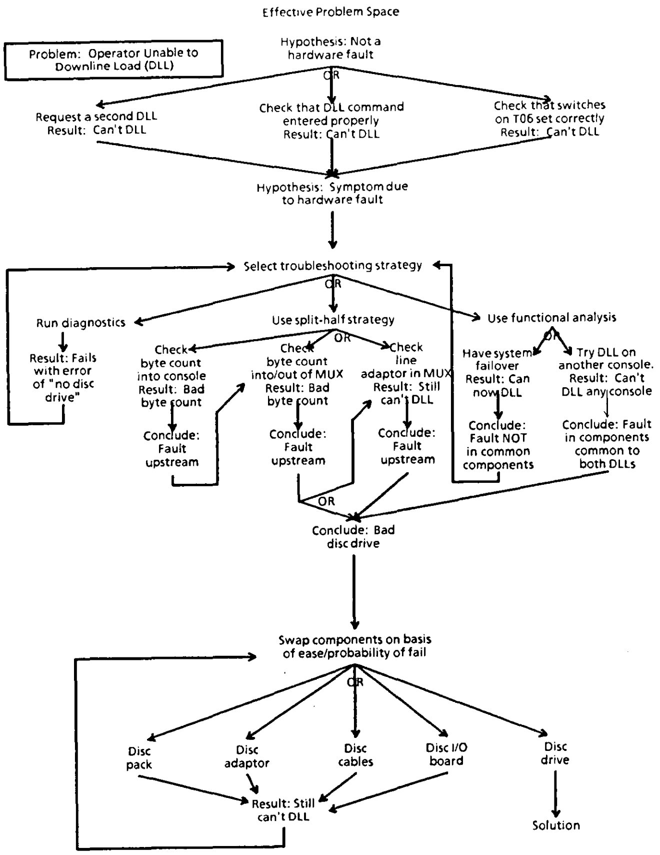
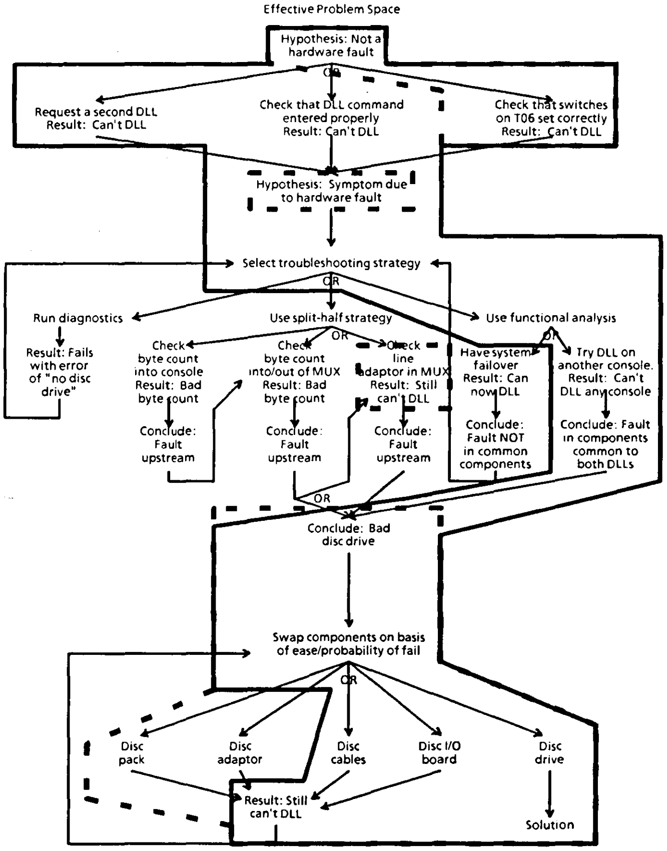

Effective Problem Space
Problem: Operator Unable to Downline Load (DLL)
Hypothesis: Not a hardware fault
OR
Request a second DLL Result: Can't DLL
Check that DLL command entered properly Result: Can't DLL
Check that switches on T06 set correctly Result: Can't DLL
Hypothesis: Symptom due to hardware fault
Select troubleshooting strategy
OR
Run diagnostics
Result: Fails with error of "no disc drive"
Use split-half strategy
OR
Check byte count into console Result: Bad byte count
Check byte count into/out of MUX Result: Bad byte count
Check line adaptor in MUX Result: Still can't DLL
Conclude: Fault upstream
Conclude: Fault upstream
Conclude: Fault upstream
Use functional analysis
Have system failover Result: Can now DLL
Try DLL on another console. Result: Can't DLL any console
Conclude: Fault NOT in common components
Conclude: Fault in components common to both DLLs
OR
Conclude: Bad disc drive
Swap components on basis of ease/probability of fail
OR
Disc pack
Disc adaptor
Disc cables
Disc I/O board
Disc drive
Result: Still can't DLL
Solution

Figure 3
Expert and Novice Subsets of Effective Problem Space


Effective Problem Space
Hypothesis: Not a hardware fault
OR
Request a second DLL Result: Can't DLL
Check that DLL command entered properly Result: Can't DLL
Check that switches on T06 set correctly Result: Can't DLL
Hypothesis: Symptom due to hardware fault
Select troubleshooting strategy
OR
Run diagnostics
Result: Fails with error of "no disc drive"
Use split-half strategy
OR
Check byte count into console Result: Bad byte count
Check byte count into/out of MUX Result: Bad byte count
Check line adaptor in MUX Result: Still can't DLL
Conclude: Fault upstream
Conclude: Fault upstream
Conclude: Fault upstream
Use functional analysis
OR
Have system failover Result: Can now DLL
Try DLL on another console. Result: Can't DLL any console
Conclude: Fault NOT in common components
Conclude: Fault in components common to both DLLs
OR
Conclude: Bad disc drive
Swap components on basis of ease/probability of fail
OR
Disc pack
Disc adaptor
Disc cables
Disc I/O board
Disc drive
Result: Still can't DLL
Solution

workers.

Roth and Eggan then proceeded to run subjects on this task. The problem was posed verbally to airmen who had access to the full set of Technical Orders (documentation for the devices involved). Technicians were encouraged to offer their hypotheses, state their plans, and announce specific steps they would take in attacking the problem. Whenever an overt step was stated by a technician, he was immediately informed of the outcome of that step.

Figure 3 shows the basic results that Roth obtained. The experts, whose performance is outlined with a solid line, used a subset of the effective problem space, avoiding some of the steps that were less likely to be productive. The novices, whose empirical problem space is outlined in heavy dashes, failed to make some of the expert moves and did make some moves that experts would not make, such as swapping a disc pack (see bottom left of Figure 3. Further, their empirical problem space was discontinuous, a set of islands.

To better understand what knowledge separated the experts from the novices, Roth and Eggan then tried to determine the specific knowledge that experts used at critical points in the problem space. Knowledge involved in making a move that experts make and novices do not is a clear candidate for inclusion in the curriculum. At the individual problem tutoring level, a wrong choice at a decision point in the effective problem space then becomes a mandate for specific instruction on the knowledge that could have led to correct performance.

Several things became clear from this work. First, there will be a lot of variability from one trainee to the next, so that the coaching for this sort of problem solving performance cannot be prespecified completely. Rather, some online intelligence will be required. Second, the summary measures of planning, systematicity, etc., do not provide the detail on which instructional design and specific coaching must be based. Third, the validity of any conclusion about what a trainee is doing during an effort to solve one of our problems must rest on a pattern of logical relationships among the observed details of performance, the expert knowledge of the domain, and the cognitive psychology of expertise. That is, it is reasonably easy for an electronics wizard and a cognitive psychologist to get a good picture of what knowledge a novice has or does not have, through cognitive extensions of rational task analysis approaches. This is a form of diagnosis more like what a physician does in an individual case than like what a testing expert does. Fourth, the scientist can reasonably ask what empirical support there is for this approach. Several successful analyses of technical jobs have taken place, in three different laboratories, but continued work is needed.

Simulating the Work Environment

I turn now to a totally different kind of tool, a software tool. Given the goal of building a problem-based tutor, we need to be able to simulate the job world in which the problem solving takes place. Here, I also have an interesting discovery to report. Like a number of laboratories, ours has been working on tools for simulating devices. By a device, I mean a network of objects that pass information between themselves. for example, a resistor circuit can be thought of as a set of objects, batteries and resistors perhaps, that pass information about current and voltage along paths that correspond to wires. To simulate a resistor circuit, we need to specify a method for each object that permits it to demonstrate its behavior. For example, a resistor is an object that, given modifications in its resistance, changes the current it will pass at a given voltage, according to Ohm's Law.

Of course, a device simulation must also model the behavior of the device as a whole, and that overall behavior is constrained by properties of its components, too. Further, simulated of response to changes must be instructive, which may require a departure from reality at times, but that is not the issue I want to address in this paper. What is critical for the present purposes is that a device model must be prepared to illustrate both local component behavior and overall system behavior.

It rapidly occurred to my colleagues Jeffrey Bonar and John Corbett that this loca/global distinction can be applied recursively. A computer has as components a number of subdevices; they in turn are composed of boards; the boards are composed of chips; many of the chips are arrays of logic gates. Thus, what we really have to be able to simulate is an entity that has an overall functional description and also a description in terms of the behaviors of its individual components. Corbett has built a system that can do this, called Flowchips. It is still evolving, but we already know that it is a very powerful tool, both for providing an environment in which problem solving activity can take place and also for providing explanatory simulations of complex objects.

The next step in the story represents a stroke of genius, for which Bonar and Corbett, not I, deserve the full credit. They realized one day that hierarchical displays of devices are no different from hierarchical displays of texts. Indeed, texts can be included as components of devices, too. They are special components, perhaps, that don't necessarily change; or they are labels; or they are function descriptions that changed as parameter thresholds are crossed, as in labels on a capacitor that might say *charging* or *discharging* or *saturated*. They can also be terse statements of function that can be "opened up" to reveal more detailed statements of function, just as a chip might be opened up to reveal its logic-gates contents.

In the limit, then, a device model is more or less like a hypertext system. Information is presented at multiple levels of detail in multiple media (for us, currently, line drawings and text). However, it also has some special properties. The entities of which it is composed are not passive. They propagate changes in themselves throughout the simulation system; they respond to the student or trainee not only by elaboration, but also by demonstration, by explanation, and most importantly by affording opportunities for manipulation.

In an idea processor, the user can change an entry, but the change is passive. In the Flowchips system now under development by Corbett and Bonar at the Learning Research and Development Center, the user can make changes that affect the entire system and can then open up the system to examine those changes. For example, changing a few switch settings might totally change which parts of a complex circuit are active and also change the texts that explain what different parts of the circuit are currently doing. It is this level of device simulation that we believe is crucial.

One final viewpoint on such change propagations is worthy of note. If a change that has to do with a parameter being simulated, such as voltage, can be propagated through the system then it should also be possible to propagate changes having to do with pedagogy, such as advice that a particular student reads poorly, has trouble interpreting graphics, speaks only a particular language, is a computer scientist, not an electrical engineer, etc. Hopefully, we can move beyond our current efforts, which are much more mundane, and consider adding directly to the device simulation two other component objects, a student model and a teacher of sorts.

In a real device, certain changes in parameter values can be catastrophic. For example, pouring cold water into a very hot steam plant can be deadly. In the kinds of device simulations that now appear to be possible, whether the simulation explicitly makes a big fuss about such an act should be mediated by an intelligent system that decides whether focusing attention on the catastrophic implications of the last step taken will be useful in teaching the student. Similarly, while there are many aspects of a circuit to which an expert should attend, an active simulation might elide over some components and concentrate on others to which a novice should be particularly attentive or which relate to an active instructional goal.

SOFTWARE SYSTEMS ARENA

Software Engineering

TRACK CHAIR: Prof. Laszlo Belady
MCC

Unix

TRACK CHAIR: Prof. Domenico Ferrari
University of California at Berkeley

TRACK CO-CHAIR: Dr. Luis Felipe Cabrera
IBM Research Laboratory

Program Testing

TRACK CHAIR: Prof. William Howden
University of California at San Diego

Programming Languages, Compliers and Environments

TRACK CHAIR: Dr. John White
Xerox Corporation

AN APPROACH TO TYPE SAFETY IN A TRAITS SYSTEM

Gael Curry

Sequent Computer Systems

Abstract

This paper extends some of the multiple inheritance subclassing ideas developed for use in early versions of the Xerox Star workstation. It first shows how to capture the class hierarchy in compiler-visible form. Next it shows how to attribute a useful notion of "type" to instances. Then it shows how a compiler and a programmer relate to this approach.

1. Introduction

This paper describes an approach to supporting type-safe multiple-inheritance object-oriented programming. It is the outgrowth of experience with traits as supported in the Mesa language for Xerox Star.

For completeness, the paper will first describe what traits are, how they were used in Star, and what the problems were.

Then, it will describe how to embed a multiple-inheritance class hierarchy in compiler-visible structures, and discuss the meaning of type-safety in these terms.

2. Traits

This section reviews the central ideas behind traits and compares them to similar facilities in other environments.

2.1. Basic Concepts

A "trait" is a property that an object may have which allows clients to interact with the object in a certain way. To a client, a trait appears as a set of operations, any of which may be applied to any object carrying the trait. For example, the trait "stores_integer" may have two associated operations: put_integer and get_integer.

A trait does not capture the essence of an object. It captures the essence of a property of an object. In this respect it is different from an "abstract data type". Traits are, however, similar to abstract data types in the sense that they are information-hiding encapsulations of (properties of) objects.

Traits can be mixed rather freely in objects. Often primitive traits are independent of other primitive traits, such as "stores_integer" and "has_name". Under these circumstances, it is quite natural for objects to carry several traits.

Several traits can also be combined into more complex traits. For example, "is_integer_variable" could be considered to be the combination of "stores_integer" and "has_name". These compound traits can then be carried by objects.

A "class" (for a given trait) is something which generates instances whose only property is the trait, and, by implication, its subtraits.

Traits are described in greater detail in [Curry81] and [Curry83].

2.2. Similarities to Other Environments

The idea of primitive abstractions which can be combined into more complex ones has been used elsewhere.

In Smalltalk [Goldberg83], the rough analog to trait is "abstract class", a class without instances. The "carries" relation on traits corresponds to the "superclass" relation on classes. See [Borning82] for a description.

In KEE, the rough analog to traits are "units". See [Intelli83].

CH2345-7/86/0000/0025$01.00 © 1986 IEEE

In various versions of LISP, "flavors" are the analog to "traits" [Weinreb80]. The LOOPS extension to Interlisp also has corresponding concepts [Bobrow83], [Stefik85].

A comprehensive comparison of the various approaches can be found in [Carnese84].

2.3. Differences from Other Environments

While conceptually similar to the various other mechanisms previously mentioned, traits were motivated by different types of concerns. The approach arose out of the need to control the complexity in the Star implementation by increasing the reusability and sharability of common program fragments. Subclassing first, and multiple-inheritance second, helped to accomplished this.

There was another point of departure. It was vital that the implementation of multiple-inheritance be lightweight. For this reason, message-passing approaches to supporting multiple-inheritance were rejected. Message-passing is an expensive way of implementing inheritance in cases where the very dynamic binding permitted by message-passing is not required.

Since it was acceptable to hold the class structure constant for a particular build of Star, it became possible to precompute (at load-time or earlier) all of the values inherited by a class.

A more subtle difference between traits and more dynamic systems also exists. The central traits notion is that of an abstract specification of a property of an object; the semantic specifications of trait operations are the constants. Other schemes seem to focus more on message passing; message names are the constants. At issue is whether alternate methods selected by a given message need to meet a given specification.

3. Traits in Star

This section discusses the use of traits in Star: the programming conventions, binding times, and type-safety. The casual reader can skip this section.

Traits were used extensively to build the early versions of Star. Versions of Star released in 1983 included some 260 different traits.

The Star programmer dealt with traits as a pattern of programming conventions over the Mesa language.

3.1. Mesa

Mesa was the systems implementation language for Star. It is a strongly-typed, modular language [Geschke77], [Mitchell79]. "Definitions modules" represent the abstract interface to "program modules," which implement the interface.

The strong-typing of Mesa had a deep influence on the Star programmer. Because Star was such a large piece of software - on the order of a thousand program modules - any support for maintaining and making comprehensible large software systems was welcome. The Mesa compiler's support for strong-typing was invaluable; breaches of the type system (which traits seemed to require) were frowned upon. The desire for a type-safe way of supporting traits led to the thoughts behind this paper.

3.2. Trait Client Programming Pattern

A trait appeared to its client as a Mesa definitions module. The module would contain a set of procedural interfaces for trait operations. The first parameter of each procedure was the handle to the object (allegedly) carrying the trait. It might be declared as:

```
     "Op: PROC[oH:Handle, t:T]".
and would be invoked as:
     "Trait.Op[oH, t];"
```

It was the client's responsibility to ensure that, before applying a trait operation to an object, that the object did in fact carry the trait. In general, there were no compile-time or run-time guarantees that it did. It was a serious error when a trait operation was applied to an object not carrying the trait.

3.3. Trait Implementer Programming Pattern

Implementing a trait was more involved than simply using one as described above. The implementer of a trait had to obey certain programming conventions since traits were not supported directly by the language.

Each instance reserved storage for all the traits that it carried. The conventions required that each trait declare how much storage it needed in instances which would carry it. For example, the trait "stores_integer" would require enough storage for one integer in every object that carried it.

In addition, each class reserved storage for all the traits that it carried. This storage was used exclusively for storing procedure values reflecting the inherited "methods" of the class. The conventions required that each trait declare how many overridable operations it had.

A "trait manager" (TM) existed which represented compile-time and run-time knowledge of global aspects of the trait hierarchy.

Each trait declared certain things about itself to TM (e.g., amount of trait storage required in each instance carrying the trait, amount of trait storage required in each class carrying the trait, whether the trait would generate instances, which (sub) traits it carried). Each trait registered this information with TM at load-time. Logically, this could have all happened at compile time.

The TM used this information to determine the locations of trait data in instance storage, and of trait data in class storage. Traits could query TM to determine where their data was in instances and classes. For accessing its data, a trait would ask TM to help it open a scope onto its data in the instance:

```
     put_integer(oH: Handle, i:int]
          p = TM.Data[oH, me];
          p.storage = i;
```

In addition, where carrying traits (subclasses) were able to override this trait's methods with their own, the trait implementer needed to provide a "switch". The pattern was similar; a trait would ask TM to help it open a scope onto its data in the class:

```
     put_integer(oH: Handle, i:int]
          p = TM.Ops[oH, me];
          p.put_integer[oH, i];
```

Initialization of instances and classes was done by explicit bottom-up traversal of the traits hierarchy by each individual trait. That scheme was fine in the single inheritance world (where there was only one immediately carried trait, or "superclass"), but broke down completely for multiple-inheritance. A better approach would have been for TM to drive the static initialization of instances and classes, rather than conversely.

The pattern was too involved to be satisfactory. The whole scheme could have been supported much more cleanly by a compiler (if the information known to the Star traits manager were only available at compile-time).

3.4. Pre-calculated inheritance

The class information massaged by TM was in fact static. All method overriding was performed at boot-time, and recorded in the class structure. It could as well have been specified at compile-time (this is not necessary for type safety as described below).

3.5. Pre-calculated initial instances

Similarly, statically initialized instances for each class could have been calculated at compile-time.

3.6. Optimized instance, class layout

When a trait accesses its data in instances or in classes, it does so via TM. It actually has no idea where its data is with respect to the class or instance. It is the association of trait storage with instances that is important, not the location of the storage. In many cases, it is possible to arrange that, for all classes, trait data is in the same object-relative location (class-independent positioning). This can be used to generate optimal accesses to trait data.

For single-inheritance systems, class-independent positioning is always possible. For multiple-inheritance systems, it is easy to show that it is sometimes impossible if no "holes" are permitted in instance storage. In general, class-independent positioning of trait data in class and instance storage is excessively expensive.

The Star trait manager originally calculated a globally optimized layout for trait data in instances and classes. A recompilation of Star with this globally optimized layout made trait data access even faster.

3.7. Type-safety

There were many places where the Mesa type system was breached by the traits programming convention. We were unable to find a safe pattern which still retained the information-hiding benefits.

4. Problems with Traits in Star

While traits worked well enough to get the job of building Star done, they had certain drawbacks.

This section describes the major problems encountered which using traits in Star.

4.1. Traits were unsupported

The most basic problem was simply that that style of programming was unfamiliar. As a language feature it was unsupported by the production tools

4.2. Runtime support was required

One of the more interesting aspects of multiple-inheritance is that runtime support seems to be required to locate trait data in instance storage (and class storage), at least when no gaps are permitted in instance storage. For single-inheritance systems, the convention of placing superclass data closest to the base of instance storage provides class-independent positioning for all traits. For multiple-inheritance systems, there is no convention which assures class-independent positioning for all traits. There will always be a trait whose data (in an instance) can only be located by accessing a runtime which can only be constructed after all traits' implementations are compiled.

If gaps are permitted in instance storage, a compile-time database of trait data locations can eliminate the need for runtime access.

4.3. Global optimization was performed

The Star traits mechanism went one step beyond providing a runtime. It tried to minimize access to the runtime by doing an analysis of the entire traits hierarchy in order to achieve class-independent positioning as often as possible. From a development standpoint, the optimization was not a good idea, since addition of a single new trait could require recompiling all trait implementations.

4.4. Traits breached Mesa type system

Perhaps the most serious problem with traits in Star was the fact that its use fostered systematic breaching of the Mesa type system.

For the trait client the danger lay in invoking a trait operation on an object which did not carry the trait. Eventually, a run-time check was added; the object's "create type", the highest level trait it carried, had to carry the trait introducing the operation.

For the trait implementer, the problem lay in the way a scope was opened on trait data in class and instance storage; a (generic) TM function was used, returning a pointer which was then coerced into a trait-specific type.

In practice, surprisingly, these breaches did not cause problems very often. Their existence was disturbing, however.

In addition, the only checks which could be implemented were run-time checks. This deferred detection of what was basically a type breach to late in the development cycle.

5. Elements of a Solution

Static typing is always a compromise. In exchange for compiler support for the more static aspects of a program, the programmer gives up some dynamic control.

The most important element of type-safe multiple-inheritance traits programming is to embed the trait hierarchy in structures which are identifiable by a compiler. At the very least, trait "names" and the "carries" relation should be visible; the compiler needs to be able to determine whether one trait carries another or not. For type-safety, only this is necessary. It is also useful if method inheritance can be precalculated at compile-time; cases of inconsistent inheritance can be resolved by the compiler. It is more efficient to calculate the value of statically-initialized instances at compile-time also.

The second necessary element for type-safe trait programming is a change of viewpoint about what "type" is.

Conventionally, variables which hold values of a given type are tagged with the "create type" of the variable, reflecting total knowledge of the characteristics of the value. For example a variable which holds an object of the (earlier-described) type "is_integer_variable" would be tagged with "is_integer_variable". This notion of type does not help with type-safe traits programming.

An alternate approach is to tag variables holding values of a specific type with (sub) traits carried by that type, reflecting partial knowledge of the characteristics of the value. In the case above, a variable holding an object of type "is_integer_variable" might be tagged with "has_name". This tagging allows the programmer to apply the operation "get_name" (for example) to the object. Even though the operation "get_integer" makes sense, it may not be invocable.

Syntactically, one might write:

```
     <var>: Handle CARRYING <trait>
```

to declare <var> to hold handle values for objects known to carry <trait>. A more general approach allows the programmer to express partial knowledge that an object carries a set of traits:

```
     <var>: Handle CARRYING <traitlist>.
```

"Type-safety" for trait clients means that it is not possible to apply a trait operation to an object which does not carry the trait.

6. A Static Trait Hierarchy

This section describes how trait information can be embedded in compiler-visible structures - trait interfaces. A trait interface is the means by which trait clients know the types and operations introduced by the trait. In Mesa, a trait interface would be represented as a "definitions module".

A trait interface contains a single "trait declaration". A trait declaration includes a "trait id" and a list of carried traits (we exclude referential circularity). Note that a trait interface may be shared indirectly via different paths. The importance of the trait declaration is that it embeds a description of the traits graph (i.e., carries relation) in the set of trait interfaces, in a form which can be understood by the compiler.

A trait interface contains a number of operations introduced by the trait and which can be applied to objects carrying the trait. These are called "trait operations". Every trait operation has a distinguished parameter, which identifies the object to which the operation is to be applied. This parameter must be of type

```
    Handle CARRYING <trait>,
```

where <trait> is the trait being declared.

In a trait interface, each trait operation is described. For each operation, the name, input and output parameters and their types are listed. The semantics of each trait operation are described. As is usual with interfaces, the "implementations" of trait operations are usually not specified.

A trait interface specifies whether each trait operation is rebindable and if so, a preference for that operation. A syntax such as:

```
      t: TRAIT[ ... ];
      THandle: TYPE = Handle CARRYING t;
      Op: TRAIT PROC[oH: THandle, ...]
            PREFER OpDefault;
```

could be used. Preferences are not permanent bindings. That is, invoking a rebindable trait operation bound with PREFER cannot be guaranteed to invoke the preference. For example, if the trait operation is being applied to an object carrying a higher level trait which overrides the preference expressed here with its own, then the preference of the higher-level trait will be invoked instead.

Each trait interface specifies preferences for its rebindable operations. Higher level traits must be able to override preferences of traits they carry. A variant of the PREFER verb can be used. For example, if the declared trait carries a trait declared in another trait interface "T", and "Op" is a rebindable operation declared in "T", we could use:

```
    T.Op:  PREFER OpOverride.
```

This means that the trait defined here is "overriding" the earlier preferences with its own.

This ability of higher-level traits to override preferences of lower level traits is the basis for much of the power of traits. In more dynamic subclassing systems the same purpose is accomplished by intercepting messages and executing alternate "methods". In a traits system, this "message-passing" overhead would be incurred at compile time instead of run time.

The traits hierarchy implicitly expressed in the set of trait interfaces defines operation inheritance paths. It a higher level trait "Th" carries a lower level trait "Tl", then operations of Tl can be applied to objects carrying Th.

Notice that from the perspective of a given trait interface, everything about the trait hierarchy below it is known. The trait operations for this and all carried traits are known, and preferences for rebindable operations for this and all carried traits can be calculated. If the trait is a class, the inherited methods for the class can be calculated and conflicts resolved.

7. Client Perspective

From a trait client perspective, things are straightforward. References to objects are maintained as handles carrying the needed traits. For example, a "page" object could be composed of "heading", "footing" and "body" objects. "page" trait data could be declared as:

```
     RECORD[
          head: Handle CARRYING heading,
          body: Handle CARRYING body,
          foot: Handle CARRYING footing
     ].
```

An Object whose handle is stored in page.body may be a good deal more elaborate than "body", but for the purposes of "page", it does not matter. The "page" object will only deal with the object at the level of "body" or below.

If a client wishes to apply an operation introduced by a lower level trait to an object, the compiler can validate and perform the necessary coercion, since it sees the entire traits hierarchy. For example:

```
    Object.Print[page.body].
```

In this case, page.body carries the "body" trait, which we assume carries the "object" trait. The compiler notices that the programmer is attempting to apply an operation introduced by "object" (namely Print) to an object known to carry the "body" trait. This is valid exactly when "body" carries "object", a simple reachability check on the traits graph.

On those occasions when a programmer "knows" than an object carries a higher level trait than the source guarantees, he may coerce it explicitly. A runtime check that the object really does carry the trait can be performed to maintain safety.

8. Applications

The real value of traits is reusability. Traits offer a way of capturing essential abstractions more easily than single-inheritance subclassing schemes. In order to be truly useful this way, traits should be implemented over some widely used programming language. In addition to being based on a widely-used language, an extensive library of abstractions is desirable.

The approach taken by the language C++ [Stroust86] is perhaps the best way to deal with the need. In this case a preprocessor "virtually" extends a standard, widely-used language.

Mesa, Modula, and ADA are perhaps the languages best suited for these extensions, since they provide strong-typing support already, and support for interface modules.

With some work, however, the approach may be applied to FORTRAN as well.

9. Conclusion

This paper has indicated how to extend a strongly-typed language to support traits-based programming. There are two key ideas. First, embed the trait hierarchy in a set of statically defined interfaces. This potentially eliminates message-passing as a means of implementing inheritance (since the compiler can now manage inheritance and type checking). Second, rely for type-safety on a notion of "type" which reflects static knowledge that an object carries a given trait, rather than the notion of "create type". The compiler can then assist in automatic coercion and type-checking, based on its understanding of the (statically-specified) trait hierarchy.

10. References

[Bobrow83]
Bobrow, D. and Stefik, M. The Loops Manual. Technical Report, Xerox PARC, December, 1983.

[Borning82]
Borning, A. and Ingalls, D. "Multiple-Inheritance in Smalltalk-80". Proceedings of the American Association for Artificial Intelligence, 1982.

[Carnese84]
Carnese, D. "Multiple Inheritance in Comtemporary Programming Languages", Masters Thesis, MIT AI Lab, September, 1984.

[Curry81]
Curry. G.; Baer. L.; Lipkie, D.; Lee, B. "Traits: An Approach to Multiple-Inheritance Subclassing", SIGOA Conference on Office Information Systems, Philadelphia, June, 1982.

[Curry83]
Curry, G.; Ayers, R.. "Experience with Traits in the Xerox Star Workstation", IEEE Transactions on Software Engineering, September, 1984.

[Intelli83]
IntelliGenetics, Inc. "KEE User's Manual". 1983.

[Geschke77]
Geschke, C.; Morris, J.; Satterthwaite, E. "Early Experience with Mesa". CACM 20(8):540-553. August, 1977.

[Goldberg83]
Goldberg, A.; Robson, D. Smalltalk-80 - the Language and its Implementation. Addision Wesley, 1983.

[Mitchell79]
Mitchell, J; Maybury, W; Sweet, R. Mesa Language Manual. Technical Report, Xerox PARC. 1979.

[Weinreb80]
Weinreb, D; Moon, D. Flavors: Message-passing in the Lisp Machine. Technical Report AIM-602, MIT AI Lab, November, 1980.

[Stefik85]
Stefik, M; Bobrow, D. "Object-Oriented Programming: Themes and Variations," AI Magazine, Winter, 1985.

[Stroust86]
Stroustrup, B. The C++ Programming Language. Addison Wesley, 1986.

Object-Oriented Programming for Macintosh Applications

Larry Rosenstein Ken Doyle Scott Wallace

Apple Computer, Inc.

Abstract

One of the attractions of the Apple® Macintosh™ is the uniform user interface across applications. Users judge a Macintosh program not only by its functionality, but also by its adherence to the user interface guidelines. Implementing the standard user interface is tedious, however, because of the many details that each developer must program.

This paper describes an object-oriented system, called MacApp,™ that is designed to improve both programmer productivity and user interface consistency. MacApp consists of a set of object types that implement the standard Macintosh user interface. Programmers customize MacApp not by editing its source code, but by defining new object types that override methods in MacApp.

Introduction

Apple's Macintosh computer is noted for the consistent user interface across applications. Users judge Macintosh programs not only by their functionality but also by their adherence to the established user interface guidelines. For example, they expect applications to use windows and pull-down menus, to transfer data from one program to another, and to provide an undo operation for every command.

User interface consistency is encouraged by the *Macintosh Toolbox*,[1] which is a set of several hundred procedures and functions built into read-only memory. The Toolbox provides the low-level implementation of the standard user interface, including windows, menus, and scroll bars.

The Toolbox does not provide an overall structure for an application; it is up to each developer to call the correct Toolbox routines in response to user actions. Most of the source code needed to implement the standard user interface is identical in all applications. In addition, some of it can be tedious to write and debug. The time a programmer spends implementing the user interface could be better spent working on the specific features that make the application unique.

The typical approach to assisting programmers has been to provide a series of sample programs that they can modify to suit their needs. We have taken a different approach by implementing a generic Macintosh application, called MacApp. MacApp automatically handles the common user interface details such as moving, resizing, and scrolling windows. It is structured so that programmers can easily override the standard program behavior and add application-specific behavior.

MacApp is written using object-oriented programming techniques. It consists of a set of object types, each of which corresponds to an aspect of the Macintosh user interface, such as a document or window. An object type defines the behavior of instances of that type in terms of: (1) the state information stored in every instance and (2) the methods that act upon the state information. The Document type in MacApp, for example, includes the name of a disk file as part of its state and methods for opening the file, reading the document from disk, and closing the file.

Developers use MacApp by defining descendants of the basic MacApp object types, which automatically inherit the standard behavior of MacApp. They customize MacApp, not by editing its source code, but by overriding the methods they wish to change. In the example above, the programmer would define a descendant of the standard Document object type and override the method that reads the disk file with one that can interpret a particular file format. The methods that open and close the disk file, however, would be inherited from MacApp.

MacApp is similar in purpose to the Model, View, Controller classes of Smalltalk-80;[2, 3] both provide a standard framework within which to implement applications. The design of MacApp was also influenced by the Lisa® Toolkit,[4, 5] which was an earlier object-oriented system developed at Apple for the Lisa operating system.

Why Object-Oriented Programming?

There are two basic reasons for using object-oriented programming in MacApp.

First, object-oriented programming generally promotes better programming style.[5, 7] Programs are easier to maintain, because the interactions between objects are through well-defined method interfaces. Also, code can be more easily reused and extended through subclassing and inheritance.

The second reason involves the structure of a Macintosh application. All applications follow the same basic structure, at the heart of which is an event loop. The event loop reads an event from a queue, classifies it, and then processes it. Pressing the mouse button, for example, could be interpreted as choosing a menu command, moving a window, or selecting an icon, depending on where the user was pointing at the time.

Since the event loop is common to every application, it should be implemented once in MacApp and used by every application. This is a radical change from the typical application because the primary control loop of the application resides in MacApp, and is not implemented by the programmer.

Although MacApp can handle generic actions such as moving a window, it cannot handle application-specific actions such as choosing an icon. For the latter kind of actions, MacApp must make calls to procedures written by the programmer. This is inconvenient to do in a conventional programming language, because a procedure call must specify at compile time which piece of code to execute.

Object-oriented programming is ideal for this situation. Sending a message to an object corresponds to calling a procedure. The difference is that the method to be executed is determined at run time, based on the type of the object. Although the same effect can be achieved with pointers to procedures, the resulting programs are more readable and easier to maintain if they use object-oriented constructs.

Object Pascal

MacApp was developed using Object Pascal,[7] which is an object-oriented extension of Pascal. Object Pascal provides language constructs for defining object types, similar to the facilities of Simula or Smalltalk-80.[2, 3] Object Pascal is descended from Clascal,[4, 5] which is an earlier language designed for the Lisa computer. Two important design goals of Object Pascal were: (1) to make it as simple as possible to learn and use, and (2) to integrate the object-oriented extensions with the rest of Pascal.

An object type definition is written much like a record type definition. For example, the following defines a Shape object type:

```
Type Shape = Object
     bounds:    Rect;
     color:     Integer;

     Function   Shape.Area: Integer;
     Procedure  Shape.Draw;
     Procedure  Shape.MoveBy(dh, dv: Integer);

     End;
```

The main difference between a record definition and an object definition is that the latter can contain declarations of procedures and functions, which are the methods of the object type.

Another difference is that an object definition can contain an optional ancestor type designation. For example, the following two object types have Shape as their immediate ancestor object type:

```
Type Circle = Object(Shape)
     Function   Circle.Area: Integer; Override;
     Procedure  Circle.Draw; Override;
     Procedure  Circle.SetRadius(radius: Integer);
     End;

Type Triangle = Object(Shape)
     vertices:  Array[1..3] of Point;

     Function   Triangle.Area: Integer; Override;
     Procedure  Triangle.Draw; Override;
     Procedure  Triangle.MoveBy( dh, dv: Integer);
                                            Override;
     Procedure  Triangle.SetVertices(v1, v2, v3: Point);
     End;
```

The descendent object type inherits all the fields and methods of its ancestors, and can add new fields and methods. In the definition above, Circle inherits the bounds and color fields and the MoveBy method from Shape, and adds the new method SetRadius. An object type can also override methods defined by its ancestor; for example, Circle overrides the Area and Draw methods.

In this example, Shape is an abstract object type. There would not be any instances of Shape itself, only of descendants such as Circle and Triangle. The reason for defining a Shape object is to establish a standard interface that all kinds of shapes can share. As we will see below, it is possible to refer to generic Shape objects without knowing until run time if they will actually be Circle or Triangle objects.

When a programmer declares a variable of a particular object type, the variable is a reference to an instance of that type. In the current Object Pascal implementation, an object reference is a pointer to a pointer to the actual object data. (In the Macintosh terminology, a pointer to a pointer to a data block is called a handle.)

Fields of an object can be referenced much like fields of a record. For example:

```
Var   aCircle:   Circle;
      aShape:    Shape;

aCircle.bounds := aRect;
oldColor := aShape.color;
```

Methods are invoked using the same syntax:

```
aCircle.MoveBy(10, 20);
aShape.Draw;
```

Since Circle and Triangle objects inherit all the properties of Shape objects, they can be used in any situation where a Shape is required. In particular, the object reference 'aShape' could refer to a Circle, or a Triangle. Because aShape is declared as a Shape reference, the Object Pascal compiler restricts accesses to aShape to those that are defined in the Shape object type. For example, the compiler would flag the statement 'aShape.SetRadius(r);' as a syntax error, since Shape objects do not define a SetRadius method. This is different from Smalltalk, where the error would not be caught until the statement was evaluated.

The determination of which method implementation is called for a particular method call is made at run time. To illustrate this, consider the following procedure:

```
Procedure  MoveTinyShapes(aShape: Shape);
Begin
    If aShape.Area < medianArea Then
        aShape.MoveBy(100, 0);
    aShape.Draw;
End;
```

When the Object Pascal compiler compiles this procedure it has no way of knowing whether a Circle object or a Triangle object (or for that matter an instance of an entirely new Shape subclass) will be passed to MoveTinyShapes. Method calls such as aShape.Draw do not directly call a particular method. Rather, they call a special method dispatch routine that examines the object to determine its type and then refers to memory-resident method tables to determine which method to call.

MacApp

MacApp is a generic Macintosh application. By itself, it can be compiled into a runnable program that would have many features of the Macintosh user interface. For example, it would support multiple windows that can be resized and scrolled. The windows would be blank, however, since MacApp does not specify what should appear in the windows.

There are six basic object types that MacApp programmers use:

Application	Document	Window
Frame	View	Command

Each of these object types correspond directly to an aspect of the Macintosh user interface.

There is one Application object during the execution of a program. The Application object receives events such as mouse presses and distributes them to other objects. It also handles desk accessories and any commands that apply to the application as a whole, such as opening a new document.

Document objects contain the data that the program uses; for example, formatted text or a spreadsheet model. In addition to providing methods for manipulating data in memory, it also provides methods for saving the data onto the disk and later reading it back into memory. There can be more than one type of Document object in a MacApp program: an integrated application might define both word processing and spreadsheet Document types.

The Window, Frame, and View object types all deal with displaying information on the screen. Window objects represent Macintosh windows that can be resized and moved. Frames are used to partition a window into independent pieces, each of which contains a View object. For example, MacDraw™ windows contain a tools palette as well as a drawing area.

View objects translate between the Document's data structures and an image within a Frame. The main function of the View object is to draw the data when its Draw method is called. Before MacApp calls the Draw method, it sets up the graphics clipping and translation so that only the desired portion of the image is drawn. MacApp calls the same Draw method when printing each page of the document; in this case, the graphics state is set up so that the image appears on the printer rather than on the screen.

The last object type that MacApp programmers commonly use is the Command. Command objects provide a convenient framework for implementing the Undo command, an important part of any Macintosh application. Command objects encapsulate information about how to execute, as well as undo, an operation.

When the user selects a command from a menu, the application does not immediately carry out the operation. Instead, it creates a Command object and returns that object to MacApp. MacApp will then send a DoIt message to the Command. The DoIt method is responsible for carrying out the user's command and updating the image on the screen. It also must remember enough information so that the operation can be undone if necessary.

If the user selects Undo from the menu, MacApp sends an UndoIt message to the same Command object. The object restores the previous state of the Document and again updates the screen image. If the user selects undo again, MacApp sends the message RedoIt to the Command object.

A MacApp programmer will define a subclass of Command for each kind of operation that can be performed. For example, the Copy and Paste menu commands would be handled by different Command types. Often it is possible for one Command type to handle several menu commands; the Cut, Copy, and Clear commands are usually similar enough to each other that a single type of Command object can be written to handle all three.

In addition to the six basic object types described above, MacApp also contains support for other important application features; such as: creating and using dialog boxes, sending and receiving data over the AppleTalk network, and recovering from I/O and out-of-memory errors

Experiences with MacApp

Programmers, both inside and outside of Apple, have been using pre-release versions of MacApp since April, 1985. Of the approximately 200 people who have received MacApp, we have received comments from about 20 developers who have been actively using it in serious Macintosh applications. Three of these applications have reached "beta" test stage; one is a boat navigation program (which took six months to develop), another is a recording studio management program (nine months), and the third is a stuctured program editor and run time system (twelve months).

Before MacApp was developed, Macintosh programmers would spend several weeks reading *Inside Macintosh*[1] and example programs, in order to understand the structure of an application. Programmers who begin to use MacApp still spend several weeks reading the MacApp documentation. The important difference between the two approaches is that with the same investment of learning time, they end up with a much more functional application. By writing about three pages of source code, they can have a working application that has multiple scrolling windows, that reads and writes documents, and that prints.

Much of the difficulty in learning MacApp comes from its use of object-oriented programming. It takes time for a programmer who is not familiar with object-oriented systems to understand method calls and inheritance. In addition, MacApp requires thinking about an application in a different way, since most of the application control structure is contained in MacApp itself. Once developers overcome these obstacles, however, they are able to produce applications more quickly with MacApp than when using conventional programming techniques.

Programs written using MacApp follow the user interface guidelines more closely than most Macintosh applications. In addition, they are structured in a way that makes them easy to maintain and enhance. While developing MacApp, we talked to many Macintosh developers and incorporated their ideas and techniques into the final system.

There is a space and performance penalty for using Object Pascal. Each method call requires the same space as a regular procedure or function call; Object Pascal programs, however, also require memory space for method tables (about 3,000 to 5,000 bytes for a complete application). Method calls are also slower than regular procedure calls; the extra time required can range from 50 to 200 microseconds per method call.

The penalties for using MacApp in a serious applciation are much less, however, because most of a MacApp program is written in standard Pascal. We estimate that there is a 10 to 15% increase in application size and a 5 to 15% performance degradation, compared to a program written in standard Pascal. In most cases, these disadvantages are outweighed by the reduced development time and maintenance costs.

At the time this paper was written, we were beginning to look at ways to reduce the size of MacApp applications and improve their performance. Most of this involves standard code tuning that any application requires before being released. In addition, we are implementing an approach that will speed up method calls by analyzing the object type hierarchy.

Summary

We have developed an object-oriented system called MacApp that is intended to help programmers develop Macintosh applications. MacApp automatically implements the standard features of the Macintosh user interface and provides a convenient way for programmers to implement specific applications. Instead of editing the MacApp source code, programmers define new object types that override methods in MacApp, and inherit the standard behavior.

MacApp is implemented using Object Pascal, a version of Pascal that includes extensions for object-oriented programming. The object-oriented extensions were integrated with standard Pascal, so that programmers who already know Pascal could easily learn the language.

Applications written using MacApp follow the user interface guidelines more closely than most applications. In addition, MacApp encourages a program structure that makes applications easier to maintain and extend.

References

[1] Apple Computer, Inc. *Inside Macintosh*, Addison-Wesley Publishing, 1985.

[2] A. Goldberg, and D. Robson, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley Publishing Company, 1983.

[3] A. Goldberg, *Smalltalk-80, The Interactive Programming Environment*, Addison-Wesley Publishing Company, 1984.

[4] G. Williams, "Software Frameworks," *BYTE Magazine*, December 1984.

[5] K. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, 1986.

[6] B. Cox, *Object Oriented Programming, An Evolutionary Approach*, Addison-Wesley Publishing Company, 1986.

[7] L. Tesler, "Object Pascal Report," *Structured Language World*, volume 9, number 3.

CLASSES VERSUS PROTOTYPES IN OBJECT-ORIENTED LANGUAGES

A.H. BORNING*

* Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195

Abstract

Smalltalk uses classes to describe the common properties of related objects. Unfortunately, the use of classes and metaclasses is the source of a number of complications. This paper discusses prototypes as an alternative to classes and metaclasses. In a prototype-based language, copying rather than instantiation is the mechanism provided to the user for making new objects. Inheritance constraints are proposed as a way of representing object hierarchies and supporting the automatic updating of related objects when edits are made.

1. Introduction

The Smalltalk-80 language[1], as well as a number of other object-oriented languages, uses *classes* to describe the common properties of related objects. Unfortunately, classes and the class-instance relation are the source of a number of complications. First, for an object to have a distinct message protocol, a separate class must be created for it. If, as in Smalltalk, classes themselves are objects, then to allow different classes to understand different initialization messages, each class must itself be an instance of a different class (called a metaclass in Smalltalk). Metaclasses add to the complexity of the language; a recent study[2] on difficulties encountered in teaching and learning about Smalltalk indicates that metaclasses are uniformly regarded as the single worst barrier to learnability by both teachers and students. Second, the emphasis on classes in the programmer's interface is .at odds with the goal of interacting with the computer in a concrete way. When designing a new object, one must first move to the abstract level of the class, write a class definition, then instantiate it and test it, rather than remaining at one level, incrementally building an object. This problem is most apparent in systems for graphical or visual programming.

The alternative suggested in this paper is the organization of the programming environment around *prototypes* rather than classes. A prototype is a standard example instance; new objects are produced by copying and modifying prototypes, rather than by instantiating classes.

The remainder of the paper is organized as follows. The following subsections list some sources of complexity in the current Smalltalk metaclass-class-instance mechanism, and then describe a small gedanken experiment in language design, in which prototypes are used instead of classes. This very simple language has several limitations, and a more realistic design is presented in Section 2. Following this is an enumeration of the advantages and disadvantages of the proposal. The final section provides comparisons and references to related work.

1.1. Some Sources of Complexity

One source of the complexity surrounding classes in Smalltalk is the interaction of message lookup with the role of classes as the generators of new objects, which gives rise to the need for metaclasses. Another source is the use of classes for several different functions.

In Smalltalk, when an object receives a message, the interpreter goes to the object's class and looks in its method dictionary for a method for receiving that message. If a method isn't found, the interpreter looks in the class's superclass, and so on up the class hierarchy. Classes are themselves objects, and to make new objects, one sends appropriate messages to classes. In general these messages will vary from class to class. For example, to make new points, one wants to send an *x:y:* message to the class Point, so that the *x* and *y* coordinates for the new instance can be passed as arguments. Given the way message lookup is done, this requires that the class Point be an instance of a different class from (say) class Rectangle, which should not understand the *x:y:* message, but rather a different initialization message specific to rectangles. This pragmatic need for class-specific initialization methods was satisfied by the introduction of metaclasses: each class is an instance of a separate metaclass. However, as noted above, this design decision has had unfortunate consequences for the teachability and learnability of the language.

In regard to the use of classes for several different functions, some of the roles that classes play in Smalltalk are as follows:

- generators of new objects
- descriptions of the representation of their instances
- descriptions of the message protocol of their instances
- elements in the description of the object taxonomy
- a means for implementing differential programming (this new object is like some other one, with the following differences ...)

CH2345-7/86/0000/0036$01.00 © 1986 IEEE

- repositories for methods for receiving messages
- devices for dynamically updating many objects when a change is made to a method
- sets of all instances of those classes (via the *allInstances* message)

While some of the above items are related, it is clear that classes in Smalltalk are playing multiple roles.

1.2. A Gedanken Experiment in Language Design

To help unravel the complexity, let's do a small gedanken experiment in language design. Considerations of space and time efficiency are to be ignored for the moment, to avoid needlessly intertwining semantic and implementation considerations.

Suppose that objects are completely self-contained, so that an object consists of *state* and *behavior*. One can send messages to an object asking it for information, asking it to change its state, or asking it to change its behavior. The only way to make a new object is to make a complete copy of an existing object, copying both state and behavior. Once the copy is made, there is no further relation between the original and the copy. (Creating new objects by copying eliminates the need for metaclasses, since creation and modification messages are sent to prototypes or other individuals rather than to classes.)

This is a clean model, and would be easy to teach about. It handles object creation, modification, and representation. What is missing? First, there is no notion of classification of kinds of objects, either by message protocol or by representation. Second, there is no way to update a whole group of objects in a similar manner at one time (the equivalent of such actions as adding new methods to a class in Smalltalk). These are both important, and so the model needs to be augmented to support classification and updating.

2. A Proposal for a Prototype-Based Language

In this section a proposal for a more realistic language is presented, in which the simple model described above is augmented to support object classification and updating. Many of the ideas used here have arisen from the author's work on constraint-oriented languages and systems[3, 4, 5], where a constraint describes a relation that must hold. In this proposal, constraints are used to express inheritance relations among objects. However, the set of inheritance constraints used here is limited and straightforward to maintain, and a general-purpose constraint representation and satisfaction mechanism is not required.

In this proposed language (as in the language described in the gedanken experiment), an object has state and behavior. The state of an object is represented by a set of named *fields*. We will on occasion be interested in an object's field names, and this list of names can be accessed separately from the contents of the fields. There are two components of an object's behavior. The first component is a *method dictionary*, which is similar to that in Smalltalk, except that there may be several methods for receiving a given message. (The way in which one method is chosen, or several are combined, is discussed in Section 3.2.) The second component is a *protocol* that describes the set of messages the object declares that it can understand, the protocols required of the arguments to the messages, and the protocols of the results returned by the messages.

New objects are produced by copying other objects. Thus, to make a new point, one would make a copy of the prototypical point, with new values substituted for the *x* and *y* fields. Once a prototype has been copied, there would be no hidden relation between the prototype and the copy; any further relations that were desired would be explicitly represented using constraints.

2.1. Inheritance Constraints

The proposed language does not include a general constraint mechanism. Rather, there is a fixed set of *inheritance constraints*–constraints on an object's field names, methods, and protocol–built into the language. These are as follows:

- *inherits-field-names(x,y).* This constraint holds if every field name of *y* is also a field name of *x*.
- *inherits-behavior(x,y).* This constraint holds if all of the methods in *y*'s method dictionary are also in *x*'s method dictionary.
- *inherits-protocol(x,y).* This constraint holds if the protocol of *y* is a subset of the protocol of *x*, i.e., if every message that *y* can understand is also understood by *x*, and if each object returned by *x* in response to one of these messages also obeys the corresponding protocol declared in *y*.

In general these three constraints are independent. For example, an object *x* might inherit the protocol of *y* but not its methods (*x* would implement the necessary methods in completely different ways). If an object *y* does not use all of its fields, then *x* can inherit the behavior of *y* but not all its field names. Of course, the constraints are not totally independent–for example, if *x* inherits behavior from *y*, all of the field names used by *y* must also be field names of *x*.

Nevertheless, it will often be the case that these three constraints will be applied together, and so a *descendant* constraint is defined as follows:

$$\text{descendant}(x,y) \equiv \text{inherits-field-names}(x,y) \wedge \text{inherits-behavior}(x,y) \wedge \text{inherits-protocol}(x,y)$$

2.2. Object Creation Messages

There are two messages available for creating new objects: *copy* and *descendant*. The *copy* method makes a complete copy of the receiver and returns it. There is no further relation between the receiver and the copy. The *descendant* method makes a copy, and also sets up a one-way descendant constraint between the original and the copy.

2.3. Examples of Use

In place of the class Point would be a prototype point. The prototype point would have *x* and *y* fields, each initialized to 0. (Alternatively they could be left as *nil*). It would understand messages such as *+*, *printOn:*, and so forth. To make a new point, one would evaluate

```
point x: 4 y: 6
```

which would make a descendant of the prototype point, and then set its fields to 4 and 6. The code for *point x:y:* is as follows:

```
x: newx y: newy
  ↑ point descendant setx: newx sety: newy
```

The message *setx:sety:* is defined as in Smalltalk:

```
setx: newx sety: newy
  x ← newx.
  y ← newy.
```

The following messages would build a new kind of object, *threepoint*, and define an addition method for it.

```
threePoint ← object descendant.
threePoint hasFields: 'x y z'.
threePoint hasMethod: '+ p
     ↑ threePoint  x: x + p x  y: y + p y  z: z + p z'
...
```

Naturally, there would be user interface support for the creation and modification of prototypes. This could be done using a browser, which could have much the same appearance and functionality as the browser in the current Smalltalk system.

3. Implementation

The *descendant* method should be implemented as a primitive, and the primitives for *new* and *new:* eliminated. It might also be useful to implement *copy* as a primitive. To make new variable-length objects, one would make a copy or descendant of an appropriate prototype, and then grow or shrink the copy as needed. It might be useful to combine copying and growing in *descendant:* and *copy:* methods.

To test the scheme, it could be implemented using the present Smalltalk bytecode set by simulating the new primitives. Classes would be given an additional field named *prototypes* that points to the collection of prototypes for that class. (Usually, a class would have a single prototype, but multiple prototypes are possible.) Below is the Smalltalk code for simulating *Object copy* and *Object descendant*, along with some auxiliary methods.

```
copy
  "if I am a prototype, need to copy my class;
   otherwise just make a simple copy"
  self isPrototype ifTrue: [↑ self class copy prototype]
                  ifFalse: [↑ self simpleCopy]

descendant
  self beProtoype. "make sure that I'm a prototype"
  ↑ self simpleCopy

simpleCopy
  "return a complete copy of me, taking account of
   shared substructure. IdentityDictionary is a
   dictionary in which == is used for key comparison"
  ↑ self copyWithDict: IdentityDictionary new

copyWithDict: dict
    | copy f |
  (dict includesKey: self) ifTrue: [↑ dict at: self]
  "make the shell of the new copy, then fill it in"
  copy ← self class new.
  dict at: self put: copy.
  1 to: self class instSize do:
      [:i | f ← (self instVarAt: i) copyWithDict: dict.
          copy instVarAt: i put: f].
  ↑ copy

bePrototype
  "make me into a prototype, if I'm not already"
    | newSelf |
  self isPrototype ifTrue: [↑ self].
  newSelf ← self class newSubclass prototype.
  1 to: self class instSize do:
    [:i | newSelf instVarAt: i put: (self instVarAt: i)].
  self become: newSelf.

isPrototype
  ↑ self class prototypes includes: self
```

These general methods would be overridden for classes, and for primitive objects such as numbers.

3.1. Classes

For objects such as points and rectangles, the field names, method dictionary, and protocol will be the same for many objects. In an implementation, then, it is reasonable to group these together into a class. Further, rather than having multiple methods in a given dictionary, the methods could be partitioned among sub- and superclasses, as in Smalltalk. Thus, there would be classes as well as prototypes. However, classes in general won't have global names, there won't be any metaclasses, and the user will usually interact with a prototype rather than a class.

3.2. Multiple Inheritance

An object can have *descendant* constraints that relate it to several parents, i.e. multiple inheritance is supported. Each of the three constraints that compose the *descendant* constraint (*inherits-field-names*, *inherits-behavior*, and *inherits-protocol*) establishes the correct relation when multiple inheritance is used. The only difficult question is which method (or methods) to execute in response to a given message, if there are several conflicting inherited methods. However, this problem arises equally in class-based systems, and the same sorts of choices are applicable. For simplicity, for the present the rules described in the Smalltalk multiple inheritance implementation[6] are to be used.

4. Evaluation

In this section, a (doubtless biased) listing of the benefits and drawbacks of the proposed scheme as opposed to that in the standard Smalltalk-80 language is presented.

4.1. Benefits

Some benefits of this scheme over the current one are:

- The initial explanation of the language is much simpler. One can just talk about objects as having state and behavior.

- Even at a deeper level, there are fewer concepts–metaclasses are no longer needed. It is simpler to explain how message lookup occurs when teaching about the language.

- Fields are automatically given default values in newly created objects (by copying the contents of the corresponding field in the prototype).

- A prototype-based language would provide better support for concrete, visual programming systems.

- Any object can be given individualized behavior. This could be useful for example in debugging, when one might want to set a halt in the method for one particular object, and not for all instances of a class.

- The semantics of inheritance are described in terms of constraints; sharing is regarded as simply an implementation technique. This distinction might be useful when building distributed systems, systems that run on multiprocessors without shared memory, object servers, or the like. In such systems, sharing would *not* be the only technique used to implement inheritance.

4.2. Drawbacks

- The use of prototypes seems natural for things like windows, but unnatural for such basic objects as integers. What is the prototypical integer? The decision here is that *all* integers are prototypes, that is, that a change to the methods of any integer affects all integers. Alternate possibilities are that 0 or 1 is the prototypical integer, or else that the prototype is a special integer whose value is undefined. None of these possibilities seems completely natural.

- A related drawback is that the concreteness of prototypes may be inappropriate for describing such standard data structures as stacks or queues. Rather than talking about a prototype stack, one wants to talk about stacks in general.

- There is a danger of inadvertently modifying a prototype. One can of course inadvertently modify a class in Smalltalk, but this seems less likely since it has a different message protocol from its instances.

- There is an efficiency problem in regard to making new objects by copying. For example, when building a new rectangle the system will make a copy of the prototype rectangle, probably only to replace its *origin* and *corner* fields immediately with new values.

The first drawback listed above applies to the extreme situations in the language. This seems to be analogous to the situation in Smalltalk itself. Objects and messages are a great idea most places, but they seem bizarre for things like integers ("3 + 4 means sending the message + 4 to the object 3??"). However, the benefits of uniformity are such that making 3 be an object that understands messages is the right choice in Smalltalk.

Regarding stacks and similar data structures, the explicit existence of classes in this scheme may a help, since one can still talk about classes if one wants.

Regarding inadvertently modifying a prototype, I believe that the solution to this is not to introduce a different message protocol for prototypes, but to introduce some form of protection. For example, one might make prototypes read-only except in particular environments. One should also be able to designate some messages (e.g. *setx:sety:* for points) as being private, and allow this message to be sent only by *self*.

Regarding the efficiency problem, an obvious step is to code the *descendant* primitive efficiently. Beyond that, it appears that code sequences like *point descendant setx: newx sety: newy* will occur frequently. Therefore, in addition to the *descendant* primitive, for each object the system might automatically compile a method for e.g. *descendantWithx:y:*, which would accomplish the same thing, but more efficiently. This would call a new primitive that makes a clone of the receiver and then sets all of the instance fields in the clone. (Since this message exposes the object's representation, it is best regarded as a private initialization message.)

5. Related Work

The idea of prototypes is not new, and discussions of prototypes from a variety of perspectives appear in the literature. (However, the idea of using constraints to establish and maintain inheritance relations does appear to be new.)

One sort of system in which prototypes have often been used is systems for visual or concrete programming. In such applications, prototypes are more useful than classes, since it is more straightforward to display them for viewing and manipulation by the user; their concreteness also makes them valuable for less experienced users. Examples of systems of this sort that have been built in Smalltalk include ThingLab[3, 4], Programming by Rehearsal[7], the Alternate Reality Kit[8], and Animus[9, 10].

Languages in the Actor family are general-purpose programming languages that use prototypes. Rather than inheritance, the Actor languages use a more general concept of *delegation*, in which any object may be delegated to handle a message for another; Lieberman[11] provides a useful and readable discussion of both prototypes and delegation. LaLonde[12] describes an exemplar-based Smalltalk (an exemplar is the same as a prototype); this language allows a given class to have multiple exemplars, an idea that has been borrowed and used in the design described here. In Biggertalk[13], an object-oriented language implemented in Prolog, instances are like classes in all respects, except that they cannot be further refined. Finally, prototypes are often used in artificial intelligence representation languages[14] to store default or typical information.

The language proposed in Section 2 does not include type declarations. However, if type declarations were to be added, protocols would be the logical entity to use in the declaration and checking of type. An object-oriented language that does have strong typing, along with a separation of protocol and implementation, is Trellis/Owl[15].

Acknowledgements

Thanks to Dave Robson, Randy Smith, Adele Goldberg, Tim O'Shea, and other members of the System Concepts Laboratory at Xerox PARC, and to David Notkin and Andrew Black at the University of Washington. This research was sponsored in part by the Xerox Corporation, and in part by the National Science Foundation under grants MCS-8202520 and IST-8604923.

References

1. Goldberg, A.J., and Robson, D., *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, 1983.

2. O'Shea, T., "Why Object-Oriented Programming Systems Are Hard to Learn", *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM, September 1986.

3. Borning, A.H., *ThingLab -- A Constraint-Oriented Simulation Laboratory,* PhD dissertation, Stanford, March 1979, A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

4. Borning, A.H., "The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory", *ACM Trans. Programming Lang. and Systems,* vol. 3, no. 4, October 1981, pp. 353-387.

5. Borning, A.H., "Constraints and Functional Programming", Tech. report 85-09-05, Computer Science Dept, University of Washington, September 1985.

6. Borning, A.H., and Ingalls, D.H.H., "Multiple Inheritance in Smalltalk-80", *Proceedings of the National Conference on Artificial Intelligence,* American Association for Artificial Intelligence, Pittsburgh, August 1982, pp. 234-237.

7. Gould, L., and Finzer, W., "Programming by Rehearsal", Tech. report SCL-84-1, Xerox Palo Alto Research Center, May 1984, A shorter version appears in *Byte*, vol. 9 no. 6, June 1984

8. Smith, R.B., "The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations", *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages,* IEEE, June 1986.

9. Duisberg, R.A., "Animus: A Constraint Based Animation System", *Proceedings of the ACM CHI '86 Conference on Computer-Human Interaction,* ACM, Boston, April 1986, pp. 131-136.

10. Duisberg, R.A., *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System,* PhD dissertation, University of Washington, 1986, Forthcoming

11. Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM, September 1986.

12. LaLonde, W.R., Thomas, D.A., and Pugh, J.R., "An Exemplar Based Smalltalk", Tech. report TR-94, Computer Science Department, Carleton University, May 1986.

13. Gullichsen, E., "BiggerTalk: Object-Oriented Prolog", Tech. report STP-125-85, MCC, November 1985.

14. Fikes, R., and Kehler, T., "The Role of Frame-Based Representation in Reasoning", *Comm. ACM*, vol. 28, no. 9, September 1985, pp. 904-920.

15. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C., "An Introduction to Trellis/Owl", *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM, September 1986.

Why Properties are Objects or Some Refinements of "is-a"

Stanley B. Zdonik

Brown University
Department of Computer Science
Providence, RI 02912

Abstract

This paper contains several examples that illustrate some problems with the is-a relationship as defined by many object-oriented programming languages. These problems relate to two distinct areas: 1. the confusion between the inheritance of behavior and the inheritance of representation and 2. the lack of any requirement for semantic relationships between a named operation on a type and a replacement operation with the same name on a subtype.

We indicate how these problems can be improved by making some useful distinctions. We then show how these distinctions can be built into the system easily by treating properties as first-class objects and using the basic specialization techniques of the language to express the differences.

1. The Data Model

We have been developing an object-oriented database system that provides a data model that strongly resembles some of the type systems in many popular object-oriented programming languages. As a database system, however, all objects, including the types and operations, persist beyond the current session and are sharable between users and user processes. Our system is also strongly influenced by work in the area of semantic database models [ADP, Ch, Co, GAL, HM, MBW, S, SS].

Our model centers around the notion of a type. A type is a behavioral template for its instances. The behavior of a type T is expressed by a set of operations O(T), a set of properties P(T), and a set of constraints C(T). The type type defines an operation called Instances that can be applied to any type. It returns a set containing all instances of that type that are currently in the database. The functions legal-operations and legal-properties return the set of all operations that can be applied to its argument and the set of all legal properties that are defined for it argument repsectively. The following condition is always met by a type definition T:

$$
\begin{array}{l}
(x \in \text{Instances } (T)) \\
(o \in O(T)) \; (p \in P(T)) \; (c \in C(T)) \\
\quad o \in \text{legal-operations } (x) \text{ and} \\
\quad p \in \text{legal-properties } (x) \text{ and} \\
\quad c(x)
\end{array}
$$

Properties, operations, and constraints are all objects in their own right. That is to say, there exists a type called Property that defines the general behavior of a property, a type called Operation that defines the general behavior of an operation, and a type called Constraint that defines the general operational characteristics of a constraint. A type is also an object and as such is an instance of a type called Type that describes the behavior of all types. Since type Type is a type, it is an instance of itself.

An operation is an active element in our database. It is essentially a piece of code. All operations support the operation *invoke* which takes an operation and a set of arguments as parameters and invokes the given operation on the set of parameters. Since operations are code (written in a language like C), in order to use them, the system must link any symbols used in this code to the library routines and other code objects to which they refer. This requires a dynamic linking loader.

A property can be conceptualized as a piece of string which binds together two objects. For example, Cars can have owners. Therefore, we define a property type that can link a Car to a Person. An instance of this property type will link an individual car (e.g., my car) to an individual owner (e.g., me). All properties support the operation *get-value* which, given a property, returns the value of that property. Get-value (my-car) returns me.

† This research was supported in part by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786.

CH2345-7/86/0000/0041$01.00 © 1986 IEEE

Constraints are predicates. They are defined on types and take an instance of the type as an argument. All instances of a type must always satisfy that type's constraints.

A type in our model is an abstract data type [LSAS] in the sense that it defines a private representation that is not visible to any code other than the operations that are defined on that type. The representation (rep) is an object of some other abstract type. As we shall see later, the rep is not even available to other subtypes. The type is, therefore, the unit of modularity.

Inheritance is provided in our system by a special property that is defined to hold between instances of the type Type. This property is called is-a and if there is an is-a property between types A and B such that B is-a A, then all instances of B are also instances of A. All behavior (operations, properties, and constraints) that is defined on A is also defined on B. We say in this case that B is a subtype of A and A is a supertype of B. It is possible for a type to have several supertypes from which it inherits behavior. We call this phenomenon multiple inheritance. Circularity in the type lattice is not allowed. It is important in the example that is presented later to remember that is-a is just another property.

Object-oriented databases [CM, DGM, MS, MSOP, Zd1, ZW] differ from their programming language counterparts in the following fundamental ways.

1. persistence
2. unique naming
3. sharing
4. transactions

Objects that are created by a process persist beyond the lifetime of that process. The database system assigns all objects a unique identifier that is guaranteed to remain unique even across multiple processes. Any number of applications can share the objects that reside in this persistent memory space. In the process of using these objects a given process can define the boundaries of transactions that are guaranteed to be atomic and resilient and that preserve some set of correctness criteria. If we adopt a strong world view, these criteria might encompass serializability, however, they may also be defined to be weaker [SZR].

2. The Role of Properties

One of the strengths of the object-oriented approach is the ability to create new types by specializing the behavior of old types. In our object-oriented database [ZW1, ZW2], we treat properties as objects, thus providing the ability to specialize the behavior of properties by defining new subtypes of the type Property. This paper describes this technique and discusses a few interesting property subtypes as an illustration of this approach. This example further illustrates how we might fix what we consider to be a common defect that occurs in some object-oriented systems. This defect arises from the confusion between the inheritance of specification and the inheritance of implementation. The example illustrates how these concepts might be separated.

It is, of course, possible to get the behavior of simple properties by defining a set of operations on the type of the object to which the property applies. These operations would get and set the value of the property. For example, suppose that the type Person has a property mother-of that relates a person to his or her mother. We could achieve same behavior by defining a get-mother-of and a set-mother-of operation on the type Person.

In our view, properties are objects which implies that as such they can partake in all behavior that objects in general have. This means that it is possible to define properties on properties since all object types have the ability to define properties. This allows us to model some situations more precisely. Suppose that John loves Mary. This could be modeled as two instances of type person with a loves property between them. If we wanted to assert that the intensity of this relationship is 8, placing the intensity on either John or Mary would be somewhat inaccurate. The intensity does not assert anything about John or Mary by themselves, but rather, it describes the loves property.

More importantly, by making properties be objects, we can refine their behavior by defining subtypes for them. Although there seem to be operations that do not apply to properties, yet are influenced by their definitions, a closer inspection reveals that these operations are really more complex composites that make use of the operations of the property types. For example, all objects are defined to have a get-property-value operation. This operation takes an object x and a property name p as arguments and returns the value of p if the type of x is defined to support properties of the type of p. This is an operation on x that appears not to use the property object. In reality, the get-property-value operation is implemented as follows:

```
get-property-value (x, p) =
        get-value (get-property (x, p))
```

A useful modeling tool is that of a derived property. A derived property is one whose value is dependent on other information that is contained in the state of the object. For example, employees might have a Salary property which is stored with each employee. They might also have a FICA-payment property that is not stored at all. Instead its value is computed as a function of the employee's salary.

It is possible to add the notion of a derived property to our model by noticing that derived properties are just like regular properties except that it is not possible to set their values. The derivation function is embodied in the get-value operation for the property. We define type property to have a get-value operation, but no set-value operation. We define a subtype of the type Property called Setable-property that adds the set-value property. We then add another subtype of Property that adds a property to Derived-properties called the derivation-expression. The get-value operation for derived properties evaluates this expression.

3. Some Important Distinctions

We begin our extended example by pointing out again that in our system a type exports an interface that is the only way that other code can manipulate objects of this type. This is also true for subtypes and supertypes of a given type. For example, suppose that the type Toyota is a subtype of the type Car. The create operation on the type Toyota has a side effect of invoking the create operation on the type Car and all types between Car and type Entity (the root of the hierarchy). Each of these create operations may allocate additional storage to store the state of its instance. Type Car has a move operation defined on it. In defining a drive operation for Toyota, if we want to change the position of the car, we must call the move operation defined at the Car level. The code for Toyota cannot manipulate the state of the Car variables directly.

This choice was made to incorporate in our system the benefits of data abstraction to the fullest possible extent. It now becomes possible to change the code for any type without having to modify the code for any other type (including subtypes and supertypes) as long as the interface remains unchanged. If the code for type Car changes but its operations and properties remain fixed, the operations on type Toyota will still work since they can only rely on the interface to Car. Of course, if the code for Car changes, any instances of Car that are already in the database might be affected. The problem of coping with this problem has been discussed in [SZ, Zd2].

In some systems such as Smalltalk [G,GR], it is possible for a subtype to directly access the instance variables of its supertypes. By saying that B is-a A, type B inherits the specification and the implementation from type A. B acquires all the operations defined on A as well as all of the storage level representation that is used to implement type A.

3.1. Behaves-like

We recognize that there are cases in which our strong view of data abstraction may sometimes interfere with what the programmer really needs to accomplish. For these somewhat rare cases we make a few distinctions about the is-a property. These distinctions will attempt to separate the inheritance of behavior from the inheritance of representation.

Our first example of this is a subtype of type Property called Behaves-like. The is-a property that we have already described is a subtype of Behaves-like. Behaves-like makes a guarantee about the specification of the two types that it relates. If B behaves-like A, B must have at least the behavior of A. B may add additional behavior (properties, operations, and constraints), but all of the behavior of A must be supported on B.

Unlike is-a, behaves-like has no side effect of creating instances of the higher-level types. This extra behavior is added in the definition of the is-a property. As a result, when B behaves-like A, no additional storage is allocated for the supertype A if an instance of B is created.

For example, this might be useful if we wanted to define two types Stack and Small-stack. Stacks in general might be represented by a list since that is most flexible for things that can grow without bound, while Small-stacks might be represented by an array since we can require that a small-stack not grow beyond a certain limit. We would then specify:

Small-stack behaves-like Stack

A create operation on the type Small-stack would have the effect of allocating space for the array and not allocating any space for the list that is the representation at the Stack level. Small stack must reimplement all of the behavior of the Stack. That's what it means for it to behave like a Stack. The system does not check this requirement. We assume that the type definer is a good citizen.

It is worth pointing out that the same effect could be achieved by rearranging the type lattice such that a third type called Generic-stack becomes the supertype of both Stack and Small-stack. Generic stack would have a null representation and be non-instantiable. That is to say that there is no create operation defined on the Generic-stack type. Generic stack could specify the standard stack operations (i.e., push, pop, empty) and the two subtypes could redefine these operations to work with their respective storage structures.

Although this change works, suppose that a definition for Stack already existed. Further suppose that the definition of Small-stack is added at some point' after many instances of Stack have already been created. This is common since we are concerned with a database system with persistent objects. Once a type has been created, we can assume that there will be many instances of it in our persistent store. Changing the type hierarchy, is therefore a very difficult and error-prone activity since the changed type structure might not fully suport the old instances. Although work is being done to ease some of these difficulties [SZ, Zd2], we would like to minimize the number of times that type changes occur. The behaves-like property allows us to retain the old structure, while achieving the behavior that we want.

3.2. Subsumes

There is another distinction that we often want to make. This is exemplified by the is-a relationship in Smalltalk. Here we would like a subtype to have access to the representation of its supertypes. We define another subtype of behaves-like called subsumes that accomplishes this.

Subsumes also guarantees that a subtype have at least the specifications of its supertypes, but it adds the ability for the subtype to access any state that is available in the supertype instance. One way of thinking about the subsumes property is that if B subsumes A, then A exports it get-rep operation to the B type module. The get-rep operation takes an object of an abstract type and returns an object of the concrete type (i.e., representation type). It is similar to the CLU [LABMSSS] down operation.

Although this variation of the behaves-like is potentially dangerous, we will give an example of where this kind of capability is necessary. Suppose that we have a type called Set. The representation of the Set type is an array. The set type has no operations that can observe or exploit the fact that an array is an intrinsically ordered type. Further, suppose that we wish to define a subtype of Set that is called Ordered-set. If we used strict data abstraction, we would have to store something like ordered pairs in the unordered set. The first element of the pair might be the set element and the second element of the pair might be its position or index. Because of information hiding, we have been forced to reinvent the ordering that is already available in the array data type. This is very inefficient.

Instead, by saying that Ordered-set subsumes Set, we can allow the code for Ordered-set to access the array directly. Now this code has the ability to exploit the natural ordering for the array. Of course the code for Ordered-set must maintain any rep invariant that is specified in the Set module. The Ordered-set module can add additional state of its own if this is deemed necessary. A create on the type Ordered-set would then allocate storage for both the Set and the ordered-set. The difference here is in what is accessible.

We have distinguished three different types of behavior related to the notion of is-a. We can define these property types in terms of each other as shown in Figure 1. Each of the arrows in this picture are behaves-like properties.

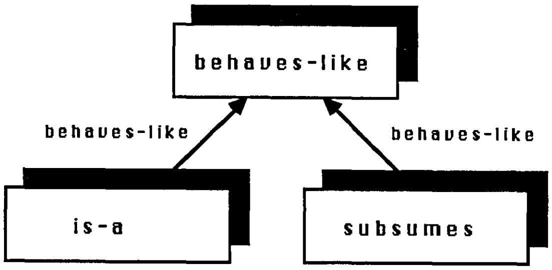


Figure 1: Distinguished versions of is-a

4. Operation subtyping

Another problem with many object-oriented systems is that the notion of operation refinement is not based on any semantic properties of the operations involved. In a language like Smalltalk, one may define an operation Op on a type B that has the same name as an operation Op that has been previously defined on type A, a supertype of B. If x is an instance of B, Op(x) will invoke the definition of Op that is attached to the subtype. This definition will block the definition that is provided by the supertype. Here, the paradigm is operation replacement, not operation refinement. There is no requirement that the two operations named Op bear any semantic relationship to each other. The only semantic tie is that they share the same name.

The problem with this undisciplined use of names is that if we insist that a type hierachy ought to induce a subset relationship among the sets of all instances of the types, we are left with a situation in which some instances of a type may have wildly different behavior from other instances of the same type. If Orange is-a Food-stuff, both types might define a squeeze operation. Squeezing a food stuff is defined to return a firmness coefficient that is useful for determining how fresh that item is, while squeezing an orange might have a side-effect which is to produce a refreshing citrus drink. Although all Food-stuffs support squeezing, some support it with very different results than others.

We prefer to use a somewhat different technique for operation (and property) refinement. Our technique is based on the use of operations and properties that are subtypes of each other as refinements. Therefore, our solution to this problem is another example of why it is useful to treat properties as objects having a type.

We will allow an operation Op2 on a subtype B to refine an operation Opl on a supertype if and only if Op2 behaves-like Opl. Notice here that there is no need to define the two operations as having the same name. In fact, if an operation Op on B is defined with the same name as an existing operation Op on A and the two operation types are not related by a behaves-like property, the system flags this as an error.

In order for the above definitions to make sense, we must explore the meaning of operation and property subtypes a little more closely. What does it mean for an operation type to be a subtype of another operation type? What conditions must hold?

Let us suppose that the type Car has a paint operation defined on it. Paint takes two arguments: a car and a color. Suppose that there is a subtype of car called Model-T that defines an operation called Mpaint. Mpaint takes a Model-T and the color black as arguments and paints the model-T black. We can now ask what the relationship between Paint and Mpaint is?

Cardelli [Ca] holds that, in general, an instance of a subtype should be usable anywhere an instance of the supertype is usable. By this definition, we would say that Paint is a subtype of Mpaint to see this consider the following piece of code:

```
procedure decorate-fleet (p:paint, c:car)
          . . .
          p (c, "black");
           end;

mt: Model-T:
decorate-fleet (Mpaint, mt);
```

Since Mpaint will work very well as the value of p in the decorate-fleet procedure with a Model-T and "black" as arguments, Paint will work just as well in the same context. This leads us to say that Paint will work wherever Mpaint works. Paint is, therefore, a subtype of Mpaint. This is backwards from what intuition would tell us.

Instead we will adopt a point of view in which Mpaint is a subtype of Paint. This is the case because all of the constraints on Paint are met by Mpaint but the opposite is not true. These constraints include the pre-conditions on the argument list and the post-conditions on the return value. In this way we can say that the operation subtype inherits all pre-conditions, post-conditions, and exceptions from the operation supertype.

We can now give a more precise definition of operation refinement. Assume that B behaves-like A. We will say that an operation Op2 on B refines an operation Opl on A if and only if Op2 behaves-like Opl. B inherits all operations O defined on A such that O is not refined by an operation defined on B.

A type is represented as a 5-tuple (N, O, P, C, S) where N is its name, O is a set of operations, P is a set of properties, C is a set of constraints, and S is a set of supertypes. If $T_1 = (N_1, O_1, P_1, C_1, S_1)$ and $T_2 = (N_2, O_2, P_2, C_2, S_2)$, then

$(op \in O_1)$ and $\neg (op \in O_2)$
iff ($op' \in O_2$) such that (op' behaves-like op)

In order for an operation type to be a subtype of another operation type, it too must obey the rule that it have at least the behavior of its supertype. That is, the subtype must inherit all operations, properties, and constraints (pre-conditions and post-conditions) from the supertype.

5. Operation Refinement

We would like to define the behavior of new subtypes such that changes in the semantics of operations (and properties) will not have serious effects on old programs. It would be desirable to have a method that would allow old programs to function properly in the face of a changing type lattice.

There is a tradeoff between the amount of information that we can rely on at compile-time and the ability for programs to adapt at compile time. The more assumptions that we build into compiled code, the more brittle that code will be. In order to allow programs to adapt at compile time, we are forced to defer certain decisions until runtime. This often introduces additional overhead because of the additional runtime checking. Often this checking can be minimized if our compiler is intelligent enough to introduce it only where necessary.

We introduce another subtype of the behaves-like property type called refines. This property is used to relate operation types. It is like behaves-like for operations except that it introduces an additional piece of functionality. If B behaves-like A and an operation Op2 on B refines an operation Opl on A (i.e., Op2 refines Opl), then an invocation of Opl with an instance of B as an argument will cause Op2 to be invoked if all other preconditions match. The preconditions include the decalred types of the arguments to Op2. Opl may only be refined once on a given subtype of A.

An example of this is given below in Figure 2. Here Porsche behaves-like Car. Wash and Wash-gently are defined on Car and Porsche respectively. Wash-gently refines Wash. Suppose that we had the following code:

```
Procedure spring-cleaning (c:car)
     . . .
     wash (c, "ivory")
     . . .
     end;
p: porsche;
spring-cleaning (p);
```

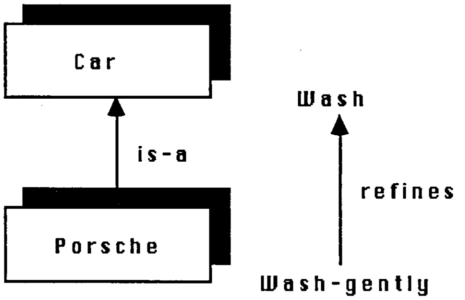


Figure 2: Example of Operation Refinement

Spring-cleaning is called with a porsche which gets passed to the procedure wash along with a second argument of ivory. Since both arguments match the preconditions for wash-gently, a refinement of wash, the wash-gently code is used instead. This is the behavior that you might want in the best of all worlds since you have met all the requirements and would be very dismayed if your porsche were scratched from the use of conventional brushes.

Notice that if in the previous example we had not used ivory soap or if the wash-gently operation had not been defined to be a refinment (perhaps it was only defined to behave-like wash), none of this dispatching would have occurred. It is available only through the use of the refines property and only in cases where the environment is right.

6. Property refinement

All of the above discussion about operation subtyping is applicable to properties. Rather than going through all of that detail again for properties, we will give an example of how property refinement would work.

Suppose that we have two types Person and Porsche-owner with Porsche-owner as a subtype of Person. Type person defines an Age property with a constraint that ages must be integers between 0 and 110. Porsche-owner refines this age property by tightening the constraint to be integers between 35 and 45.

Consider the following piece of code:

```
p: Person;
po: Porsche-owner;

p := po;
p.age := 60;
```

Although the last line is legal, since it is possible to set the age of some people to 60, in the example, p will be assigned a Porsche-owner. The interpretation of p.age will get the age property that is defined on Porsche-owner not the one defined on Person. This property is constrained such that 60 is an illegal value and the assignment will not succeed.

Property refinement is accomplished by refining the get-value and the set-value operations on the property subtype. In the example, the set-value operation for age of Porsche-owner refines the set-value operation for age of Person such that set-value on age of Porsche-owner raises an exception if it is given a value that is not between 35 and 45.

7. Summary

We have presented some examples of the ways in which distinctions ought to be made in the treatment of the is-a hierarchy. We have shown that by treating properties as objects, it is possible to incorporate the desired behavior into our system in a clean and homogeneous manner. We have also assumed that is-a is not a special system-defined resource, but rather that it is simply another property type that relates objects of type type.

It should also be emphasized that this work has been done in the context of an object-oriented database system. In this paper, we have concentrated on some of the linguistic characteristics of the data definition and manipulation languages.

One of the strengths of the object-oriented paradigm is the uniformity with which it treats information. Here, we have extended the uniformity argument by stating that properties or relationships should be treated as objects as well as the more conventional examples. Other systems have made a similar observation, most notably the entity-relationship model [Ch] and several knowledge representation systems [Brl, Br2]. In these systems as in ours relationships or links are denotable. The principle difference here is that we have added this facility to a system that has a very strong notion of type. We have incorporated this view into a paradigm in which a type supports the notion of data abstraction and information hiding. It gives us a modular way to modify the behavior of a property. We do this by using the inheritance mechanisms that are available on the subtree of the type hierarchy that is rooted at the type Property. This approach increases the expressive power of our data modeling language, although it does not provide additional computational facility.

8. References

[ADP] J.M. Smith, S. Fox, and T. Landers, "ADAPLEX: Rational and Reference Manual", second edition, Computer Corporation of America, Cambridge, Mass., 1983.

[Brl] R.J. Brachman, "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks", IEEE Computer, October, 1983, pp 30-36.

[Br2] R.J. Brachman, "I Lied about the Trees Or, Defaults and Definitions in Knowledge Representation", The AI Magazine, Fall, 1985.

[Ca] L. Cardelli, "The Semantics of Multiple Inheritance", in Semantics of Data Types, Lecture Notes in Computer Science 173, Springer-Verlag 1984, to appear in Information and Control.

[Ch] P.P.S. Chen, "The Entity-Relationship Model: Towards a Unified View of Data", ACM TODS 1, 1, March 1976.

[CM] G. Copeland and D. Maier, "Making Smalltalk a Database System", Proceedings of the ACM SIGMOD, Boston, Mass., June, 1984.

[Co] E.F. Codd, "Extending the Database Relational Model to Capture More Meaning". ACM Transactions on Database Systems 4, 4 (December 1979), 397-434.

[DGL] K. Dittrich, W. Gotthard, P.C. Lockemann, "DAMOKLES - A Database System for Software Engineering Environments", Proceedings of the IFIP 2.4 Workshop on Advanced Programming Environments, Trondheim, Norway, June, 1986.

[GAL] A. Albano, L. Cardelli, and R. Orsini, "Galileo: A Strongly Typed Interactive Conceptual Language", Technical Report 83-11271-2, Bell Laboratories, Murray Hill, New Jersey, July, 1983.

[G] A. Goldberg. Introducing the Smalltalk-80 System. *Byte* (August 1981), 14-26.

[GR] A. Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[HM] M. Hammer, D. McLeod, "Database Description with SDM: A Semantic Database Model", ACM TODS 6, 3, September 1981, 351-387.

[LABMSSS] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, A. Snyder, CLU Reference Manual, Springer-Verlag, New York, 1981.

[LSAS] B Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU", Communications of the ACM, Vol 20, No. 8, August, 1977.

[MBW] J. Mylopoulos, P.A. Bernstein, H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications", ACM Transactions on Database Systems, Vol 5, No. 2, June, 1980, pages 185-207.

[MS] D. Maier, J. Stein, "Indexing in an Object-Oriented DBMS", Technical Report CS/E-86-006, Oregon Graduate Center, Beaverton, OR, May, 1986.

[MSOP] D. Maier, J. Stein, A. Otis, A. Purdy, "Development of an Object-Oriented DBMS", Technical Report CS/E-86-005, Oregon Graduate Center, Beaverton, OR, April, 1986.

[S] D.W. Shipman, "The Functional Data Model and the Data Language DAPLEX", ACM TODS 6, 1 (1981), 140-173.

[Sc] J.W. Schmidt, "Type Concepts for Database Definition", in Schneiderman, B. (editor), Databases: Improving Usability and Responsiveness, Academic Press, 1978.

[SS] J.M. Smith, D.C.P. Smith, "Database Abstractions: Aggregation", CACM 20, 6 (1977).

[SZ] A.H. Skarra and S.B. Zdonik,"The Management of Changing Types in an Object-oriented Database", Proceedings of The ACM Conference on Object-oriented Programming Systems, Languages, and Applications, Portland, OR, September, 1986.

[SZR] A.H. Skarra, S.B. Zdonik, and S.P. Reiss, "An Object-Server for an Object-Oriented Database System", IEEE International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, September, 1986.

[Zd1] S.B. Zdonik, "Object Mangement System Concepts", Proceedings of the Second ACM-SIGOA Conference on Office Information Systems, Toronto, Canada, June, 1984.

[Zd2] S.B. Zdonik, "Maintaining Consistency in a Database with Changing Types", ACM SIGPLAN Notices, September, 1986.

[ZW1] S.B. Zdonik and P. Wegner, "A Database Approach to Languages Libraries and Environments", Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large, Harwichport, Massachusetts, June, 1985.

[ZW2] S.B. Zdonik and P. Wegner, "Language and Methodology for Object-Oriented Database Environments", Proceedings of the Nineteenth Annual International Conference on System Sciences, Honolulu, Hawaii, January, 1986.

A Systolic Parsing Algorithm for A Visual Programming Language

A. W. Bojanczyk and T. D. Kimura

Department of Computer Science
Washington University
St. Louis, MO 63130

ABSTRACT

In this paper we consider a problem of parsing a two dimensional visual programming language Shaw and Tell on a two dimensional array of processors.

A program in Show and Tell is a bit mapped two dimensional pattern satisfying a certain set of grammatical rules. The pattern consists of a partially ordered set of rectilinear boxes and arrows distributed over the space of nxn pixel area. The corresponding directed graph, the box graph, where boxes are nodes and arrows are directed edges, may not have a cycle in a Show and Tell program. The cycle detection is the most computationally intensive stage of the parsing section of a Show and Tell program.

We propose to exploit the concept of systolic array in parsing of Show and Tell programming language. A given bit pattern is mapped onto nxn array of mesh connected processors with one pixel assigned to one processing element. We show an algorithm for cycle detection which runs in time proportional to the size of a box graph. The complexity of any individual processor is independent on n, the parameter describing the size of the array.

1. Introduction

Thanks to the recent advancement of VLSI technology, high resolution graphics capabilities have become a dominant computer interfacing mechanism for end users. Efficient man-machine communication can be achieved through a graphic interface because of its high bandwidth. Visual programming[1] is a new concept in software engineering that takes advantage of such high bandwidth for designing better software environments. In a visual programming language, a two-dimensional pattern (or an icon) is used to represent various programming concepts, such as iteration, concurrency, recursion, and so forth. One key problem in visual programming language design is to find a formal syntax of such a two-dimensional language and a parsing algorithm based on the formalism.

A systolic array[2] is a collection of identical processing elements (PE, a CPU with local memory), mesh-connected in a two dimensional form. Each PE communicates with its neighbors by passing messages through a thin-wire. There is no shared memory among the PE's. The concept of systolic array was proposed to take advantage of the reduced hardware cost due to the VLSI technology advancement. Many systolic algorithms are designed for numeric, database, and textual applications.

We propose in this paper another application area; a parsing of two dimensional graphic programming language. A program in a such a language is a bit-mapped two-dimensional pattern satisfying a certain set of grammatical rules. The systolic parsing problem is to construct a systolic array such that when a visual program is mapped onto the array with one PE assigned to one pixel, the array can decide whether the pattern satisfies the grammatical rules or not. It is desirable for the array to make the decision in linear time to the size of the pattern. It is also desirable that the complexity of each PE is independent of the size of the array.

Show and Tell™ Language[3] (STL) is a visual programming language designed for novice computer users such as school children and implemented on the Apple® Macintosh™ personal computer. A STL program is entered through mouse clicking and it consists of a partially ordered set of boxes and arrows. Arrows and boxes may not form a cycle in a STL program. The cycle detection is algorithmically the most complex component of the parsing section of the STL system.

As a part of our efforts to find formal specification methods for visual programming languages (syntax and semantics), we are investigating possible systolic parsing algorithms for STL. In this paper we will present a design of a systolic array that can detect a cycle in a 'box graph' in linear time to the size of the graph, where box graph is an abstraction of a STL program capturing features relevant to the detection of cycles.

In the next section we will briefly introduce STL to provide the background for the problem. Section 3 will define the problem in terms of a formal definition of box graph and systolic array. Section 4 and 5 will describe the results, i.e., the design of a finite state machine that can detect a cycle in linear time to the size of the box graph. Section 6 will conclude with possible extensions of the results.

2. Show and Tell Language

Keyboardless programming is one of the main goals of the STL design. A STL program can be created by using a mouse only. A keyboard is needed only for entering texual or numeric data. STL is designed for computer users who are not familiar with keyboarding.

An STL program consists of nested boxes connected by arrows. Loops and cycles are not allowed. A box may be empty or may contain a data value, an icon which is a name of a system or user-defined operation, or another STL program. Arrows allow data to flow from one box to another. An

CH2345-7/86/0000/0048$01.00 © 1986 IEEE

operation in a box will be executed when and only when all incoming values have arrived at the box. Except for this data dependency there is no inherent sequencing mechanism in STL. The semantic model of STL is based on the concept of dataflow. However, since there is no loop in an STL program, once an operation is executed and the result is registered in an empty box, the value will never change. There are no side effects. Thus, STL is a functional parallel programming language.

An empty box can be filled with a data object. Some empty boxes are used for communication with the environment of the program. They are called the *base boxes* of the program and are depicted by a thicker box frame. They correspond to formal parameters of a subroutine in traditional programming languages. The STL interpreter executes a STL program by filling the base boxes with appropriate solution values.

An STL program can be named by a user defined icon. Any Macpaint™ picture can be used as a name. When a box containing a user defined icon is evaluated by the STL interpreter, the named program will be evaluated, after the incoming data values are moved into the base boxes of the called program. An icon in STL is a subroutine name. Recursive definition of a program is also allowed.

An example of STL program is given in Figure 1. The program defines the factorial function recursively. Note that the program has one input and one output parameter. It consists of four boxes one of which contains five boxes. The inconsistent box, which is defined as any box that contains *conflicting information* such as "5 flowing into 0", is shaded by the interpreter. The data flows from the top to the bottom of the graph. The name of the program is given on the upper left corner of the editing window. The leftmost column provides the editing tools for program construction. Any box that overlaps with other boxes or forms a cycle will be rejected by the editing program of the STL system.

The language is implemented on the Apple® Macintosh™ personal computer. Using the editing tools provided by the STL system, a user constructs a program on the editing window through a sequence of mouse clicking and dragging. In order to draw a box, for example, the user drags the mouse from the upper-left corner of the box, causing a mouse-down event, to the lower-right corner, causing a mouse-up event. The editing program in the STL system recognizes these two mouse events and constructs an internal data structure representing the box, provided that the box is acceptable, i.e., the box does not overlap with other boxes nor forms a cycle with the existing arrows.

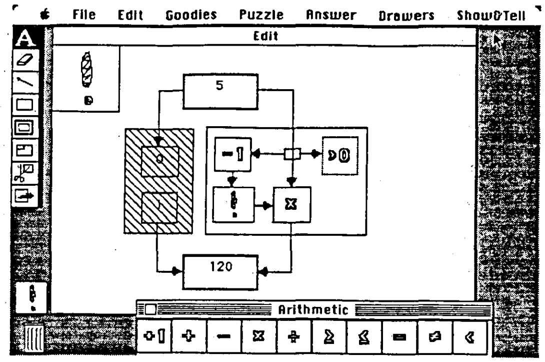


Figure 1: A Recursive STL Puzzle (Factorial Function)

The current STL system does not have capability of parsing a bit mapped image as a STL program. For example, if the factorial program of Figure1 is constructed by MacPaint™, and the resulting MacPaint file is pasted onto the editing window of the STL system, the current STL editor program will interprete the image as a background (comment) image of some other program, and will not be able to parse the bit image as a STL program. Thus, *programming in MacPaint* is not possible with the current STL system.

In this paper we try to solve the problem of finding a parsing algorithm for two-dimensional representation of STL programs. One approach is to find a formal syntax for STL and to construct a parsing algorithm based on the formalism. Our efforts in this direction is reported in a separate paper[4]. Another approach is to construct an algorithm which can accept all and only images that represents a legal STL program. We will take the second approach in this paper.

3. Problem Definition

In this section we will define the problem. Our solution to the problem will be given in the next section. First we will define the concept of *box graph* as a syntactic abstraction of STL program. Then, we will define the concept of *systolic array* as a model of parallel computer architecture. Finally we will define the problem of detecting a cycle in a box graph by an systolic array in linear time to the size of the box graph.

3.1 Box Graph

A box graph is a collection of boxes and arrows geometrically distributed over the space of n x n pixel area. No two boxes overlap with each other. No box contains another box. Every arrow starts from the boundary of one box and ends on the boundary of another box. Arrows may intersect with one another, but they don't branch out. No two arrows intersect with a box at the same location. There must be no loops or cycles in a box graph, i.e., there is no sequence of arrows that connects a box to itself. Boxes and arrows are composed of horizontal and vertical line segments. No diagonal lines exist in a box graph. An example of box graph is given in Figure 2. Note that a box graph is an abstraction of a STL program in hiding the type and the content of each box. Thus, in a box graph no nesting of boxes exist. There is no semantics associated with a box graph.

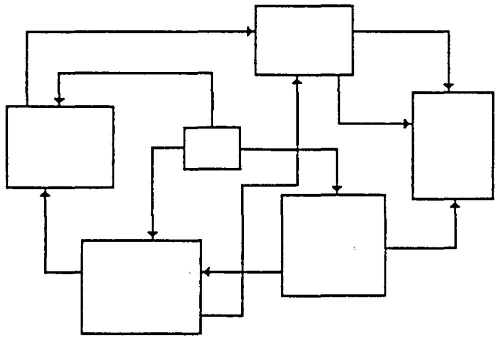


Figure 2: An Example of a Box Graph

The exact definition of a well formed box graph as a bit image will be given in the section 3.3.

3.2 Systolic Array

A systolic array is a collection of n x n processing elements (PE: a CPU with local memory), mesh-connected in a two dimensional form. See Figure 3. Each processor is a simple finite state machine with local memory registers. The memory size of a single processor is independent on n, the parameter describing the size of the array. The processors communicate only with their four nearest neighbors, i.e., north, east, south and west neighbors. Knowing what operations the processors must perform in order to solve a problem, we define a *time unit* to be the maximal time that is necessary for a processor to perform the most time consuming operation together with loading and unloading its registers. Like the memory size, the duration of the time unit is independent on n. A synchronization mechanism allows processors to exchange data at time instants separated by integer multiples of a time unit. We assume that processors are microprogammable. This allows us to change the set of operations performed by each processor when necessary. This approach is similar to one adopted in the PSC project[5].

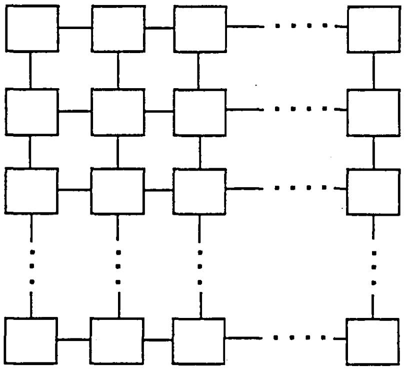


Figure 3: A Systolic Array

3.3 Problem

To design a systolic array of size n x n that can decide whether a given bit pattern is a well formed box graph or not, in $O(n^2)$ steps with constant memory requirement for each PE, where the bit pattern is mapped onto the array with one pixel assigned to one PE.

A well formed box graph is defined as follows:
(1) A box and an arrow consists of line segments. A line segment is one pixel wide. A line segment may be horizontal or vertical, but never diagonal. Two parallel line segments must be separated by at least one pixel. An arrow does not start at a corner of a box.

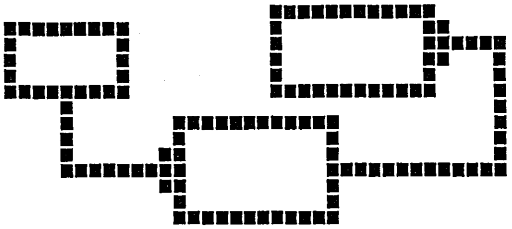


Figure 4: A legal box graph

The following configurations are not allowed.

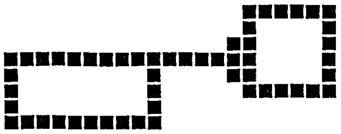


Figure 5: Illegal Box Arrow Combination

(2) A box is a rectangle enclosed by line segments. A box contains nothing and it has at least one empty pixel inside the enclosure.

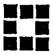


Figure 6: The Smallest Box

(3) An arrow head is represented by extra two pixels as follows:

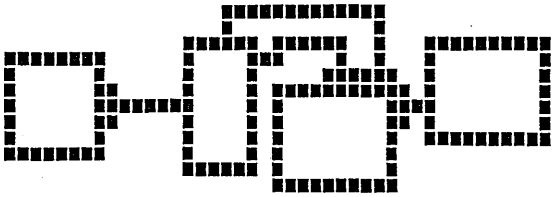


Figure 7: Legal Arrow Heads

The following configurations are not allowed:

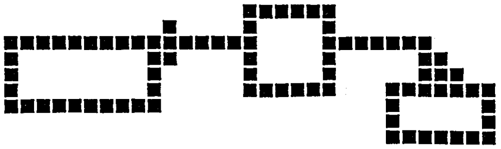


Figure 8: Illegal Arrow Heads

(4) There must be no cycle. The following is an example of cycle:

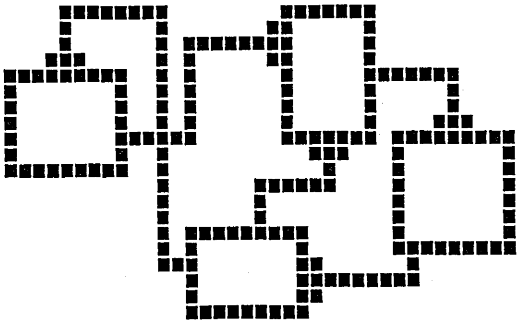


Figure 9: An Example of a Cycle in a Box Graph

3.4 Representation of the Box Graph

A box graph is made up of boxes and arrows (connecting directed lines). Arrows and boxes in turn are made up of pixels, single points of rectilinear grid. Although as individuals the points are indistinguishable, they represent different elements of boxes and arrows, i.e., corners, T-junctions (which are arrow tails), cross junctions, straight line segments and arrow heads. These elements form *basic building blocks* for any box or connecting arrow.

Decision of what a given non-empty pixel represents can be made locally by examining its neighbourhood. The neighbourhood about a pixel consists of all the pixels in a 3x3 window whose center is the given pixel. The pattern of the neighbourhood determines the role played by each non-empty central pixel. Figure 10 illustrates possible representation of basic buildings blocks where 1's denote non-empty pixels, 0's empty or background pixels and X's either of these two. There is also unique pattern code associated with each pattern.

Note that all fifteen patterns are mutually exclusive.

There is a restriction that no arrow can start from a box corner. This restriction excludes the possibility that a T-junction could represent a box corner and a tail of an arrow starting from that corner, or that a cross could represent a box corner and tails of two arrows starting from that corner. See Figure 11 for illustration.

```
 Cross:               Line:
code: 1    2:horizontal   3:vertical

0 1 0           X 0 X          X 1 X
1 1 1           1 1 1          0 1 0
0 1 0           X 0 X          X 1 X

                Corner:
4:NW       5:NE        6:SE        7:SW

0 0 X      X 0 0       0 1 X       X 1 0
0 1 1      1 1 0       1 1 0       0 1 1
X 1 0      0 1 X       X 0 0       0 0 X

              T-Junction:
8:north     9:east     10:south    11:west

0 1 0       X 1 0       X 0 X       0 1 X
1 1 1       0 1 1       1 1 1       1 1 0
X 0 X       X 1 0       0 1 0       0 1 X

              Arrow-head:
12:north    13:east    14:south    15:west

1 1 1       X 1 1       X 0 X       1 1 X
1 1 1       0 1 1       1 1 1       1 1 0
X 0 X       X 1 1       1 1 1       1 1 X
```

Figure 10: Basic Building Blocks

```
1 1 1 1 1 1                   1 1 1 1 1 1
1         1                   1         1
1   box   1                   1   box   1
1         1                   1         1
1 1 1 1 1 1 1 1 1             1 1 1 1 1 1 1 1 1
                                        1
                                        1

not allowed T-junction           not allowed cross
```

Figure 11: Illegal Patterns

The representation of the arrow head involves two additional non-empty pixels which are superfluous, they simply help to identify the arrow-head. This is not the only possible identification. There are many other but all, including the one considered here, have some drawbacks.

4. Recognition of the Box Graph

This section describes the procedure for recognizing a box graph which is initially defined by a set of non-empty pixels distributed over a square nxn grid.

The grid is mapped onto the square array of processors with one grid point assigned to one processor. Each processor has a *pixel register* where non-empty pixel is represented by 1 and background pixel by 0.

The recognition of the box graph proceeds in two stages. In the first stage processors recognize the basic building blocks by comparing the patterns of the neighborhoods with the predefined patterns of the basic building blocks. Note however, that basic building blocks still do not uniquely determine elemments of rectangles or arrows. A corner may be the box corner or turning point of an arrow. A line may be a part of an arrow or a side of a box. Basic building blocks together with specification whether they belong to rectangles or arrows form *box graph building blocks.* In the second stageof box graph recognition, the processors uniquely determine the box graph building blocks.

4.1 Identification of Basic Building Blocks

All fifteen patterns defining the basic building blocks (see Figure 10) are stored in local memories of all processors. Processors which correspond to non-empty pixels have to examine their neighborhood in order to determine what building blocks they represent. Succesive shift operations bring pixels from the neighborhood to the center processor. The pattern of the neighborhood is compared to each of the fifteen basic patterns. If there is a match to any predefined pattern, the pattern code is stored in the pattern code register (LPCR) of the center processor and the center processor is marked as initially recognized. However, for some non-empty pixels the pattern of the neighborhood do not match the pattern of any basic building block. This is a side effect of the representation of the arrow-head. Only the pixels that serve as identifiers of arrow-heads may find themselves in such position.

The resulting ambiguity can be solved in the following way. The processors which were identified as arrow-heads transmit that information to those neighbors which served as markers of the arrow-heads. Because all arrow-heads were found, the markers are not needed any longer and can be erased. The erasing removes ambiguity. Now, by repeating the matching process for all "undecided" processors the identification of basic building blocks is complete, all non-empty pixels know exactly what they represent.

Note that the identification of the basic building blocks takes constant time, independent on n, the array size or even the graph size.

4.2 Identification of the Graph

Once basic building blocks are identified, the array is set to recognize the box graph. Processors have to decide whether they constitute a part of an arrow or a box. In order to make that decision processors must receive enough information possibly coming from the distant regions.

Prior to the computation all processors are in the initial states. In the course of the computation the processors change their states until the final states are assumed. The initial state corresponds to the starting information possessed by a processor which, in case of non-empty pixel, is the pattern code of the basic building block, while for empty pixels the pattern code is zero. The final state means that a processor recognized what element of the box graph it represents. The processor stores the pattern code of the graph building block, which is the code of the basic building block together with the indication whether the building block is a part of an arrow or a box.

For some processors the initial state is also the final state. This is the case with the processors corresponding to the background pixels. Similarly, the processors representing arrow-tails (which are T-junctions), arrow-heads and arrow crossings are in the final state to begin with. All other processors need some additional information before they are able to recognize what element of the box graph they represent.

All decisions (or state changes) are made based on states of some processors, *bounding processors*, lying in the same vertical and horizontal line as the processor making a decision. A bounding processor is such that contains information which may influance states of processors in the same horizontal or vertical line. First of all, any non-empty processor which is in the final state is a bounding processors. Corners, T-junctions, arrow heads and crossings are bounding for a processor which is connected to any of them by a straight line of non-empty processors. Any non-empty processor is bounding for another non-empty processor which is separated from it by a straight line of empty processors. In Figure 12 a possible situation is illustrated. Numbers are codes of basic building blocks. The top, the bottom, the leftmost and the rightmost processors are the bounding processors for the central processor.

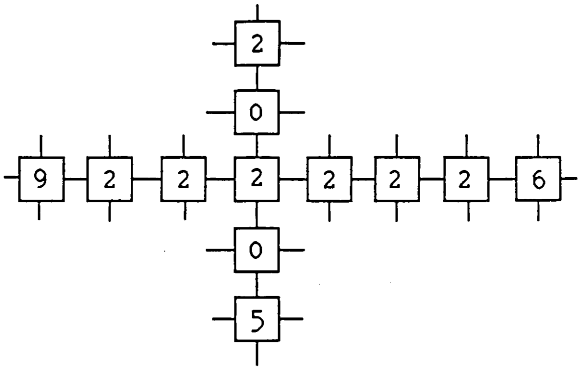


Figure 12: Bounding Processor

The array operates in the synchronous mode. Triggered by the outside control the processors start executing their individual programs. The program to be executed depends on the current state of the processor. All processors pass the data about their current knowledge to the north, east, south and west neighbours. Simultaneously, processors receive similar data from the neighbors. Based on the local data and the data received from the neighbours the processors update their states and also prepare data to be sent to the neighbours in the next cycle.

Each processor has four special registers NR,SR,WR and ER monitoring the states of the bounding processors above, below, to the left and to the right of it. Figure 13 shows some registers which are used for graph building blocks recognition.

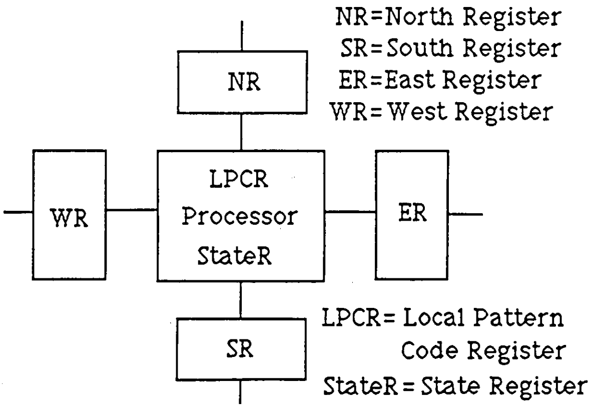


Figure 13: Processor Registers

The StateR register describes the state of the processor. When the processor is not in the final state, the content of StateR is 0. The content is A,B,E,H or T if the processor repretents arrow, box, empty processor, arrow-head or arrow-tail respectively.

The data which is sent by a processor to its neighbors is a concatenation of LPCR and StateR registers of either the processor itself or those of the bounding processors. All incoming messages are evaluated against the information possessed by the processor. A decision is made locally whether the incoming data has higher priority than the information evaluated by the processor. In the case when the incoming data has higher priority it is transmitted further in the same direction it came from. Otherwise, the processor transmits the locally evaluated data.

The data representing empty pixels have the lowest priority. The corresponding processors do not perform any computation but simply shift the incoming data along vertical and horizontal connections. The data representing non-empty pixels which are in the final state have the highest priority. The corresponding processors also do not perform any computation. They ignore the incoming data but keep sending the information what (final) part of the box graph they represent to the immediate neighbors. If both incoming data and local data represent the final states then, as the incoming data is ignored, the local data has precedence. In the case when the incoming data have the same code as the code of the local data, the local data have the higher priority. The data representing line have lower priority than the data representing any other building block. The local data representing any building block other than line have higher priority than any incoming data except when the incoming data represents the final non-empty state and the receiving processor is not in the final state.

4.3 Arrow Identification

If a processor is not in the final state but one of its non- empty neighbors is in the final state, then the final state of the neighbor uniquely determines the final state of the processor. That is if any of the nearest neighbors (north, south, west or east neighbor) represents an arrow then the processor must be a part of that arrow. Similarly, if a neighbor represents a box then the processor is also a part of that box. Recall that arrow-heads, arrow-tails or crossings are in the final state from the very begining of the computation. As a consequence, any non-empty processor which is connected to an arrow-head, arrow-tail or crossing will assume the final state A or B on completion of the first unit of time . Next, one by one, succesive non-empty processors connected to arrow-heads, arrow-tails or crossings by a chain of non-empty processors, will be able to determine their final states.

In particular, as the maximal length of any straight, horizontal or vertical, line is n, then any non-empty processor connected to an arrow-head, arrow-tail or crossing by a straight line of non-empty processors will assume the final state in at most n units of time.

When a processor is a corner connected by a straight (non-empty) line to another corner of opposite orientation then both corners must belong to an arrow. (Two corners connected by a line have the *same orientation* when traversing the line either only right or only left turns are encountered, otherwise the corners have the *opposite orientation*). Similarly, for a processor representing a line, if both end of the straight part of the line are corners of oposite direction then the line must be a part of an arrow. Both cases are easily recognized by one of the processors lying on the line that joins the two corners. The processor will receive the information that the opposite bounding processors are corners of opposite orientation and recognize that it is a part of an arrow. Next the message will spread along the line and eventually will reach the corners.

The recognition of a segment of an arrow between two corners of opposite orientation takes no more than 2n units of time.

The constant two comes from the fact that the information about two corners is combined in one of the processors lying in between and next propagated back to the corners.

The only hard situation is when following a line only corners of the same orientation are met. This can happen with a box or a *spiral*, see Figures 14a and 14b below.

```
                          1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
1                         1 1 1
1
1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1   1                                   1
1   1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1
1   1   1                           1   1
1   1   1             1 1 1 1       1   1
1   1   1 1 1 1 1 1 1 1 1 0 1       1   1
1   1                 1 1 1 1       1   1
1   1                               1   1
1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1
1                                       1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Figure 14a: Spiral

```
1 1 1 1 1 1 1 1 1 1
1                 1
1                 1
1                 1
1                 1
1                 1
1                 1
1 1 1 1 1 1 1 1 1 1
```

Figure 14b: Box.

A box has the property that its interior is "empty". Spiral on the other hand has one or more *coils* inside. This makes all but the innermost coils of a spiral easy to recognize. However, the innermost coil must end in a box so it is easy to recognize anyway. The procedure is the following.

As we mentioned before, any non-empty processor which lies on a line connecting two corners of the same orientation is either a part of a box or an arrow. However, if there is a coil inside then any corner of the inner coil will betray that.

Consider a processor which is the part of the outer coil and which lies on the same vertical or horizontal line as the corner of the inner coil. After at most n units of time (which is the maximal distance in vertical and horizontal direction in the array), the processor will have the information that two of its bounding opposite processors represent corners of the same orientation, i.e., the processor represents either a part of a box or a spiral. However, the processor will also receive the message that there is a corner inside. Hence, the processor must represent a part of a spiral. This is conclusive, the processor assumes the final state. That final state is communicated along vertical and horizontal lines. Passing through all nonempty processors the information eventually reaches the two bounding corners which in turn can now decide that they are parts of an arrow. The procedure is simultaneously executed for all four sides of all outer coils.

Any spiral can be recognized in no more than 4n units of time. All outer coils can be recognized in time at most 2n. This is because at most n units of time are necessary to propagate data from all three corners to the processor which makes the final decision. Additional n units of time are needed to propagate the message back to the corners. The innermost coil has the length at most 4n and its end must be an arrow-head or an arrow tail. As the end of the innermost coil is in the final state from the begining of the computation this fact will propagate along the innermost coil in time equal to the length of the coil which is at most 4n.

4.4 Box Identification

It remains to find a way for recognizing boxes. Without loss of generality, consider processors forming upper and left side of a box. These two sides are bounded by corners of the same orientation with the upper-left corner being the only common point. Note that a box is convex and "empty" inside. Thus all processors lying on the upper and left sides have their *counterparts* on the lower and the right sides, only empty processors separate the opposite sides.

Now consider any north-west corner. This processor initiates search for a box. A signal is sent from the north-west corner to all processors forming upper and left sides (including bounding corners) to check on their counterparts. The signal propagates along both sides until it reaches the corners.

If any of the two corners has the opposite orientation than the north-west corner, that corner immediatelly concludes that the line is a part of an arrow. The corner assumes the final state, terminates the search and propagates the information back to the north-west corner. All processors in between, one by one, assume the final states. Next, the north-west corner assumes the final state and transmits this information to its neighbors. Now, the information spreads along the second side, changing the states of all processors on that side which are not in the final states into the final states.

On the other hand, if the corner has the same orientation that the north-west corner, the search continues. The corner which received the signal from the north-west corner (and has the same orientation as the north-west corner) replies by sending boolean value true back to the north-west corner. This boolean value stops at each processor it passes. Each processor which the boolean value visits holds it until the check on the processor counterpart is finished. If the result of the check indicates that the counterpart may belong to the same box as the processor, i.e., for the north/west side the south/east bounding processor represents a line, the boolean value is passed along the line, towards the north-west corner, unchanged. Negative check, i.e., for the north side the south bounding processor is not a line, indicates that the processor is a part of an arrow. This is the final decision overruling other actions the processor may contemplate to take. The search is aborted. The information is sent to the neighbors and transmited further in succesive units of time. Eventually all processor on both sides starting from the north-west corner are informed.

In case when all checks are positive, the north-west corner will receive the boolean value true. The search along one side is positive. If also the search along the second side is positive this means that the interior of the structure is empty. As only a box has this property, the structure must a box. The north-west corner assumes the final state marking the corner as the box corner. The information is transmited to the neighbors which will propagate it further until all processors forming the box are reached. The recognition of the box is now completed.

The recognition of a box takes no more than 4n units of time. This follows from the fact that the message from the north-west corner must travel to the south-east corner and back to the north-west corner, which takes at most 4n units of time.

The results of this section can be summed up in the following theorem.

Theorem 1: A well defined box-graph given as a colection of basic building blocks can be recognized in at most 4n units of time on an nxn array of processors. The size of the local memory of any processor is independent on n.

5. Cycle Search

Let b be any box in the box graph G. By indeg(b) we denote the number of arrows pointing at the box b and by outdeg(b) the number of arrows originating at the box b.

The algorithm for establishing whether a box graph is acyclic is based on the simple observation that removal of all boxes, together with arrows originating at or pointing to them, for which indeg(b)*outdeg(b)=0 will leave the subraph cyclic if the graph is cyclic. If a graph is acyclic the procedure will render the subgraph empty.

The cycle search is initiated by boxes north-west corners. For every box the computation consits of two local phases, checking phase, which may be repeated number of times for the box, and erasing phase, which when entered is executed only once for that box.

In the erasing phase each north-west corner propagates two control bits, *head* and *tail*, along the box perimeter. The initial values for these bits are zeros. The value of tail is changed to one when at least one arrow-tail is found on the perimeter. Similarly, the value of head is changed to one when at least one arrow-head is found on the perimeter.

After at most 4n units of time (the maximal perimetr length of any box), the north-west corner receives both control bits. The next action depends on the value of the bits. When both are ones this indicates that indeg(b)*outdeg(b)>0, i.e., there is at least one arrow pointing to the box and at least one arrow leaving the box. The north-west corner sets both bits to zero and the checking cycle repeats.

In the case when indeg(b)*outdeg(b)=0 the north-west corner initiates erasing phase. A single control bit *erase* is propagated along the perimeter and further along all arrows leaving or pointing to the box up to the point where another box is met. When a processor receives the control bit 'erase' it marks itself as erased and transmits the 'erase' bit to all non-empty and not erased neighbors. When the 'erase' bit is received by a processor belonging to another box, which must be an arrow-head or arrow-tail, the processor replaces the content of LPCR register, which is the code of an arrow-head or an arrow-tail, by the code of a line having the same direction as the direction of the side the processor represented originally. This operation removes one arrow-head or arrow-tail from the second box, possibly changing the product of indeg and outdeg for that box. The (local) erasing phase terminates at this point, however a new (local) erasing may start at the second box.

It is clear that if the original box-graph is acyclic, the cycle search procedure will terminate leaving all non-empty processors marked as erased. The execution time is proportional to the total number of marked processors. We now prove this assertion. To do that we will use the standard graph terminology. In addition, let card(b) and card((b1,b2)) denote the number of processors forming the box b and the number of processors forming the arrow (b1,b2), respectively. Finally, let card(G) denote the total number of non-empty pixels forming the graph G. We are ready to prove the main result of the paper.

Theorem 2:
If a box-graph is acyclic then the cycle serch procedure terminates in time at most 3*card(G).

Proof: Without loss of generality we can assume that G is connected.

As G is acyclic there is at least one box s, a source box, such that indeg(s)=0 and outdeg(s)>0. Similarly, there is at least one box t, a terminal box, such that outdeg(t)=0 and indeg(t)>0.

Consider all source boxes b(1,1),b(1,2),..,b(1,i_1), and all arrows starting at these boxes. The arrows point at boxes b(2,1),b(2,2),...,b(2,i_2). Choose the source box b for which the sum of card(b) and card((b,c)) of the longest arrow staritng at that box is maximal. Let it be the box b(1,1) and the arrow (b(1,1),b(2,1)). From the description of the cycle search procedure it follows that after

2 * card(b(1,1)) + card((b(1,1),b(2,1)))

units of time all boxes b(1,1),b(1,2),..,b(1,i_1) and all arrows which started at these boxes are erased. (We take card(b(1,1)) twice as there are two traverses along the box perimeter, in the first travers the product of outdeg and indeg is calculated, and in the second, the erasing phase is processed).

Some of the boxes b(2,1),b(2,2),..,b(2,i_2) in the resulting subgraph may now become the source boxes for that subgraph and some may have been partially or fully erased. Again, consider all source boxes b(3,1),b(3,2),..,b(3,i_3) and all arrows originating at these boxes in the current subgraph. Note that the intersection of the sets {b(2,1),b(2,2),..,b(2,i_2)} and {b(3,1),b(3,2),..,b(3,i_3)} may be empty. Nevertheless, in the current subgraph there is at least one source box. Otherwise the subgraph would have been cyclis which is impossible as the graph itself is acyclic. Choose the box b and the arrow (b,c) starting at that box for which sum of card(b) and card((b,c)) is maximal. Let it be the box b(3,1) and the arrow (b(3,1),b(4,1)). After

3 * card(b(3,1)) + card((b(3,1),b(4,1)))

units of time all boxes b(3,1),b(3,2),.., b(3,i_3) and all arrows which started at these boxes are erased. (We count card(b(3,1)) three times as now there may be up to three trverses along the box perimeter. In the first travers the check for the product of outdeg and indeg may still be negative. In the second travers the fact that indeg=0 is discovered, and finally in the third travers the erasing phase is processed).

As there are no cycles in the graph, the procedure must terminate on one of the terminal boxes. This happens after

M=[2*card(b(1,1))+card((b(1,1),b(2,1)))]+...+
[3*card(b(2*m-1,1))+card((b(2*m-1,1),b(2*m,1)))]

units of time. But M<3*card(G) which completes the proof.
QED

In the worst case the procedure takes order of n**2 units of time. The reason for this is that the longest "non-empty" path cannot exceed the overall number of processor, which is exactly n**2.

If there is a cycle in the graph, the procedure will not terminate without the intervention of the outside control. However, we know that after at most 3*n**2 units of time the acyclic graph would have been erased. Thus after that time, if there are still not erased processors in the array, the graph must have cycles.

6. Conclusions

We introduced a concept of a box graph and proposed a systolic algorithm for an nxn array of processors which dedects cycles in a well defined box graph. The algorithm is synchronous and its behavior in the worst case is proportional to n**2. The memory requierement is constant per each processor. When a graph is acyclic the algorithm can recognize that fact in time proportional to the cardinality of the graph. However, it cannot discover a cycle until "full time" n**2 is reached. It is an open question whether it is possible to detect cycle in time, say, proportional to the cardinality of a box graph. This question is related to the another one. We assumed that the array is synchronous. It is not known to the authors whether the cycle search procedure could terminate if the array operated in asynchronous manner.

In our considerations we assumed that the initial box graph does not have flaws and that no arrow starts or ends at box corners. The question is whether these assumption can be relaxed while maintaining the execution time proportional to cardinality of the graph.

7. References

[1] G. Raeder. "A Survey of Current Graphical Programming Techniques," *IEEE Computer*, August 1985, pp 11-25.

[2] H.T. Kung. "Why Systolic Architecture?," *IEEE Trans. Computer* 15(1):37-46.

[3] T. D. Kimura, J. W. Choi, and J. M. Mack. *A Visual Language for Keyboardless Programming*. Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, March 1986.

Show and Tell is a trademark of Computer Services Corporation.

[4] W.D. Gillett and T.D. Kimura. Parsing Two-Dimesional Languages. *Proc. COMPSAC86*, Chicago, October 1986.

[5] A.L. Fisher, H.T. Kung, L.M. Monier, and Y. Dohl. "The Architecture of a Programmable Systolic Chip," *J. VLSI and Computer Systems* 1(2):153-169 (1984).

LEARNING, RESEARCH, AND THE GRAPHICAL REPRESENTATION OF PROGRAMMING

Robert P. Taylor
Nancy Cunniff
Minh Uchiyama

Center for Intelligent Tools in Education
Department of Communication, Computing and Instructional Technology in Education

Teachers College, Columbia University
New York, New York

ABSTRACT

Although there are many programming languages, there are few alternatives in terms of presentation mode. This paper describes the programming language FPL, a graphic representation of classical programming. The premise behind the design of FPL is that some novice learners would benefit more from a visual presentation of conceptual programming structures rather than from the traditional textual representation. FPL's interactive, graphic programming environment, which supports the learner throughout the programming process is discussed. This environment provides a rich research environment for the study of novice programmers and the effect of this graphic programming language. Preliminary research on FPL suggests that this programming environment has a positive effect on the programs written by novices.

INTRODUCTION

Despite the number of languages available, the learner of programming has few alternatives. Though there are many detailed differences encompassed by the range of most programming languages available today, in one pedagogical sense they afford no variety whatever: all rely exclusively on text for representing the program. For those teaching programming this is unfortunate. It means there is no graphic alternative for the student who is more visually than textually oriented.

This paper describes work addressing this problem. It involves FPL, a visually represented programming language, and the interactive FPL environment used by novices for learning computer programming.

The paper proceeds by successively describing:

1. the contextual background;
2. the FPL language;
3. the interactive environment in which FPL is used;
4. FPL-based research on the learning of programming;
5. concluding observations.

CONTEXTUAL BACKGROUND: GRAPHICS AND ALTERNATIVES FOR LEARNING TO PROGRAM

The work reported here assumes that a visual approach to programming is an important alternative for many learners, and discusses some appropriate development work and research based on that assumption. This first section discusses the foundation for such an assumption.

Classical Programming

Classical programming refers to well-structured programming as done in a number of now well-established, structurally similar languages such as Algol, Basic, Cobol, Fortran, Pascal or Pl/1. A casual glance at programs written in these languages makes one thing immediately obvious: all are shaped by their adoption of our natural language. All use words, punctuation, special characters and narrative sequence to represent programs. In particular, all use key words to designate specific programming structures and to indicate a program's flow of control.

Classical programming is widely taught in educational settings. However, it is not clear that the exclusively textual representation characteristic of all the classical languages can meet the needs of all types of learners equally well. Although the sequential, linear nature of language may match the computer's organization (linear, sequential and rule-governed), it does not necessarily match the way many people think about problems and problem solving, either on or off the computer.

Graphics vs. Text

Although little is known about how graphics influence learning, intuition seems to suggest that the use of graphic materials makes learning easier. "It is commonly acknowledged that the human mind is strongly visually oriented and that people acquire information at a significantly higher rate by discovering graphical relationships in complex pictures than by reading text"[1] (p. 12).

Educators have long realized that not all learners have identical learning styles and preferences. Supporting alternative learning styles by providing a variety of methods and materials is an important and worthy educational goal. Teachers and learners of computer programming have not had alternatives available to them, in one sense, at least. The textual focus of all of the commonly available languages severely limits the learner's options. A language that presents programming graphically would provide visually oriented learners with an appropriate alternative vehicle for learning about programming.

CH2345-7/86/0000/0056$01.00 © 1986 IEEE

A Graphical Approach To Programming

FPL graphically represents programming through the set of 11 icons, each of which represents one programming construct. Two icons are textless, the other nine include text, in the form of variables and/or constants. Together, the icons eliminate the need for key or reserved words, so variable names and constants are the only textual elements in an FPL program. Each icon is directly translatable into corresponding text in a traditional classical language. For example, Figure 1 shows the icons and their Pascal translations.

In FPL the spatial arrangement of icons embodies the flow of control so an FPL program is drawn, not listed. Because they are entirely textual, traditional program listings visually suggest sequential action, regardless of the actual nature of the program flow. By contrast, an FPL drawing more immediately indicates the program's actual structure. Refer to Figure 2 for a representative FPL program and its textual Pascal equivalent.

Design Criteria

The design criteria were shaped by the underlying purpose for FPL: to provide novices with a simple, visually oriented alternative for learning classical programming. These criteria can best be summarized by the following goals:

1. To reduce classical programming constructs to a minimum set reflecting the underlying similarity of all languages in this classical group;
2. To graphically represent each such construct with a unique icon, and to represent the logic of programming through these icons and the manner of connecting and relating them;
3. To make each icon easy to draw and remember;
4. To implement the language in an easy to use environment that provides extensive explanatory help on demand and blocks any error with an immediate and precise explanation.

The resulting system provides an alternative programming vehicle for the student whose aptitude favors a visual rather than a purely textual approach to learning and remembering, in a supportive editing and testing environment. It also provides a rich environment for studying the programming process.

Readers who want a more complete description of the FPL language should consult Programming Primer[5], the FPL icon design criteria, should see Taylor's description[6], and those interested in the general visual programming context should see IEEE's Special Issue on Visual Programming[3].

Visual Programming

Early attempts to define an alternative mode for representing logic in programming gave rise to standard flowcharts. They became a widely used, popular vehicle for planning and debugging program logic and algorithms. Despite on-going debate over the usefulness of this particular tool, the value of graphical representation was recognized, and many more attempts have been made to design viable tools for graphically representing programming[2]. One limitation of many of these early visual representations is that they offer the programmer only a limited graphical representation; actual coding must still be done in standard textual form.

Recent efforts to expand the use of visualization in programming systems has given rise to the genre of "visual programming," defined by Grafton and Ichikawa[3] as encompassing three distinct areas of research: "graphic techniques that provide both static and dynamic multidimensional views of software, graphics-based very high-level programming languages, and animation of algorithms and software" (p. 7). The field of visual programming is a relatively new one, and much of the development has focused on systems for graphic display of the program flow and data values during execution and debugging. Such systems make visual the dynamic nature of a program. As Grafton and Ichikawa[3] point out, "...the ability to see data flow and control structures of algorithms and software as they execute will give software engineers and computer scientists an ability to understand and 'feel' the action of software or algorithms" (p. 8). Meyers[4] has carefully reviewed many of these systems and has devised a taxonomy for classification of the wide variety of languages, environments and systems which claim to be "visual" in one sense or another.

Systems employing any of the currently available visual techniques can certainly help both novice and expert programmers, but they often confine the use of graphic representation to a limited subset of the actions involved in the programming process. What is needed is an integrated system that provides a programmer with a graphic tool for program planning, an interactive, graphic, computer-based interface for program writing, and a program execution environment that permits examination of memory contents while seeing data flow and control structures. Such a system seems ideal for novices who are grappling with understanding programming language constructs at the same time as they are trying to understand how the computer actually acts on the written program. An integrated, graphically represented language would enhance understanding and simplify the cycle of program composition, comprehension and debugging. It ought to especially suit those novices who are more visually than textually oriented.

The growing interest in visual programming suggests that some developers, at least, believe graphic tools make it easier to understand the complex action of the computer. However, while intuition and anecdotally recorded observation may have persuaded many people of the viability of graphical systems, empirical research to confirm that viability is still missing.

A firm understanding of the programming process, the behavior of programmers, and the interaction of language, graphics and programming achievement can only follow the development of contrasting alternative environments and well-designed empirical research on their use. As a graphically represented programming language, FPL and its implementation constitute such an environment. The FPL-based research described in the fourth section of this paper is designed to illuminate specific aspects of the programming process and thus contribute to our general understanding of it.

THE FPL LANGUAGE: A VISUAL REPRESENTATION OF LOGIC

FPL (First Programming Language) reflects the structure common to the entire family of classical languages and is designed to facilitate learning that structure. However, though it shares the structure, it does not share the exclusively textual, keyword representation of that structure. Instead, FPL uses iconic representation. The first two figures illustrate the difference. Figure 1 presents the eleven FPL icons and their equivalent in Pascal, a representative classical language. Figure 2 broadens the comparison by presenting parallel versions of the same program.

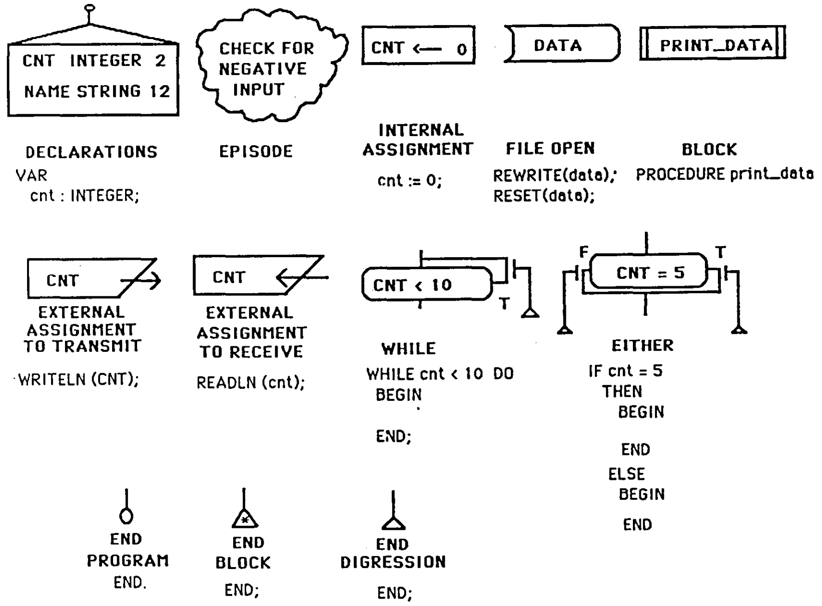
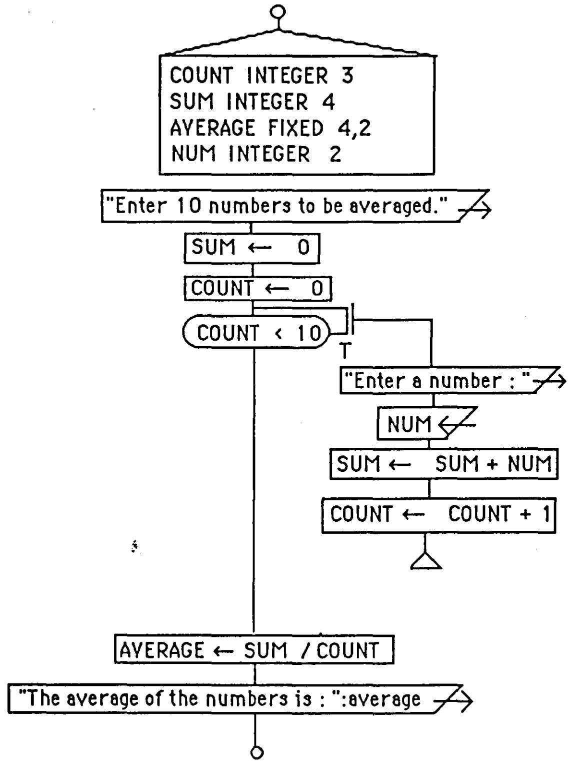


FIGURE 1: FPL Icons
with Pascal translations

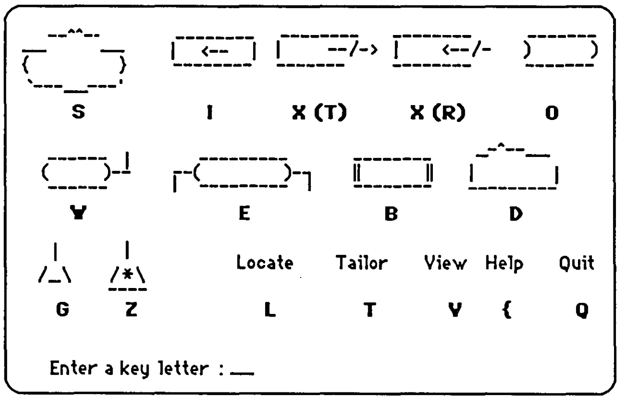


```
PROGRAM test(INPUT,OUTPUT);
 VAR
   count : INTEGER;
   sum : INTEGER;
   average : REAL;
   num : INTEGER;
 BEGIN
  WRITELN('Enter 10 numbers to be averaged.');
  sum := 0;
  count := 0;
  WHILE count < 10 DO
    BEGIN
      WRITELN('Enter a number : ');
      READLN(num);
      sum := sum + num;
      count := count + 1;
    END;
  average := sum / count;
  WRITELN('The average is : ',average:4:2);
END.
```

FIGURE 2: FPL PROGRAM
and PASCAL TRANSLATION

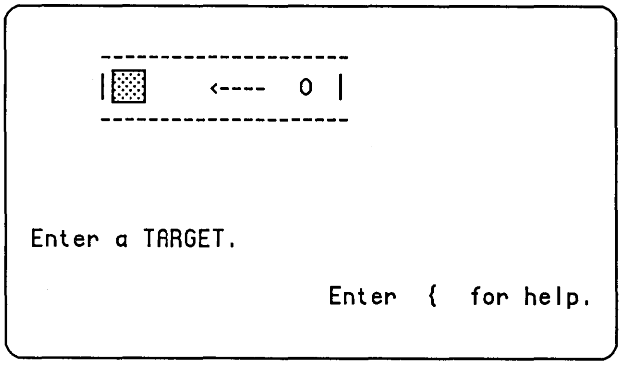
THE FPL ENVIRONMENT: STUDIO CREATION AND LAB TESTING

In one respect, FPL goes beyond the scope of many other attempts at visual programming: the FPL environment supports the user from the initial problem solving and computer representation stage through debugging, refinement and execution. The FPL environment runs on the IBM PC/XT/AT family of microcomputers and has two parts, a Studio and a Lab. In the Studio the program is created and refined. Figures 3, 4, and 5 (discussed further below) illustrate scenes of work in the Studio. In the Lab, the program is executed and its execution monitored, via an animated representation of the program, its I/O, and the dynamic display of the value of user selected variables. Figure 6 (also discussed further below) illustrates with a scene from the Lab. The Studio is discussed first and the Lab, which is still in prototype, second.



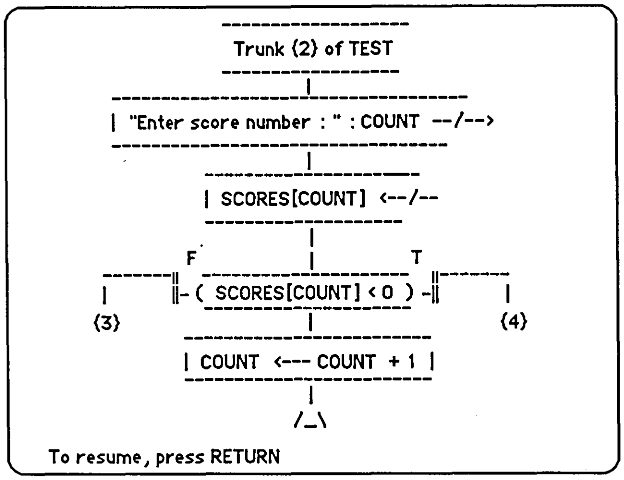
```
  S         I       X (T)      X (R)       O

  W                E          B          D

  G    Z          Locate    Tailor   View  Help   Quit
                    L         T        V     {      Q

Enter a key letter : __
```

FIGURE 3: The main prompt line



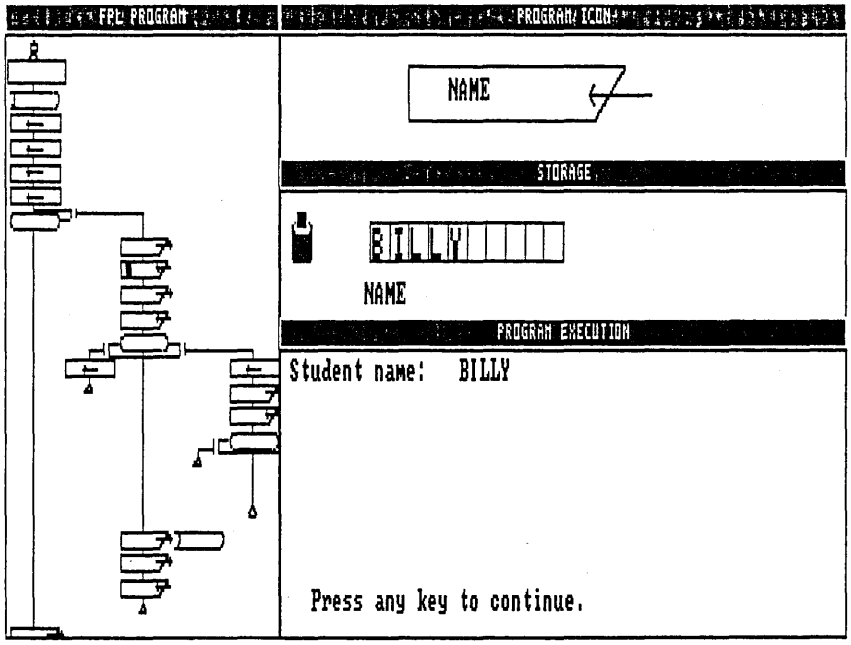
```
          ---------------------
          |▒      <----   0  |
          ---------------------



Enter a TARGET.

                          Enter  {   for help.
```

FIGURE 4: Sample icon entry screen



```
                     -------------------
                     Trunk (2) of TEST
                     -------------------
                             |
           -------------------------------------
           | "Enter score number : " : COUNT --/-->
           -------------------------------------
                             |
                     --------------------
                     | SCORES[COUNT] <--/--
                     --------------------
                             |
                  F          |          T
        --------||  -----------------------  ||--------
        |       ||-( SCORES[COUNT] < 0 ) -||        |
       (3)          -----------------------          (4)
                             |
                  ---------------------------
                  | COUNT <--- COUNT + 1 |
                  ---------------------------
                             |
                            /_\
 To resume, press RETURN
```

FIGURE 5: Display of a single "trunk"



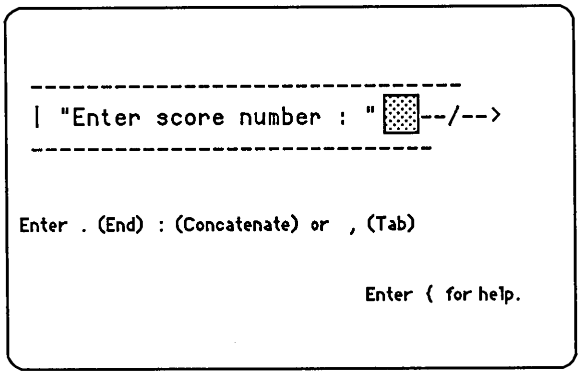
```
Student name:   BILLY


Press any key to continue.
```

FIGURE 6: FPL LAB screen layout

The Graphic Studio: Program Creation and Editing

A student working in FPL is encouraged to sketch a program first on paper, planning how to solve the problem. When ready to enter or edit a version of his/her program, the student invokes the FPL Studio.

From the studio, the user can create or refine his/her program by adding new icons or by deleting, replacing or modifying existing ones. When creating or refining, the user focuses on a single icon, and the displayed context reflects this. Figure 4, for example, illustrates a scene from a user's entry of an "I" or Internal Assignment icon; figure 7, entry of an "X" or External Assignment icon. Though the user can easily move in and out of broader visual contexts (see next section, Program Viewing and Drawing), the immediate one is that of the single icon. In that context, the user is graphically prompted for each component of the entry (again, see figures 4 and 7 for examples), in a set order, based on the icon type. The user can freely intermix work on any icon with access to and browsing within the help system (see section below, System Support Features). The user can exit from work on any icon via this help system, or following system detection of any syntax error in the icon entry.



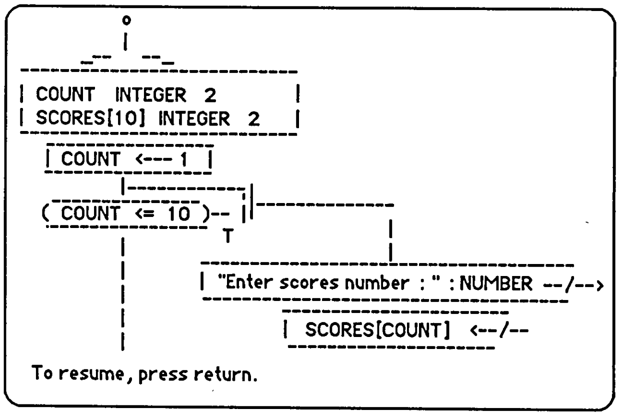
```
-------------------------------
| "Enter score number : " ▒--/-->
------------------------------

Enter . (End) : (Concatenate) or , (Tab)

                              Enter { for help.
```

FIGURE 7: External Assignment icon prompt

While entering icons, the user is always returned to the main entry prompt line (see figure 3) once entry of a specific icon is completed. When refining generally, the user is returned to a broader prompt to determine if he or she wishes to edit further, and if so, where. In deciding where to edit, seeing more context than a single icon is essential. For this, the user must be able to scan all or part of his/her program.

Program Viewing and Drawing. There are two different perspectives from which one can view the program under construction: immediate and global. In the first, the user can see the icons in the trunk currently being worked on. In the second, the user can view any part of the program whatever. Thus, though while working on a specific icon the programmer sees only that one icon, more of the current program can be viewed on demand.

Immediate viewing (Figure 5) shows one vertical line, or trunk, at a time, either the one being worked on, or any other trunk the user chooses. If this doesn't provide sufficient context, once the user has finished entering any logically complete version of his/her program, the system can create a complete drawing of the program to date. At that point, without leaving the Studio, the user can display the program on screen (see Figure 8) or stop and print out a drawing of the program on paper.



```
          o
          |
       _--  --_
--------------------------
| COUNT  INTEGER  2       |
| SCORES[10] INTEGER  2   |
--------------------------
  ---------------
  | COUNT  <--- 1 |
  ---------------
          |-----------|
  ---------------     ||--------------
 ( COUNT  <= 10 )--  ||             |
  ---------------  T                |
          |                         |
          |          ----------------------------------
          |          | "Enter scores number : " : NUMBER --/-->
          |          ---------------------------------
          |                 --------------------
          |                 |  SCORES[COUNT]  <--/--
          |                 -------------------
 To resume, press return.
```

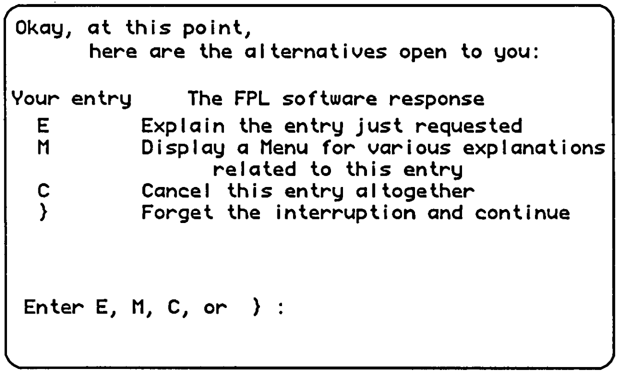
FIGURE 8: On-screen drawing

Program Preservation. When the programmer finishes working on a program, it is saved on disk in an intermediate representation form. At any later time, the programmer can reenter the Studio, access that program, and work on it further.

System Support Features. The FPL Studio offers comprehensive support to the programmer throughout the programming process by means of several special features which are similar in many ways to features employed by other programming language environments (*e.g.*, Macintosh Pascal[7] and Support[8]).

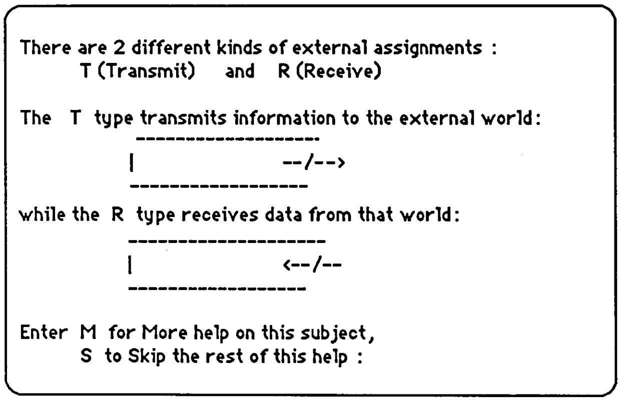
1. Tailoring by the user. To better reflect the learning styles and stages of different learners, the system can be moderately tailored by the user, to control how often he or she will be prompted to accept or cancel an entry. The user can select any of three frequency levels of such verification prompting :
   a. Beginning user : user verification required upon entry of each icon component and upon completion of each entire icon;
   b. Familiar user : user verification required only at the completion of each entire icon;
   c. Competent user : no verification required; assumes user can edit if required.

   The last enables the experienced user to work quickly; the first and second allow less experienced users to proceed more cautiously, at either of two lesser speeds.
2. On-line help. The Studio includes a comprehensive on-line help facility. At any point, the user can get help by means of a character reserved for this purpose. The system's response allows the user to either see an explanation specific to the immediate context, (see figures 9 and 10 for examples) or by going through a hierarchical menu structure, to access help on virtually any aspect of FPL (see figure 11). Following the invocation of the help system, the user may either continue from the point of interruption or cancel the entry or edit attempt which raised the need for help.



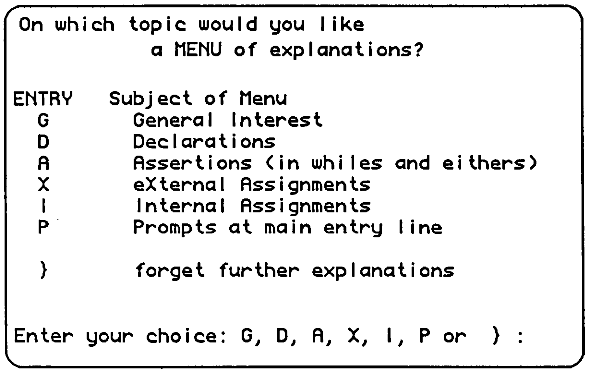
```
Okay, at this point,
     here are the alternatives open to you:

Your entry     The FPL software response
  E        Explain the entry just requested
  M        Display a Menu for various explanations
                related to this entry
  C        Cancel this entry altogether
  )        Forget the interruption and continue



 Enter E, M, C, or  ) :
```

FIGURE 9 : Help System main prompt



```
There are 2 different kinds of external assignments :
      T (Transmit)    and   R (Receive)

The  T  type transmits information to the external world:
              -------------------
              |              --/-->
              ------------------
while the  R  type receives data from that world:
              --------------------
              |              <--/--
              ------------------

Enter  M  for More help on this subject,
      S  to Skip the rest of this help :
```

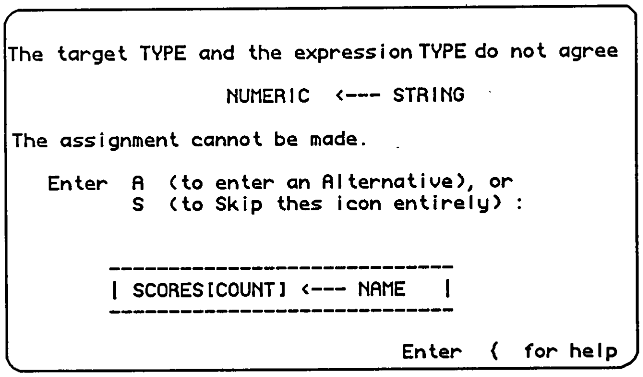
FIGURE 10: Sample help message



```
On which topic would you like
           a MENU of explanations?

ENTRY   Subject of Menu
  G       General Interest
  D       Declarations
  A       Assertions (in whiles and eithers)
  X       eXternal Assignments
  I       Internal Assignments
  P       Prompts at main entry line

  )       forget further explanations


Enter your choice: G, D, A, X, I, P or  ) :
```

FIGURE 11: Help System sub-prompt

3. Context driven error messages. The FPL Studio also includes immediate, comprehensive syntax error detection, and follows the detection of any illegal entry with a message informing the user of the exact error, explaining why it is an error. It then prompts the user to either enter an appropriate choice or to cancel the entry altogether. An example appears in Figure 12. As a result, every completed FPL program is free of syntax errors and can be executed. This comprehensive error screening moves the novice programmer directly to the logical level of debugging without waylaying him/her in a swamp of indecipherable or misleading compiler messages.



```
The target TYPE and the expression TYPE do not agree

                    NUMERIC  <--- STRING

The assignment cannot be made.

   Enter  A  (to enter an Alternative), or
          S  (to Skip thes icon entirely) :


         -----------------------------
         | SCORES[COUNT] <--- NAME    |
         -----------------------------
                                  Enter  (  for help
```

FIGURE 12: Sample error message

<u>The FPL Lab: Executing and Monitoring the Program</u>

Since the FPL Studio always produces executable code, the learner can immediately execute his/her program upon leaving the Studio. However, simply executing it in traditional fashion may do little to help the novice understand the dynamics of any logical shortcomings. Given the aims of FPL and the support it provides for program development, such minimal execution support was inappropriate. Consequently, we began to develop the <u>FPL Lab</u>, as an interactive, animated environment for program execution. It completes FPL as a visual language, rounding out the planning - entry - execution - debugging - refinement cycle in a highly visual manner. The FPL Lab described below is a prototype for the system we are developing.

<u>Screen Design</u>. The FPL Lab divides the screen into four windows. See Figures 6 and 13 for typical screen scenes from the FPL Lab. Each window is dedicated to the presentation of one particular type of information. The windows are :

1. the user's FPL program drawing (in reduced size);
2. the precise icon being executed;
3. the contents of memory for one variable;
4. the normal I/O of the program.

The largest window, "FPL Program," displays a miniature version of the program drawing. (To conserve space, all icons there are reproduced without text, in outline only.) It reveals the path of program execution by moving a lighted "blip" about the miniature, from icon to icon, as each icon is actually executed. Because space is so limited, even in this miniature form most program drawings do not entirely fit in this window. To overcome this limitation, the system shifts the window's contents to show the executing portion of the program.

Since the Lab drawing of the program is textless, as each icon is executed, a full scale version of that icon, including any datanames and constants, is displayed in the "Program Icon" window.

The "Storage" window displays the data value in memory for any program variable the user selects, to help him/her realize how variables work, and to show where and when data values change.

<u>Dynamic Program Representation</u>. By comparing the currently displayed icon, the memory values in the storage window, and the textless program drawing, the user can clearly see where, why, and when values are getting changed. In short, the novice programmer "sees" the dynamics of execution, a prerequisite to developing a real understanding of what a computer program is.

Modes of operation. The FPL Lab has two modes of operation: Step Mode and Quick Mode. In Step Mode (see Figure 12), the program is executed one element (one icon) at a time at a rate under user control. All four windows are active to provide maximum visual feedback about all aspects of the program.

In Quick Mode (see figure 14), only two windows are active: "FPL Program" and "Program Execution". The user can watch the blip traverse the miniature program and can see the normal output and input through the "Program Execution" window. This mode does not display individual icons or storage values. It pauses only for the particular input entries required by the program, and between pauses it moves at a fixed and rapid pace.

RESEARCH: ASSESSING THE HELP FPL AFFORDS NOVICES

FPL was designed originally as a language for novices. We have continued to focus on that group, and have developed the FPL environment so that it can be used as a research laboratory for the study of the learning of programming.

Visual vs Textual Programming: Bug Studies

Two recently completed pilot studies have indicated that FPL may help novice programmers avoid some programming bugs commonly found in beginners' programs[9,10]. Both of these studies compared programs written in FPL with Pascal solutions to the same problem. The results of these studies suggest that FPL's graphic representation may help beginning programmers avoid some common programming pitfalls. Specifically, several types of syntax-related conceptual bugs evident in Pascal programs are eliminated in the FPL solutions. Also, bugs related to updating and initializations were considerably more common in Pascal solutions than in the FPL programs. There were some indications, however, that there are classes of bugs that appear to be language independent, and thus have less to do with the language of implementation than with the programmer's understanding of the flow of control of the program.

The results of these initial investigations were intriguing, and we plan to continue this line of research with an eye toward identification of ways of helping novices to avoid conceptual bugs and to develop methods of effective debugging. We think that development of a tool such as the FPL Lab may be an important step in this process of identification.

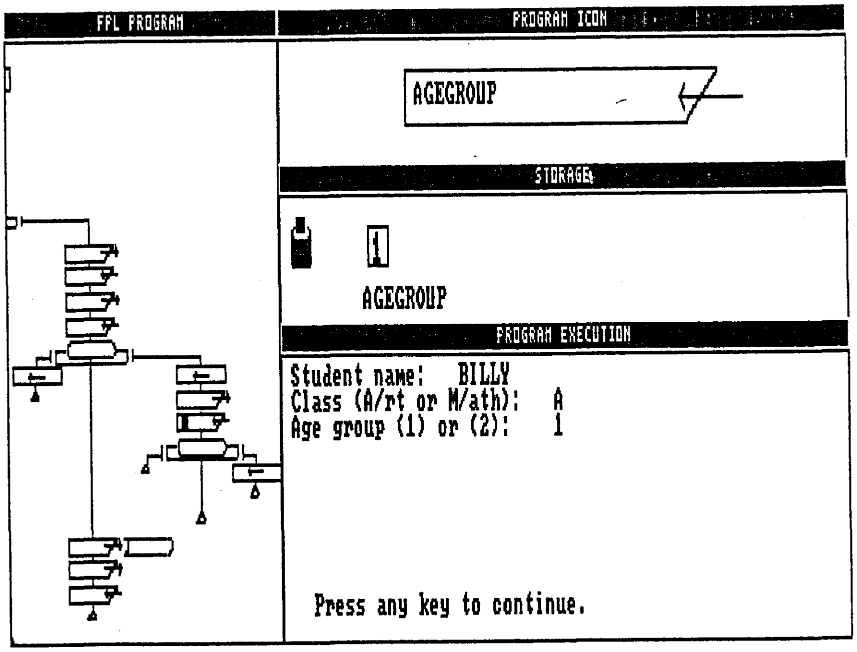
Study of On-Line Help and Errors

We have also recently completed a pilot study of FPL's on-line help system[11]. The study investigated the use of the help system by experienced and inexperienced computer users. The results indicated that although computer-experienced users seemed to use the help facility more than inexperienced users, by the end of several weeks of work using FPL, most users no longer asked for help at all.

For extended analysis of both the help and error sub-systems, the system records, by date and user identification number, all help requests and error messages generated. Analysis of this usage data pinpoints user difficulties with the system, and provide general insight into the process of learning programming. The resulting information is being used to inform instructors about difficulties they are having with introductory programming assignments.

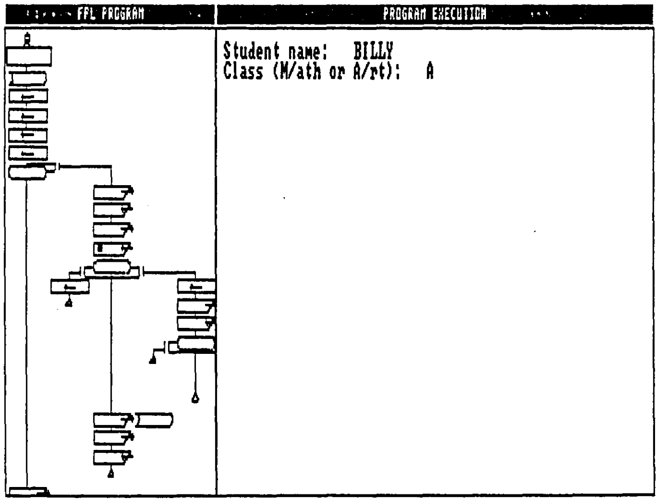
Influence of Graphics on Learning

We continue to be intrigued by the influence of graphics on learning, particularly in relation to programming. To investigate the effect of the graphical interface of FPL, we are beginning to study how learners with different learning aptitudes use it. For example, we are investigating whether learners with high spatial aptitudes understand and learn programming more effectively when using this visual



```
Student name:  BILLY
Class (A/rt or M/ath):  A
Age group (1) or (2):   1



Press any key to continue.
```

FIGURE 13: FPL LAB / STEP MODE



```
Student name:  BILLY
Class (M/ath or A/rt):  A
```

FIGURE 14: FPL LAB / QUICK MODE

approach than when using a traditional textual approach. One study now in design will compare comprehension of FPL programs with comprehension of programs written in a textual language.

We are also in the process of developing a non-graphic version of FPL to be used in a full-scale study of novice programmers. It will preserve the interface design as much as possible, limiting the changes to replacement of the graphic icons by key words. This will facilitate a comparative study of a graphic and non-graphic language without the confounding effects introduced by comparing subjects using two entirely different editing and execution environments. Such a study will address the question of whether a graphic approach to programming is really more appropriate for some learners than is a traditional, textual approach. Through a group of such studies, we hope to prove that alternative ways of teaching and learning programming are important in light of the fact that learners are so different one from another.

SUMMARY

This paper describes the graphically represented programming language FPL, the environment in which the FPL user works, and some preliminary evaluative research. The premise for the development of this system is that for many learners, a language and environment that provides a consistent, graphical representation throughout the programming process is an appealing and effective alternative. We theorize that the consistent visual approach of FPL makes the abstract world of computer programming more concrete, and consequently, helps at least some novices to learn to program more easily and more effectively. Furthermore, because we believe that it can be empirically demonstrated, we have begun a series of studies aimed at testing our theory.

Although our preliminary research does not conclusively prove that FPL improves a novice's learning of programming, it certainly suggests that this alternative, graphic approach to improve the lot of the visually-apt, would-be programmer merits further exploration.

References

[1] Raeder, G. (1985, August). A survey of current graphical programming techniques. IEEE Computer (pp. 11-25.)

[2] Schneyer, R. (1984). A survey of graphic algorithmic representation techniques. Interface. Spring. (pp. 38-48).

[3] Grafton, R. B., Ichikawa, T., eds. (1985, August). Special Issue on Visual programming IEEE Computer. (pp. 6-9.)

[4] Myers, B. A. (1986). Visual Programming, programming by example and program visualization: A taxonomy. Human Factors in Computer Systems: Proceedings of CHI'86 (pp. 59 - 66).

[5] Taylor, R. P. (1982). Programming primer. Reading, MA.: Addison-Wesley.

[6] Taylor, R. P. (1985). FPL: Graphical representation of classical programming. Teachers College, Columbia University: Department of Communication, Computing and Instructional Technology in Education.

[7] Macintosh Pascal. (1984). Lexington, MA: Think Technologies.

[8] Zelkowitz, M. (1986). The SUPPORT Environment for the IBM PC. University of Maryland.

[9] Cunniff, N., Taylor, R. P., and Black, J. B. (1986). Does programming language affect the types of conceptual bugs in novices programs? A comparison of FPL and Pascal. Human Factors in Computer Systems: Proceedings of CHI'86 (pp. 175 - 182).

[10] Cunniff, N., Taylor, R. P., and Taylor, S. J. The effect of programming language on the conceptual bugs in novices' programs: A comparison of FPL and Pascal. (In press.)

[11] Taylor, S., Taylor, R.P. & Cunniff, N. (1985). The use of on-line help in a programming environment. Teachers College, Columbia University: Department of Communication, Computing and Instructional Technology in Education.

COMMAND LANGUAGE SUPPORT FOR APPLICATION PROGRAMS

Christine Genet

Grumman Data Systems Corporation
1000 Woodbury Road
Woodbury, New York 11797

ABSTRACT

Efficient data analysis programs must maximize the productivity of the data analyst/computer combination. To do this, the analysts must be able to use the program easily and efficiently. Thus, the human-computer interface is crucial to the design of an effective data analysis program. We develop an interpreter entitled KEYLAB that allows an application programmer to create programs driven by an English-like command language that is based on the functional decomposition of the applications requirements. We also develop a code generator named Keystone that creates the code necessary for the human-computer interface. The combination of these two programs allows a programmer to develop an application program with minimum effort and maximum uniform connection to functional requirements.

INTRODUCTION

Several types of human-computer interfaces exist in industry today. These include menu-driven systems, fill-in-the-blank systems, and parametric systems. Menu-driven systems are common in the microcomputer industry where users tend to be novice computer users. Parametric systems are used in applications such as airline reservation systems. Fill-in-the-blank systems are common in engineering environments since these applications require input of alphanumeric values, integers, and floating point values[1]. KEYLAB (the KEYword LAnguage Builder) is an interpreter that helps application program designers create a system that is controlled with keyword commands, yet has characteristics of a fill-in-the-blank system. The application program designer designs the command language so that the number of user type-ins is minimized, thus making the program easier to use. A programming aid entitled Keystone uses the command language design specification and generates the Fortran 77 code necessary to provide the specified command language. These programs have proven to be useful for our applications.

In a KEYLAB program, the user controls the program flow by entering a series of keywords and their associated arguments. The complete set of keywords for a given application program is called a command language. The KEYLAB parser detects syntax errors in the command string and immediately issues error messages to the user.

Command languages are based on the functional decomposition of the problem. When the program designer designs the command language, he takes human factors into account to try and make the command language more useable. Thus, the final command language is somewhat different from the original functional decomposition.

An example of a simple command language is given by the following structured list. The strings in quotations are short descriptions of the keyword functions.

```
ARTIST
  PICTURE      "Display the picture"
  SET          "Set variables"
    ROW        "Screen rows"
    HELPSET    "Display HELP"
    COLUMN     "Screen columns"
    BRUSH      "Drawing symbol"
    SCREEN     "Background symbol"
  STOP         "Terminate program"
  HELPART      "Display HELP"
  DISPLAY      "Display variables"
  DRAW         "Draw X1,Y1 to X2,Y2"
```

A code generator named Keystone reads this structured list and generates Fortran code with stub-subroutines at each leaf of the tree (e.g. ROW, COLUMN). The programmer then completes the Fortran code for each stub so that his program performs the desired function. If a keyword (ex. ROW) requires input, then the programmer adds a call to a KEYLAB subroutine to get this input. The programmer compiles the Keystone-generated code, and his own code and then links it to the KEYLAB library of subroutines to complete the application program. Using Keystone leads to more coherent software design since the application programmer concentrates primarily on coding the analytic sections of the code instead of on coding the human-computer interface.

Command languages have proven to be an effective human-computer interface for various data analysis programs at Grumman. TASKX, an advanced aircraft flutter analysis program [2] has used the KEYLAB prototype for three years and several new KEYLAB codes are being developed now. Both application programmers and program users are satisfied with the KEYLAB and Keystone programs.

KEYLAB - The KEYword LAnguage Builder

The application program users control KEYLAB program execution by typing keyword commands in response to prompts. The combination of a command language heirarchy and programmer-added syntax specify the valid sequences of commands. KEYLAB scans the input string from left to right without backtracking. KEYLAB terminates scanning and issues a message immediately when it finds a syntax error. KEYLAB also contains simple help facilities to help the user.

PROGRAM FLOW

The command language heirarchy can be thought of as a set of connected dictionaries of command keywords. Each dictionary in the command language is called a mode, and each mode contains keywords which either link to other modes, or cause execution of one of the program's analytical functions. The tree structure in Figure 1 depicts a command language heirarchy with seven dictionaries: COMMAND, INPUT, OUTPUT, TABULAR, GRAPHIC, INTERACTIVE, AND TAPE.

The user controls the branching through these modes by entering keywords. When the user types "OUTPUT GRAPHIC" from the command level, execution transfers to the GRAPHIC mode. If the user then wants to execute a command in the INTERACTIVE mode, he must enter a keyword which transfers execution to the COMMAND mode (e.g. INPUT). The user then enters the keywords to transfer program execution through the INPUT mode to the INTERACTIVE mode. KEYLAB provides no facility to transfer program execution from the GRAPHIC mode to the INTERACTIVE mode without first returning to the top of the command level tree structure. The command language designer takes this into account when he designs the command language heirarchy.

KEYLAB FEATURES

When users forget what keyword to type, they can type in a "?" at any point in the program execution to see the currently available keywords. Also, if they are not at the top of the functional decomposition, "??" shows the keywords that the user can type up to the top of the decomposition. For example, if the user is in the INPUT mode in Figure 1 and enters "?", he will see:

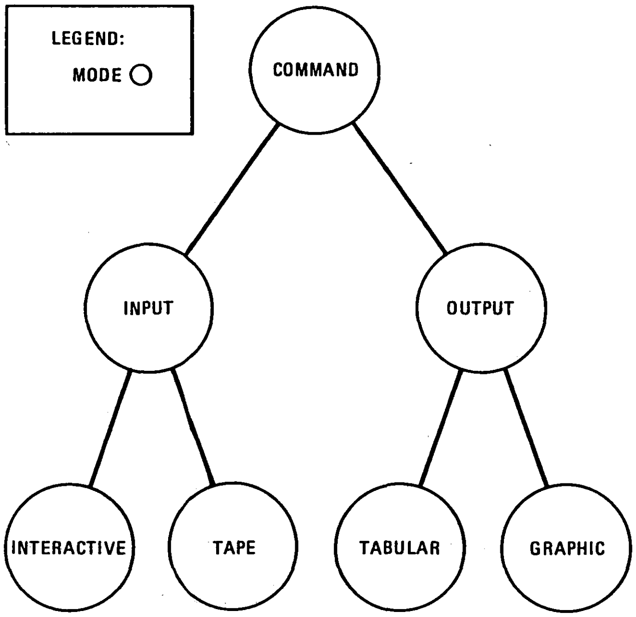


Figure 1. Example Command Language Design

```
THE CURRENTLY AVAILABLE KEYWORDS ARE:

INTERACTIVE       TAPE
```

If he enters "??" from the same point, he will remain in the current level and see keywords up to the top level.

```
THE CURRENTLY AVAILABLE KEYWORDS ARE:

INTERACTIVE       TAPE
INPUT             OUTPUT
```

They can also type in any keyword with the substring HELP to list out the keywords with a brief description. For the simple command language in the introduction, typing in HELPSET yields the following HELP screen:

```
ROW    - Number of rows in screen
COLUMN - Number of columns in screen
BRUSH  - Drawing symbol
SCREEN - Background
```

KEYLAB has a feature that allows the users to create their own procedures. Procedures specify a sequence of program functions to solve a specific problem. The users can create these procedures before they run the KEYLAB program by using the system editor. They can also create these procedures by using the KEYLAB procedure line editor named KEDIT. KEDIT lets the users edit procedures while they are running the application program. Thus, a user executes a procedure interactively by typing in the procedure name. He then examines the results, modifies the procedure with KEDIT and reruns the procedure. KEYLAB checks procedure as well as interactive type-in syntax.

KEYLAB stops parsing the input string when a syntactically incorrect token is encountered. For example, if the user types in "ROW 20 CO1", KEYLAB detects an error on the third token in the string. Since it is an undefined keyword, KEYLAB stops parsing the string, issues an error message, and reprompts the user. This ensures that the system is reliable regardless of a user error in keyword syntax [3].

The application program detects other errors such as range checking errors and syntax errors and directs KEYLAB to issue an error message. For example if the user types in "SET A 5.0" and the valid range for A is 10 < A < 20, then the application program tells KEYLAB to stop parsing the string. KEYLAB then issues the error message, and reprompts the user.

DESIGNING A COMMAND LANGUAGE

Previous sections mentioned that a command language for a program is based on the functional decomposition of the program, but is modified to account for usability. The command language design process is nontrivial and is essential for an effective application program design. This is a four step process.

The first step is to organize the requirements for the program on paper. The second step is to decompose the overall problem into a tree diagram and to assign keywords to each function on the tree diagram. The third step is to minimize the number of keywords that the user has to type in. The fourth step is to develop the command syntax requirements for the command language by imposing a logical order on the keyword type-ins. This section describes each step in the command language design process for an example program named Artist.

The method of organizing requirements for a given software system is chosen by the application program designer [4]. Here, we state the objective of the program and then describe the functions in the program that accomplish this objective. The Artist objective is:

> Artist allows the program user to
> draw lines on a terminal screen.

To do this, the program user must be able to do the following:

Choose the number of rows on screen
Choose the number of columns on screen
Choose a brush symbol
Choose a background symbol
Draw a line from X1,Y1 to X2,Y2
Display what was drawn
Display all variables that were chosen
Terminate the program

The second step in the command language design process is to organize these requirements into a functional decomposition. In our case, we will create a decomposition with four modes; SET, KEDIT, DISPLAY, and ARTIST. The ARTIST mode contains the keywords SET, STOP, KEDIT, HELPART, DISPLAY, and DRAW. The SET mode contains the keywords ROW, HELPSET, COLUMN, BRUSH, and SCREEN. The DISPLAY mode contains the keywords COLUMN, BRUSH, PICTURE, HELP, SCREEN, and ROW. The SET mode allows the user to set values for the desired variables. The DISPLAY mode allows the user to view what he has drawn and the variables that were set. The STOP keyword allows the user to terminate the application program. The KEDIT keyword allows the user to make use of the KEYLAB run-time editor. The HELP keywords display a list of the current keywords and their definitions. The DRAW keyword allows the user to specify a starting and ending point for a line. Figure 2 shows the entire functional decomposition for Artist.

The third step in command language design is to try to minimize the users type-ins by changing the decomposition. This is easily done by following two simple rules:

1. Decrease the depth of the decomposition by eliminating modes

2. Change the functionality of keywords to eliminate keywords.

If the user ran the program corresponding to the functional decomposition in Figure 2, he would have to type in the following command string to set all of the desired variables and display them.

```
SET ROW 10 COLUMN 20 BRUSH "*" SCREEN
"." DISPLAY ROW COLUMN BRUSH SCREEN
```

To draw a line on the screen, the user would type in the command string:

```
DRAW 1,1 5,5 DISPLAY PICTURE
```

To simplify this command string we can add a keyword to the ARTIST mode named PICTURE that displays the picture, and we can change the DISPLAY mode into a keyword that displays all of the variables that were set. This modified decomposition is shown in Figure 3.

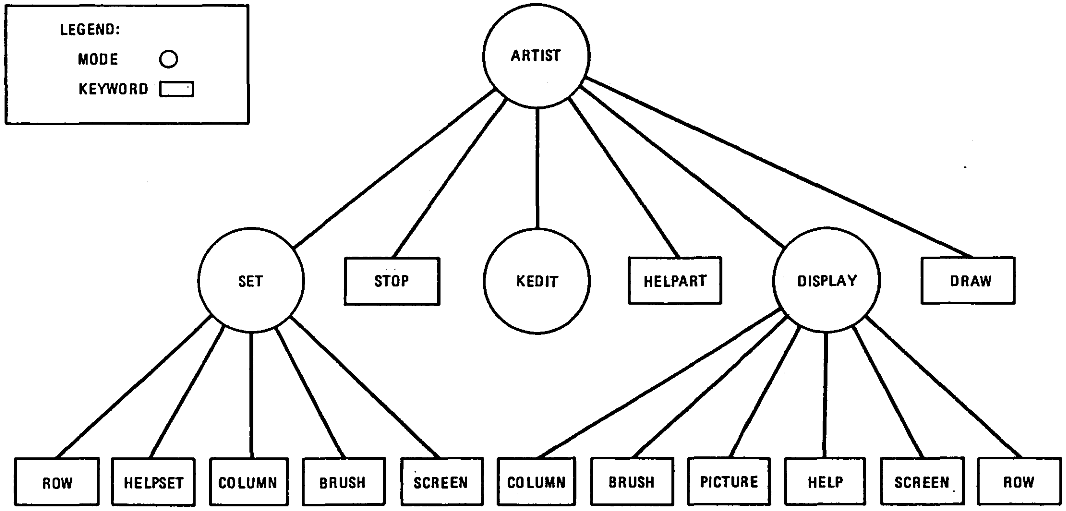


Figure 2. Artist Functional Decomposition

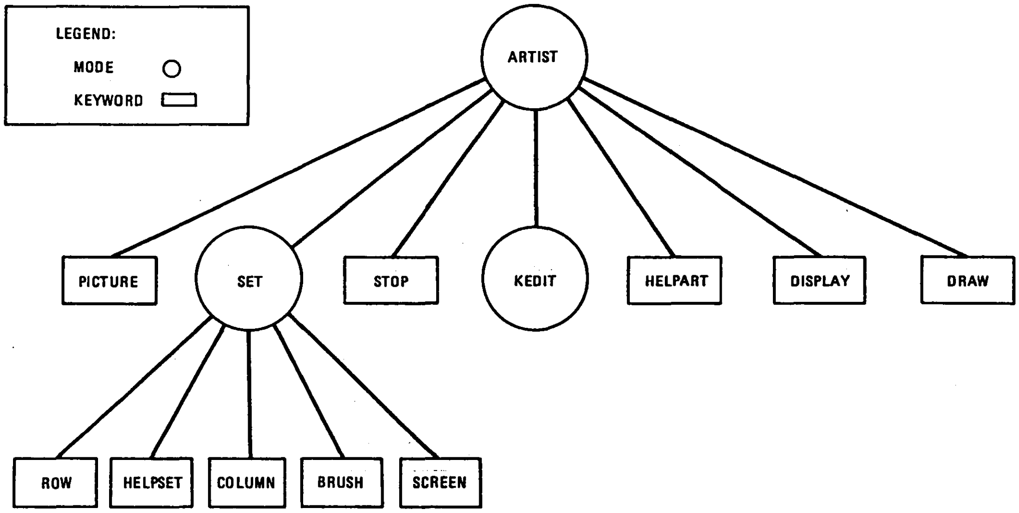


Figure 3. Command Language Design

The new command string to set and display all of the variables with four fewer tokens than before is:

```
SET ROW 10 COLUMN 20 BRUSH "*" SCREEN
"." DISPLAY
```

The new command string to draw the line is:

```
DRAW 1,1 5,5 PICTURE
```

With this acceptable sequence of type-ins, the program designer now adds rules to the command syntax so that the user is forced to type in keywords in the correct order. For example, the user should not draw until he has selected both brush and screen symbols. This means that the programmer doesn't let the user execute the DRAW command until BRUSH and SCREEN have been set. This syntax is known as Reverse Polish Notation [5] and helps to minimize the user type-ins. In our example, the syntax rules are:

1. Set BRUSH before DRAW.
2. Set SCREEN before DRAW.
3. Set ROW before DRAW.
4. Set COLUMN before DRAW.
5. DRAW before PICTURE.

This completes the Artist command language design. In Artist, we have only considered the case of one user input per keyword as in the case of ROW, or COLUMN. When more user inputs are desired per keyword, then the command language designer has to consider the possibility that the user may want to go back and change only one variable out of the entire string of inputs. Guidelines that we have found useful are:

1. When your program user is not likely to change one variable at a time (no mistakes), then you can program up to approximately seven inputs per keyword.

2. When your program user is likely to change one variable at a time (to change only one parameter in an analysis) then it is better to program only one input per keyword.

The following section describes how to take your final command language design, generate a code skeleton using Keystone, and then add the code necessary to complete the application program.

KEYSTONE

Keystone allows the application program designer to specify a command language and generate an executable program skeleton that runs with the specified keywords. The application program designer then adds the keyword functionality and syntax to complete the application program.

The programmer writes the command language design in the Keystone input file. Successive indentations in this file correspond to successive levels in the tree diagram. The input file for the example program Artist is shown in Figure 4.

```
ARTIST
  PICTURE
  SET
    ROW
    HELPSET
    COLUMN
    BRUSH
    SCREEN
  STOP
  KEDIT
  HELPART
  DISPLAY
  DRAW
```

Figure 4. Artist's Keystone Input File

Keystone reads this structured list and generates Fortran code with stub-subroutines corresponding to each keyword at each leaf of the tree (e.g. ROW, COLUMN, LINE, DISPLAY). The stub-subroutines contain print statements to which the programmer later replaces with the code for the keyword function. A stub subroutine looks like this:

```
SUBROUTINE row
Print (6,*) 'Code for ROW goes here'
RETURN
END
```

The programmer compiles this code and links it to the KEYLAB library. He then runs this skeleton via the command language that was specified in the functional decomposition. Thus, when the programmer runs the program corresponding to the decomposition in Figure 3 and types in the commands "SET ROW COLUMN DISPLAY", the program outputs the following lines:

```
The code for ROW goes here
The code for COLUMN goes here
The code for DISPLAY goes here
```

As the programmer develops analytical portions of the code, he substitutes them for the Keystone-generated Print statements. If the keyword requires input of a real, integer, or alpha- the programmer adds a call to a KEYLAB routine. This call tells KEYLAB what type of input (e.g. real input, alphanumeric input) to expect the user to enter at this point. To make sure that the program is reliable regardless of an error in the value of a number input, the application program must check that the number input by the user is within a certain range. The program then calls a KEYLAB subroutine that issues an error message such as "0 < ROW < 20".

The programmer also adds code so that the syntax specified in the command language design is present in the final application. For example, the program must check to make sure that the user SETs all variables such as SCREEN, ROW, COLUMN, and BRUSH before he attempts to DRAW a line. Thus, the user runs the application with the keywords specified by the program designer and the syntax imposed on the command language heirarchy.

EXPERIENCE WITH USE

In general, application program users, programmers, and managers have been satisfied with the capabilities of KEYLAB and Keystone as a method of human-computer interaction and as a code generation programming aid. These programs have been used successfully in data plotting application programs, aircraft trajectory estimation, flutter analysis, and others.

Program users find that it is easy to concentrate on their analysis while they are controlling the application program with keyword commands. They also like consistency of the human-computer interface across applications. Once they know KEYLAB features from learning one KEYLAB program, they can easily learn another KEYLAB program. Users like the procedure file capability since it allows them to set up a procedure with large groups of frequently executed commands and interactively enter any additional input that they need to complete the analysis. They can use these same procedures in either the interactive or batch mode.

The programmers' programming time is concentrated on the analytical sections of the code instead of on the human-computer interface details. Programmers also tend to produce easily maintainable code since the command language structure forces the programmers to write modular code. Also, programmers who write ANSI-standard Fortran 77 write portable code since the generated code is all portable.

From a manager's perspective, the Keystone input file provides a quick and simple way to verify that a given application program satisfies all of its functional requirements. This input file can be used to provide a working skeleton of the human-computer interface to the end-user before the programmer adds the functionality to the keywords. Thus, the user interface can be designed and tested separately from the analytical portions of the code.

The primary complaint with the existing system is that Keystone is initially frustrating to use because the input file structure is error-prone. Once a programmer has developed a successful template input structure, this problem is alleviated and Keystone users are generally satisfied with the ease with which they can subsequently modify their command language structure by adding or deleting modes and keywords.

The input file structure to Keystone could be changed to a less error-prone type of input file such as the Backus-Naur form (bnf) [6] . Currently, the Keystone code generator is adequate for even our largest (25,000 Fortran lines) stand-alone application programs so there is no reason to change the input file structure at this time.

It will probably be useful to incorporate some table-driven range-checking into KEYLAB so that the application programmer doesn't have to add code to check the range of user input. Thus, KEYLAB would then detect both type and range errors. This would place the burden of making sure the system is reliable regardless of user error on the KEYLAB interpreter instead of on the application programmer and would provide more consistent error messages to the user.

CONCLUSION

The KEYLAB interpreter provides a convenient and simple way to control a scientific application program. The command language design and the KEYLAB features add to the flexibility of the application program.

Command language designers go through a four-step process when they design a command language. First, they organize the requirements for the program on paper. Second, they organize the problem into a tree design and assign Keywords to each function. Third, they minimize the number of keywords. Finally, they impose rules on the command language structure to create a language syntax.

The Keystone program allows a program designer to quickly implement a language design into executable code. Once the command language is finalized, the programmer adds the analytical sections of code and the command syntax to the program. Also, if the command language design is changed later, the input file can easily be modified to account for these changes.

Both users and application programmers are satisfied with the KEYLAB interpreter and the Keystone code generator. In the future, however, the Keystone input file structure may be changed to make it less error-prone. KEYLAB may also be modified to detect out-of-range input errors.

REFERENCES

(1) Schneiderman, Ben, Software Psychology: Human Factors in Computer and Information Systems, Little, Brown, and Co., Boston, Massachusetts, pp. 238-241, 1980.

(2) Russo, M.L., Richards, P.T., and Perangelo, H.J., "Identification of Linear Flutter Models," presented at the AIAA 2nd Flight Testing Conference, Las Vegas, Nevada, November 16-68, 1983.

(3) Wasserman, Anthony I., Pircher, Peter A., Shewmake, David T. "Building Reliable Interactive Information Systems", Vol. SE-12, p. 147, 1986.

(4) Barnard, H., Metz, Robert F., and Price, Arthur L., "A Recommended Practice for Describing Software Designs: IEEE Standards Project 1016" IEEE Transactions on Software Engineering, Vol. SE-12, pp. 258-263, 1986.

(5) Tenenbaum, Aaron M., Data Structures Using Pascal, Prentice Hall, Inc. Englewood Cliffs, New Jersey, 1981.

(6) Pratt, Terrence W., Programming Languages: Design and Implementation, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, pp. 301-308, 1975.

ARTIFICIAL INTELLIGENCE ARENA

Artificial Intelligence Technology

TRACK CHAIR: Dr. Elaine Kant
Schlumberger Doll Research Center

Computer Vision

TRACK CHAIR: Prof. John Kender
Columbia University

Robotics

TRACK CHAIR: Prof. Richard Paul
University of Pennsylvania

Rule Based Systems

TRACK CHAIR: Prof. David Rine
George Mason University

Natural Language Processing

TRACK CHAIR: Prof. Robert Wilensky
University of California at Berkeley

SURVEY OF IMAGE QUALITY MEASUREMENTS

Ikram E. Abdou - Nicolas J. Dusaussoy

Department of Electrical Engineering
University of Delaware
Newark, DE 19716

ABSTRACT

Image quality is one of the most important factors in the evaluation of any imaging system. This measure can be used to compare the performance of the different systems and to select the appropriate processing algorithm for any given application. Image quality can be defined in general terms as an indicator of the relevance of the information presented by an image to the task we seek to accomplish using this image. Although there are many techniques developed to measure image quality, little has been done to unify the various concepts, or to test their relevance to specific applications. In this paper, we address the various issues related to image quality. We begin with a survey of the various factors that should determine the image quality measures. Then, we review and analyze the different analytical methods used in measuring the image quality for both image processing and image understanding applications. We survey experimental approaches to evaluate image quality in another section. Finally, we discuss some of the advantages of image quality measurements.

1. INTRODUCTION

The most important factors in the evaluation of any imaging system are the system cost, the processing speed, and the quality of the produced images. While the system cost and speed are dependent on many factors including the advancement in technology and the progress in hardware and software implementation, the image quality is a technology-independent measure, which can be used to compare the performance of the different systems and to select the appropriate processing algorithm for any given application. Image quality is determined by the relevance of the information presented by the image to the task we seek to accomplish using this image. Image quality can be divided into image fidelity, which measures the departure of a processed image from some standard image, and image intelligibility, which denotes the ability of man or machine to extract relevant information from an image.[1]

Image quality is calculated for various applications of imaging systems for which three different aspects of quality evaluation appear. First, quality evaluation techniques are necessary to measure the performance of an imaging device; the high performance of a system is usually indicated by the good quality of the final product. The second aspect is that quality evaluation techniques are the basis for the design of any image processing or understanding system as they form the framework for the optimization of the system. Finally, a very important aspect is assurance; quality assurance receives great attention because in many cases, it is necessary to ensure the presence or non-presence of features of interest in an image.

The techniques developed to measure image quality depend on the field of application. As we will see later, such applications are diverse and, therefore, there is no single standard procedure to measure image quality. However, there are some basic concepts that are reflected in the image quality measures. A summary of these ideas is given in [1, 2, and 3].

In this paper, we address the various issues related to image quality. We begin with a survey of the various factors that should be considered in determining image quality measures. Then, we review and discuss the different analytical methods used in measuring the image quality for both image processing and image understanding techniques. We survey experimental approaches to evaluate image quality, in another section. Finally, we discuss some of the advantages of image quality measurements.

2. FACTORS DETERMINING IMAGE QUALITY

In defining image quality, there are various factors reflecting the field of application, the end user, the parameters and characteristics of the system being evaluated.

2.1. The Field of Application

The field of application should affect the choice of an image quality measure because it determines the characteristics of the imaging task we would like to evaluate. These characteristics, specific to each field, are very well known in various applications including text reproduction that deals with binary images,[4,5] medical radiology,[6,7,8] industrial inspection,[9,10] and aerial photography.[3] Practical image quality measures must evaluate these characteristics and therefore, may vary according to the field of application. More general areas, where the evaluation of image quality is also of prime importance, such as image transmission,[11] general optical systems,[2,12] and electronic image display,[13] deal with several kinds of images. Quality measures should evaluate the overall performance of these systems. They are usually standard methods that can be used in most applications of these systems and they resume to rate image fidelity and image intelligibility.

CH2345-7/86/0000/0071$01.00 © 1986 IEEE

2.2. The End User of the System

Another important factor in evaluating image quality of a system is the end user which can be a human observer or a computer. The measures we use should reflect this fact. If the end user is human, the quality measures should be based on a psychophysical model. However, because of the difficulties encountered in taking subjective measurements, many researchers worked on establishing relations between subjective evaluation and some quantitative figure of merit.[14,15] On the other hand, if the images are processed by a computer, the image quality can be based on the classification accuracy of the system. One example is the use of receiver operating characteristic to evaluate diagnostic systems.[16] We believe that it is much easier to develop quantitative models for automated analysis systems, and we will discuss some of these ideas in a following section.

2.3. System Parameters and Characteristics

The various physical and mathematical parameters related to the specific application also need to be considered. For instance, in computerized tomography[17], there are a set of parameters relating to the signal being detected and to the imaging and displaying system. In this case, the object parameters such as size, shape, and contrast, and also the image forming parameters such as the spatial strength of signals used, the rate of collecting data, and noise are considered in the evaluation of the system. If the images are intended for a human observer, we need to define the display system, the viewing conditions, and the observer a priori knowledge. Of course, a more quantitative model can be established to determine the important image characteristics if the images are going to be processed automatically. All these parameters affect the success of an imaging system, and should be included in any evaluation.

Furthermore, several characteristics of the imaging system provide a guide to determine the useful quality measures. These are linearity, isotropy, contrast sensitivity, spatial invariance, spatial resolution, and also the image characteristics such as uniformity and sharpness.

3. ANALYTICAL METHODS

In the previous section, we discussed the various factors that can be considered in evaluating the quality of an image. In this section, we describe the different methods used to evaluate the image quality. We consider both image processing and image understanding techniques. In image processing, an input image is transformed to an output image in which features are enhanced, while in image understanding, the goal is to extract meaningful information and to derive a description of the scene. To achieve this goal, image understanding systems incorporate many scientific disciplines including image processing, pattern recognition, artificial intelligence, physics, and neurophysiology.[18]

In the design and the evaluation of either technique, we can use either analytical or experimental methods. In the analytical approach, models are developed to describe the interaction between the imaging device and some simple and standard signals; these models are used to predict the performance in practical applications. This approach is more general and does not require expensive equipment. However, because of the approximation involved in applying it, the results obtained may not accurately reflect the actual performance. Also, in many cases, we need to use simulation methods to measure the system response for the signals of interest because the complexity of the system makes the analytical solution extremely difficult. In the experimental approach, test patterns -or phantoms- are built to represent the various objects of interest; performance is then measured by either rating the images using human observers or by introducing a figure of merit which takes the different imaging parameters into account.[1]

In this section, we begin with a discussion of the analytical methods that are used to evaluate image processing systems. Then, we describe in detail the different models used to evaluate the performance of an image understanding system. In the following section, we will survey some experimental methods used in both image processing and image understanding systems.

3.1. Image Processing Systems

In measuring the performance of an image processing system, we distinguish between two cases depending on the ratio of the energy of the signal to that of the noise. If the signal -to- noise ratio (SNR) is high, the system performance is basically defined in terms of parameters such as the point spread function and the modulation transfer function. When the SNR is low, factors such as noise equivalent quanta, and the detection efficiency determine the system performance. The contrast can also be a tool to distinguish noiseless from noisy systems. Wagner defines a noiseless system as one processing high contrast images in contrast to noisy system processing low contrast images.[19] In the following section, we review methods that apply to noiseless, and noisy systems.

3.1.1. Noiseless System

When the SNR is high, system performance is determined by the sharpness and clarity of the images produced. Another important factor is the ability to resolve close patterns and to detect small size objects. A set of measurements in the spatial and frequency domains allows us to predict system performance for the above mentioned signals. We begin with the spatial measurements.

a. Spatial response measurements

The spatial response is one method that can be used to measure the image quality. The spatial response can be defined in terms of the response to an input point source $\delta(x,y)$ which is known as the point spread function (PSF) of the system.[3,20] The point spread function is also known by other names such as the impulse response, Green's function, or Fraunhofer diffraction pattern [3]. In optics, the PSF can be determined by the image of a star, considering the star as a point source. The PSF can be seen as the degradation caused by the system on the point source (blurring effect). Similarly, the line spread function (LSF) is determined when the point source is replaced by a line infinitely long and narrow. If the input is an abrupt step in brightness, i.e. a straight edge, then the output is the edge spread function (ESF).

The spread functions are a good basis to assess characteristics of the image processing technique. Spatial resolution can be deduced from the PSF. For instance, in optics, the PSF is a diffraction pattern with minimas and maximas, and the central bright patch is known as the Airy disc. The Rayleigh's criterion gives the resolution as the radius of the Airy's disc.[2] The shape, size, and diameter of the central lobe of the PSF not only are related to the spatial resolution, but also to the sharpness of the images being produced. Another characteristic, isotropy, applies if the PSF exhibits central symmetry.

Although the spread functions are simple and easy to determine in many applications, their use is limited to linear and spatial invariant systems. When these characteristics apply, the spread functions are really appropriate to the evaluation task, because they completely describe the performance; the system response to any arbitrary input image can be determined in terms of a convolution with the impulse response. On the other hand, if they do not apply, the determination of the performance is more delicate, and may be completely inaccurate.

The various spread functions are related for linear and spatial invariant processing techniques. First, it should be noted that the PSF has two dimensions, while the ESF and LSF have one. However, LSF and ESF exist for each line or edge orientation. If $PSF(x,y)$ represent the impulse response at a point of spatial coordinate (x,y), and $LSF(x')$ the line spread function for a line of orientation y', where x' is orthogonal to y' then, the LSF is the integral of the PSF in the direction y':[21]

$$LSF(x') = \int_{-\infty}^{+\infty} PSF(x,y)\,dy'. \tag{1}$$

Moreover, the LSF is the derivative of the ESF:

$$LSF(x') = \frac{d}{dx'}ESF(x'). \tag{2}$$

For isotropic systems, the LSFs and ESFs at different line and edge orientations are all identical because the PSF is rotationally invariant.

Another method to measure the spatial response is the resolving power which indicates the ability to discriminate fine details in an image.[3,13] Resolving power is expressed in terms of discernible line pairs per unique distance. The resolving power is easy to measure and can be applied to nonlinear as well as linear systems; therefore it is widely used in spite of the many questions in regard to its accuracy. Resolving power - or resolution - alone is not a sufficient measure for determining the relevance of the information presented by an image.[22] Although a practical and convenient guide to the performance, it is unreliable if it is used as a unique measure. Resolving Power is different for objects of various shape, size and contrast; hence the resolution in real images may be considerably different from the resolution measured.

Other spatial measurements that are helpful in determining the system performance, are the image uniformity, which measures the amplitude distortion for constant regions, and the system sensitivity, which determines the smallest amplitude that can be detected.[23,24]

b. Frequency response measurements

An important contribution to the evaluation of image quality has resulted from the two dimensional Fourier analysis. In the Fourier domain, another category of functions are defined: the transfer functions.

The Fourier transform of the PSF gives the optical transfer function (OTF):

$$OTF(u,v) = \iint PSF(x,y)\exp\left(2\pi i(ux+vy)\right)\,dx\,dy. \tag{3}$$

In general, the OTF is a complex function for which the International Comission on Optics defines its magnitude as the modulation transfer function (MTF) and its phase as the phase transfer function (PTF):

$$OTF(u,v) = MTF(u,v)\exp\left(i\,PTF(u,v)\right). \tag{4}$$

The term optical may seem to restrict these very general definitions to optical systems. However, the methods of treating an optical image are analogous to those of processing a signal. Transfer functions can serve to predict the quality of the formed images for any systems with few exceptions. They have the same inconveniences as those of the spread functions for which linearity and spatial invariance are necessary conditions to ensure confidence in the evaluation of the quality.

Indeed, they considerably simplify the analysis of systems whose characteristics are linearity and spatial invariance, because the OTFs and MTFs of a series of separable cascaded systems can be obtained by multiplying the OTFs and MTFs of each individual subsystem:[1,25]

$$OTF(u,v) = \prod_i OTF_i(u,v). \tag{5}$$

Conversely, the PSF requires convolution, a tedious process, of the separate PSFs. Just as the PSF, the OTF is a complete function to determine the output O of a system; the response is easily computed in terms of a product in the spatial-frequency domain of the input I with the OTF:

$$O(u,v) = I(u,v) \times OTF(u,v). \tag{6}$$

The OTF specifies the output that result from sinusoidal pattern input (as the PSF is the output that result from a point source) for all spatial frequencies. The sinusoidal pattern input undergoes changes in modulation expressed by the MTF and in phase expressed by the PTF, function of its spatial frequencies u and v. Its profile remains unaltered because no harmonic distorsions are introduced. Burgess noticed that the Fourier transform of the LSF gives a cross section of the OTF at the same angle.[26] For isotropic systems, all LSFs are identical; therefore the OTF is rotationally invariant as the PSF.

The MTF can be determined by measuring the ratio between the output and the input of a sinewave function. However, in some applications, it is easier to measure the square-wave response function (SWRF) and then determine the MTF in terms of the SWRF components. [27] Another frequency response function that can be applied to linear and nonlinear systems is the contrast function.[28]

Some of the image quality parameters in the frequency domain are; the amount of the amplitude, phase, and harmonic distortion, and the high- frequency cutoff.[3]

3.1.2. Noisy images

Any component of the signal that does not convey relevant information can be considered noise. Examples of noise are the fluctuation in the source signal, randomness in the detector output, and superimposed structures which are not related to the signals of interest.[29] If these variations are relatively large compared to the signal, they will be the limiting factor in the performance of the imaging system. Therefore, it is necessary to develop the appropriate tools to evaluate noisy systems. Such tools are derived from a statistical analysis, the basis of the theory of signal detection. One of the characteristics of a noise bound system is the smallest signal difference that can be detected. [19,30]

In dealing with noise, one simplified model considers the noise at various locations to be independent. For this model, a simple technique is to evaluate the standard deviation σ that measures the spread of the noise values. In computerized tomography, the noise is evaluated by experimentally measuring σ. A uniform phantom composed from only one material is scanned and reconstructed. Then, the sensitivity to the noise can be studied by estimating the standard deviation σ:

$$\bar{\sigma} = \left(\frac{\sum_{i,j} (\hat{f}(i,j) - \bar{\mu})^2}{N-1} \right)^{\frac{1}{2}}, \qquad (7)$$

where $\bar{\mu}$ is the sample mean and N the number of pixels. The advantages of using a uniform phantom is that the reconstructed image is free of artifacts and distortions. Therefore, σ is related only to the quantum and scattered noise. For certain reconstruction techniques, it is theoretically possible to relate the noise in projections to that in the reconstruction image.[31]

Such an assumption is not always true, especially after the noise is processed through the system. A more powerful descriptor related to the noise correlation is the noise power spectrum or Wiener spectrum.[32] The Wiener spectrum is determined by the Fourier transform of the autocorrelation function that indicates how the noise is correlated from point to point (Wiener-Khinchin theorem). The Wiener spectrum deals with all the various kind of noise appearing in the system and, furthermore, at various spatial frequencies. For instance, the Wiener spectrum $W_O(u,v)$ at the output of a linear system is easily related to the Wiener spectrum $W_I(u,v)$ at the input, MTF of the system, and additional noise components $W_a(u,v)$ at the output:

$$W_0(u,v) = W_I(u,v) \times MTF^2(u,v) + W_a(u,v). \qquad (8)$$

Another more accurate model to measure the performance of imaging detectors for which the noise structure is taken into consideration, is the noise equivalent quanta (NEQ) and detective quantum efficiency (DQE).[33] The noise equivalent quanta measures the ratio of the effective number of information bearing quanta used to form an image, and the detective quantum efficiency determines the efficiency of the photon detectors in terms of the ratio of NEQ to the actual number of quanta used. These descriptors combine together the MTF, the Wiener spectrum, and the sensitometric response of the system.

3.2. Image Understanding Systems

The idea of applying some image understanding measures to evaluate the image quality was introduced a few decades ago. However, the emphasis was on human image understanding. Recently, automatic image understanding has become feasible, and, for some specific applications, is the only practical solution.[18] In [34], an example of an automated computer tomography system is given. It is important to note, however, that very little has been done in this area, and more work is needed. In this section, we survey some of the methods that can be used to evaluate the performance of image understanding systems. This includes definitions of detection, orientation, recognition, and identification; the receiver operating characteristic (ROC); and measures based on information theory. In addition to these methods, any of the image processing techniques can be also used in evaluating image understanding systems.

3.2.1. Detection, orientation, recognition, and identification

The process of extracting information is called acquisition. There are different levels of acquisition which depend upon various parameters: size, contrast, noise, target and background characteristics, viewing conditions, automatic procedures, etc. Johnson considered the case when human observer processes the information and divide the acquisition into four categories: detection, orientation, recognition, identification.[15]
a. Detection: an object is present, but the recorded information is insufficient to assign a class to it.
b. Orientation: an object is present and its orientation, symmetry, or asymmetry are discerned.
c. Recognition: this is the first level of sorting the detected object into classes (house, man, animal, car, ...).
d. Identification: this is the second level of sorting the detected object into elements of specific classes (motel, policeman, dog, jeep, ...).

These four levels correspond to different levels of image quality: the first level requires the lowest quality; the last, the highest.

Johnson conducted a set of experiments to measure the minimum number of resolvable lines across the critical dimension of various targets required for each level of acquisition. Then, he produced tables giving the minimum number of resolved lines per minimum dimension of a specific object necessary at a stated level of acquisition. He found that, if the experimental conditions are the same for the targets and the test bars, the transformation targets into the number of resolved line pairs per critical dimension is independent of contrast and scene noise. His experimental results show that an object is detected for about 1 line pair, oriented for about 1.5, recognized for about 4, and identified for about 6.5 line pairs per critical object dimension.

Other methods to measure human performance are given in [35, and 36].

3.2.2. Receiver Operating Characteristic

The receiver operating characteristic (ROC) measures the relation between the probability of false alarm and correct detection as the decision level is changed. It is a useful tool for measuring image understanding performance when the image description is limited to few choices, such as the existence or nonexistence of abnormality in a patient,[16] or deficiency in a product. The ROC is more suitable for automatic image understanding systems.

3.2.3. Information theory measures

In some image understanding applications, we want to find the lowest level of signals that can be used to detect a given object. In other applications we may want to find the smallest features that can be detected. In either case, information theory may be helpful in establishing such bounds because it can relate the parameters of the system to the information required.

The information we would like to acquire may be small areas of size a (pixels) needed to be resolved and small density changes d over the area a needed to be detected. This information may be accepted with a tolerable probability P. P can be defined by $\frac{1}{R}$ where R is the number of pixels in which on the average one pixel will be classified incorrectly. The information required is represented by the triplet (a, d, P). Thus, the parameters should be determined to obtain the desired information in terms of (a, d, P). For instance, the exposure E necessary to record the features of interest can be related to (a, d, P). Examples of work in this area are given in [37, and 38]. Clearly, this analysis can be applied to either manual or automatic processing systems.

3.2.4. Image Segmentation measures

Many image understanding systems require dividing the scene into segments which have similar properties, as a first step toward classification. These properties may describe each pixel independently such as the signal intensity, or relate to a local area such as texture.[39] Recently, some effort was directed toward the performance evaluation of such systems. Methods to evaluate edge detection operators was described in [40]. In [41] various texture algorithms were compared. General analysis of image segmentation techniques were reported in [42, and 43]. Judging by the existing literature, it is clear that more needed to be done, especially in unifying these results with those obtained for image processing systems. All the work discussed so far corresponds to what is known as low level image understanding systems, because they make little use of the knowledge available about the contents of the scene. A more difficult problem is the analysis of high level image understanding systems; a survey of such systems with some qualitative comparison is given in [44].

An example is the work on low level image segmentation.[43,45] In this work, a segmentation error that measures the distance between two different segmentation outputs is used as a measure of performance. These measures are used to evaluate a segmentation system that divides images into areas and lines. Such methods are extremely important for the development of a reliable image analysis model.

4. EXPERIMENTAL METHODS

All the image quality measures, described in the previous section, are analytical. In this section, the experimental approach to measuring image quality is described. This approach is flexible and in many cases more accurate than the analytical model, because it can be made as close as possible to the actual application. Its disadvantage is that it is more costly and time consuming. In the early stages of design, experimental methods are based on simulation of the system. Later on, the same measurements are made using the actual system. Such measurements are usually more accurate and expensive than the analytical measurements.

Experimental measurements are usually made on test patterns that represent the interesting features in the images being processed. They can be as simple as a bar chart or as complicated as a 3-dimensional body phantom. The following are examples of such test patterns.

a. Resolving power charts.[3]
b. Spatial resolution patterns.[17,28]
c. Star test patterns.[26]
d. Perceptibility measurement patterns.[46]
e. Organ phantoms patterns.[28,31]
f. Industrial patterns.[47,48]

Basic resolution patterns consist of parallel bars having the same size and shape, with a spacing between bars equal to their width. These basic resolution patterns can be assembled in groups of different characteristics (contrast, size, orientation) to form the resolution test chart. Instead, a circular test pattern made of equally spaced wedges of identical sizes can be built to form a star pattern. The advantages of such a test pattern is that the density along a circle centered at its center is a square wave which has a spatial frequency proportional to the number of constituting wedges and to the inverse of the radius r of the circle. The patterns (a-d) are examples of charts constituted of bars and wedges. On the other hand, patterns or phantoms (e-f) represent as close as possible the objects (body tissues, industrial parts, ...) in the real application. They can be very complex and diverse depending on the application.

4.1. Qualitative methods

Any of the previous patterns can be processed using the imaging system being evaluated, and the performance can be measured either qualitatively or quantitatively. In the qualitative approach the processed images are shown under the same conditions to a group of observers who are asked to judge their quality. The observers may be chosen from expert or non-expert viewers.[1] Ultimately, qualitative methods are used as simple, natural, and somewhat reliable methods. However, to ensure confidence in the judgements, many observers must participate, which makes the qualitative methods cumbersome to employ. Although these subjective methods are suitable for the complicated test patterns described in e and f, they may also be applied to any of the other test patterns.

4.2. Quantitative measures

A better way for measuring the image quality for the patterns (a - d), is to introduce quantitative measures. These measures can be the resolution limits or the modulation transfer function. Also, it can be a one number merit function chosen such that it correlates with a subjective evaluation.[1] Examples of merit functions are the mean absolute error and the mean square error. The mean absolute error is defined in the general form:

$$MAE = \frac{\iint_a |\gamma(f(x,y)) - \gamma(\hat{f}(x,y))|\, dx\, dy}{\iint_a |\gamma(f(x,y))|\, dx\, dy}, \qquad (9)$$

and the mean squared error is:

$$MSE = \left(\frac{\iint_a [\gamma(f(x,y)) - \gamma(\hat{f}(x,y))]^2\, dx\, dy}{\iint_a [\gamma(f(x,y))]^2\, dx\, dy} \right)^{\frac{1}{2}}. \qquad (10)$$

where γ is an arbitrary operator, f the original image and $\hat{f}$ the degraded image. The domain of definition of f is a. They are the standard measures used to judge various processing techniques. Several transformations have been determined for the operator γ in order to compare the performance of the systems intended for human observers.[1] To get more reliable measures γ must be adapted to important factors such as the end user of the system.

5. USE OF IMAGE QUALITY MEASURES

In the previous sections, many measures of image quality were reviewed. In turn, we would like to discuss the usage of quality measures; quality measures can be applied to evaluate, to design, and to control imaging systems. We consider these three different important applications separately.

a. Performance evaluation. Image quality measures are used to evaluate the performance of processing and understanding techniques. It is important to use standard and qualified evaluation methods with confidence to compare several imaging systems.

b. Design and optimization. A second application of image quality measures is the design and the optimization of imaging systems. It is very important when we design a new system to minimize its cost, and also to maximize the processing speed and image quality. Therefore, the design engineers must be able to choose optimum parameters that satisfy image quality requirements.

c. Control. A more efficient system requires quality measures that can serve to control in real time the parameters of the system. Such a system is adaptive to the input images and can select the optimum parameters that produce the best image quality. These quality measures must estimate the relevance of the output information in real time to adapt the system parameters or the processing algorithms. Moreover, since they control the system, a large confidence in using them for this purpose is required. It is important to note that in certain field, particularly in image understanding, a successful search for new measures that can do this job without a reference to standard images is required.[45]

6. CONCLUSION

In this paper, we described the various factors that should determine the image quality measures. In addition, we discussed many of the methods that are used in the evaluation of image quality, for both the image processing and image understanding applications.

Many methods have been developed for the image processing application. They correspond to various approaches either analytical, such as spatial response measurements and frequency response measurements, or experimental. These measures have been adapted to many fields of application. However, there is still a need to introduce a unified model for the measurement of image quality and develop standard test patterns or test objects to compare the performance of various imaging systems.

On the other hand, very little has been done in the image understanding area. Major design measures that reliably indicate the performance of any image understanding technique still need to be developed. With the increasing interest in automatic image understanding. standard test patterns should be defined and used to measure the performance of various image understanding techniques. This will help in comparing the algorithms developed at different research centers.

Finally, judging from the existing literature, it is clear that while image quality evaluation has been applied to many areas such as medical radiology, aerial photography, industrial inspection, and design of optical systems, more work is still needed. We hope that this paper will inform the scientific community about the need for such work.

ACKNOWLEDGEMENTS

This work is supported in part by the University of Delaware Center for Composite Materials, funded by the NSF Engineering Research Centers under Grant CDR-8421409, and the University of Delaware Industrial Partner Program.

REFERENCES

[1] W. K. Pratt, *Digital Image Processing*, Wiley-Interscience, New York, 1978.

[2] W. B. Wetherell, "The Calculation of Image Quality", in *Applied Optics and Optical Engineering*, Vol. VIII, R. Shannon, J. Wyant, ed. Academic Press, New York, 1980.

[3] G. C. Brock, *Image Evaluation for Aerial Photography*, The Focal Press, London, 1970.

[4] D. M. Costigan, *Electronic Delivery of Documents and Graphics*, Van Nostrand Reinhold, New York, 1978.

[5] J. C. Stoffel, Ed., *Graphical and Binary Image Processing and Applications*, Artech House, Dedham, MA, 1982.

[6] A. G. Haus, Ed., *The Physics of Medical Imaging: Recording System Measurements and Techniques*, American Assoc. of Physicists in Medicine, New York, 1979.

[7] R. G. Waggener and C. R. Wilson, Eds., *Quality Assurance in Diagnostic Radiology*, American Assoc. of Physicists in Medicine, New York, 1980.

[8] *Quality Assurance in Nuclear Medicine*, World Health Organization, Geneva, 1982.

[9] D. A. Garrett and D. A. Bracher, *Real-Time Radiologic Imaging: Medical and Industrial Applications*, American Society for Testing and Materials, Philadelphia, 1980.

[10] R. S. Sharpe, *et al.*, Eds., *Quality Technology Handbook*, 4^{th} Ed., Butterworths, London, 1980.

[11] W. K. Pratt, Ed., *Image Transmission Techniques*, Academic Press, New York, 1979.

[12] L. R. Baker, Ed., *Quality Assurance in Optical & Electro-Optical Engineering*, Proc. of SPIE, Vol. 73, 1973.

[13] L. M. Biberman, *Perception of Displayed Information*, Plenum Press, New York, 1973.

[14] H. L. Snyder, "Image Quality and Observer Performance," in *Perception of Displayed Information*, L. M. Biberman, Ed., Plenum Press, New York, 1973.

[15] F. A. Rosell, and R. H. Willson, "Recent Psychophysical Experiments and the Display Signal -to-Noise Ratio Concept", in *Perception of Displayed Information*, L. M. Biberman, Ed., Plenum Press, New York, 1973.

[16] C. E. Metz, "Applications of ROC Analysis in Diagnostic Image Evaluation", in *The Physics of Medical Imaging: Recording Systems Measurements and*

Techniques, A. G. Haus, Ed., American Assoc. of Physicists in Medicine, New York, pp. 546-572, 1979.

[17] K. E. Weaver, and D. J. Goodenough, "Imaging Factors and Evaluation - Computed Tomography Scanning", in *The Physics of Medical Imaging: Recording Systems Measurements and Techniques*, A. G. Haus, Ed., American Assoc. of Physicists in Medicine, New York, pp. 309-355, 1979.

[18] L. E. Druffel, "Summary of the DARPA Image Understanding Research Program", in *Pattern Recognition Theory and Applications*, J. Kittler, *et al.*, Eds., D. Reidel Publishing Co., Holland, pp. 265-281, 1982.

[19] R. F. Wagner, "Toward a Unified View of Radiological Imaging Systems. Part II: Noisy Images", *Medical Physics*, Vol. 4, No. 4, pp. 157-174, 1977.

[20] S. Rowland, "Computer Implementation of Image Reconstruction Formulas", in *Image Reconstruction from Projections*, G. Herman Ed., Springer Verlag, Berlin, 1979.

[21] G. K. Sanderson, "Image Assessment: LSF and MTF", in *The Physics of Medical Imaging: Recording Systems Measurements and Techniques*, A. G. Haus, Ed., American Assoc. of Physicists in Medicine, New York, pp. 118-137, 1979.

[22] P. Rosenberg, "Detection, Detectability and Recognizability", *Photogrammetric Engineering*, Dec. 1971.

[23] J. R. Wolff, "Calibration Methods for Scintillation Camera Systems", in *Quantitative Organ Visualization in Nuclear Medicine*, P. J. Kenny, and E. M. Smith, Eds., University of Miami Press, Coral Gables, FL, pp. 229-259, 1971.

[24] W. J. MacIntyre, *et al.*, "Report of the ICRU Task Group on Scanning", in *Quantitative Organ Visualization in Nuclear Medicine*, P. J. Kenny, and E. M. Smith, Eds., University of Miami Press, Coral Gables, FL, pp. 167-184, 1971.

[25] M. L. Giger, and K. Doi, "Investigation of Basic Imaging Properties in Digital Radiography. 1. Modulation Transfer Function", *Medical Physics*, Vol. 11, No. 3, pp. 287-295, 1984.

[26] A. E. Burgess, "Interpretation of Star Test Pattern Images," *Medical Physics*, Vol. 4, No. 1, pp. 1-8, 1977.

[27] G. T. Barnes, "The Use of Bar Pattern Test Objects in Assessing the Resolution of Film/ Screen Systems", in *The Physics of Medical Imaging: Recording Systems Measurements and Techniques*, A. G. Haus, Ed., American Assoc. of Physicists in Medicine, New York, pp. 138-151, 1979.

[28] R. Sarper, "Evaluation of Imaging Factors in Nuclear Medicine", in *The Physics of Medical Imaging: Recording Systems Measurements and Techniques*, A. G. Haus, Ed., American Assoc. of Physicists in Medicine, New York, pp. 390-408, 1979.

[29] R. S. Holland, "Fundamentals of Radiographic Noise", in *The Physics of Medical Imaging: Recording Systems Measurements and Techniques*, A. G. Haus, Ed., American Assoc. of Physicists in Medicine, New York, pp. 152-171, 1979.

[30] P. Sprawls, *The Physics and Instrumentation of Nuclear Medicine*, University Park Press, Baltimore, 1981.

[31] L. Shepp, and B. Logan, "The Fourier Reconstruction of a Head Section", *IEEE Trans. on Nuclear Science*, Vol. NS-21, pp. 21-43, June 1974.

[32] M. L. Giger, K. Doi, and C. E. Metz, "Investigation of Basic Imaging Properties in Digital Radiography. 2. Noise Wiener Spectrum", *Medical Physics*, Vol. 11, No. 6, pp. 797-805, 1984.

[33] J. M. Sandrik and R. F. Wagner, "Absolute Measures of Physical Image Quality: Measurement and Application to Radiographic Magnification", *Medical Physics*, Vol. 9, No. 4, pp. 540-549, 1982.

[34] J. Winter, "Automated Computer Tomography Image Analysis Using Contour Map Topology", *IEEE Trans. on Medical Imaging*, Vol. MI-3, No. 4, pp. 163-169, Dec. 1984.

[35] J. A. Bencomo, L. M. Marsh, and T. J. Morgan, "An Evaluation of Digital Processing Capabilities for Improving Detection of Low Contrast Round Objects in a Radiography by Contrast Detail Diagrams", *Proc. of the SPIE*, Applications of Digital Image Processing VII, San Diego, pp. 379-383, Aug. 1984.

[36] A. E. Burgess, R. F. Wagner, and R. J. Jennings, "Human Signal Detection Performance for Noisy Medical Imaging", Intl' Workshop on Physics and Engineering in Medical Imaging, Pacific Grove, CA, pp. 99-105, March 1982.

[37] I. Brodie and R. A. Gutcheck, "Radiographic Information Theory and Application to Mammography", *Medical Physics*, Vol. 9, No. 1, pp. 79-95, 1982.

[38] I. Brodie and R. A. Gutcheck, "Radiographic Information Theory: Correction for X-Ray Spectral Distribution", *Medical Physics*, Vol. 10, No. 3, pp. 293-300, 1983.

[39] R. M. Haralick, "Statistical and Structural Approaches to texture.", Proc. of the IEEE, Vol. 67, No. 5, pp. 786-804, May 1979.

[40] I. E. Abdou, and W. K. Pratt, "Quantitative Design and Evaluation of Enhancement/Thresholding Edge Detectors", *Proc. of the IEEE*, Vol. 67, No. 5, pp. 753-763, May 1979.

[41] R. W. Conners, and C. A. Harlow, "A Theoretical Comparison of Texture Algorithms", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-2, No. 3, pp. 204-222, May 1980.

[42] E. L. Hall, *Computer Image Processing and Recognition*, Academic Press, New-York, 1979.

[43] M. D. Levine, and A. M. Nazif, "An Experimental Rule-Based System for Testing Low Level Segmentation Strategies", in *Multicomputers and Image Processing*, K. Preston Jr., and L. Uhr, Eds., Academic Press, New York, pp. 149-160, 1982.

[44] T. O. Binford, "Survey of Model-Based Image Analysis Systems", The Intl' Journal of Robotics Research, Vol. 1, No. 1, pp. 18-64, Spring 1982.

[45] A. M. Nazif, and M. D. Levine, "Low Level Image Segmentation: An Expert System", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-6, No. 5, pp. 555-577, 1984.

[46] R. Bollen, "Evaluation of Image Quality Performances of Radiographic Systems by the Perceptibility Curve (PC) Method", in *The Physics of Medical Imaging: Recording Systems Measurements and Techniques*, A. G. Haus, Ed., American Assoc. of Physicists in Medicine, New York, pp. 588-613, 1979.

[47] F. Hopkins, and I. Morgan, "X-Ray Computed Tomography for Aerospace Components", *Scientific Measurement Systems*, Technical Report, January 1983.

[48] P. Burstein, R. Mastronadi, and T. Kirchner, "Computerized Tomography Inspection of Trident Rocket Motors: A Capability Demonstration", *Materials Evaluation Journal*, pp. 1280-1284, November 1982.

ADDITIONAL REFERENCES

- R. B. Arps, *et al.*, "Character Legibility versus Resolution in Image Processing of Printed Matter", *IEEE Trans. Man. Machine Systems*, Vol. MMS-10, No. 3, pp. 66-71, Sept. 1969.
- P. C. Bunch, *et al.*, "A Free Response Approach to the Measurement and Characterization of Radiographic Observer Performance," *Proc. of SPIE*, Application of Optical Instrumentation in Medicine, VI, Vol. 127, Boston, MA, pp. 124-135, Sept. 1977.
- B. Chiang, *et al.*, "Spatial Resolution in Industrial Tomography", *IEEE Trans. on Nuclear Science*, Vol. NS-30, pp. 1671-1676, April 1983.
- R. T. Chin, and C. A. Harlow, "Automated Visual Inspection: A Survey", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-4, No. 6, pp. 557-573, 1982.
- G. Cohen, and F. A. DiBianca, "Information Content and Dose Efficiency of Computed Tomographic Scanners", in *The Physics of Medical Imaging: Recording Systems Measurements and Techniques*, A. G. Haus, Ed., American Assoc. of Physicists in Medicine, New York, pp. 356-365, 1979.
- N. Dusaussoy, "Performance Evaluation of Parallel Projection Computerized- Tomography", M. Sc. Thesis, University of Delaware, 1986.
- E. M. Granger, and L. R. Baker, Eds, "Image Quality: an Overview", *Proc. of SPIE*, Vol. 549, July 1985.
- R. Halmshaw, "Fundamentals of Radiographic Imaging," in *Real-Time Radiologic Imaging: Medical and Industrial Applications*, ASTM STP 716, D. A. Garrett, and D. A. Bracher, Eds., American Society for Testing and Materials, pp. 5-21, 1980.
- R. Y. Han, and R. J. Clark, "Characterization and Evaluation of Automatic Target Recognizer Performance," *Proc. of the SPIE*, Applications of Digital Image Processing VII, San Diego, pp. 341-351, Aug. 1984.
- K. M. Hanson, "Detectability in the presence of Computed Tomographic Reconstruction Noise", *Proc. of SPIE*, Application of Optical Instrumentation in Medicine, VI, Vol. 127, Boston, MA, pp. 304-312, Sept. 1977.
- G. Herman, *Image Reconstruction from Projections*, Academic Press, New York, 1980.
- L. Kaufman, "Nuclear Medicine Imaging," in *Medical Imaging Techniques. A Comparison*, K. Preston, Jr., *et al.*, Eds., Plenum Press, New York, pp. 263-285, 1979.
- W. Kropatsch, "Segmentation of Digital Images Using A Priori Information about the Expected Image Contents", in *Pictorial Data Analysis*, R. M. Haralick, Ed., Springer Verlag, Berlin, pp. 107-132, 1983.
- J. D. McGee, D. McMullan, and E. Kahan, Eds., *Photo-Electronic Image Devices*, Academic Press, London, 1966.
- J. D. McGee, D. McMullan, E. Kahan, and B. L. Morgan, Eds., *Photo-Electronic Image Devices*, Academic Press, London, 1969.
- J. D. McGee, D. McMullan, and E. Kahan, Eds., *Photo-Electronic Image Devices*, Academic Press, London, 1972.
- P. R. Moran, "A Physical Statistics Theory for Detectability of Target Signals in Noisy Images. 1. Mathematical Background, Empirical Review, and Development of Theory", *Medical Physics*, Vol. 9, No. 3, pp. 401-413, 1982.
- C. Parma, *et al.*, "Experiments in Schema-Driven Interpretation of a Natural Scene", in *Digital Image Processing*, J. Simon, and R. Haralick, Eds., D. Reidel Publishing Co., Holland, pp. 449-509, 1981.
- G. U. V. Rao, "Measurement of Modulation Transfer Functions," in *Quality Assurance in Diagnostic Radiology*, R. G. Waggener and C. R. Wilson, Eds., American Assoc. of Physicists in Medicine, New York, pp. 79-104, 1980.
- P. Reimers, *et al.*, "Recent Developments in the Industrial Application of Computerized Tomography with Ionizing Radiation", *Journal of NDT International*, Vol. 17, No. 4, pp. 197-207, Aug. 1984.
- H. Roehrig, *et al.*, "X-Ray Image Intensifier Video System for Diagnostic Radiology: Part 1, Design Characteristics", *Proc. of SPIE*, Application of Optical Instrumentation in Medicine, VI, Vol. 127, Boston, MA, pp. 216-225, Sept. 1977.
- C. Scheid, "Performance Measurements of Fluoroscopic Systems," in *Real-Time Radiologic Imaging: Medical and Industrial Applications*, ASTM STP 716, D. A. Garrett, and D. A. Bracher, Eds., American Society for Testing and Materials, pp. 168-179, 1980.
- G. Seeley, *et al.*, "Psychophysical Evaluation Correlated with System Measures: Part 2," *Proc. of SPIE*, Application of Optical Instrumentation in Medicine, VI, Vol. 127, Boston, MA, pp. 226-231, Sept. 1977.
- P. F. Sharp, "Physical Limitations to the Quality of X- and Gamma Ray Images", in *Technical Advances in Biomedical Physics*, P. P. Dendy, D. W. Ernst, and A. Sengun, Eds., Martinus Nijhoff Pub, The Hague, pp. 219-234, 1984.
- P. F. Sharp, "The Presentation of Photon-Limited Images," in *Technical Advances in Biomedical Physics*, P. P. Dendy, D. W. Ernst, and A. Sengun, Eds., Martinus Nijhoff Pub, The Hague, pp. 235-258, 1984.
- R. Shaw, "Some Modern Aspects of Image Evaluation", in *The Physics of Medical Imaging: Recording Systems Measurements and Techniques*, A. G. Haus, Ed., American Assoc. of Physicists in Medicine, New York, pp. 390-408, 1979.
- F. C. Southon, "CT Scanner Comparison", *Medical Physics*, Vol. 8, No. 1, pp. 62-75, 1982.
- P. Sprawls, and J. C. Hoffman, "Image Quality in Computerized Axial Tomography", *Proc. of SPIE*, Application of Optical Instrumentation in Medicine, IV, Vol. 70, Atlanta, GA, pp. 310-316, Sept. 1975.
- S. R. Sternberg, "Industrial Morphology", *Proc. of the SPIE*, Applications of Digital Image Processing VII, San Diego, pp. 202-213, Aug. 1984.
- E. Takenaka, T. Iinuma, and M. Inoue, "New Phantoms for Measuring Low Contrast Resolution and Comparison of Several CT Scanners when Using Them", Proc. Intl. Workshop on Physics and Engineering in Medical Imaging, Pacific Grove, CA, pp. 203-208, March 1982.
- J. P. J. de Valk, "Diagnostic Processing and Analysis of Medical Images", in *Technical Advances in Biomedical Physics*, P. P. Dendy, *et al.*, Eds., Martinus Nijhoff Pub, The Hague, pp. 271-286, 1984.
- M. V. Yester, and G. T. Barnes, "Geometrical Limitations of Computed Tomography (CT) Scanner Resolution", *Proc. of SPIE*, Application of Optical Instrumentation in Medicine, VI, Vol. 127, Boston, MA, pp. 296-302, Sept. 1977.
- I. T. Young, "The Use of Digital Image Processing Techniques for the Calibration of Quantitative Microscopes", *Proc. of the SPIE*, Applications of Digital Image Processing, Geneva, Switzerland, pp. 326-335, April 1983.

A Spatial Knowledge Structure for Image Information Systems Using Symbolic Projections

Shi-Kuo Chang and Erland Jungert*
Information Systems Laboratory
Department of Computer Science
University of Pittsburgh

ABSTRACT: We present a new pictorial data structure for image information systems. This pictorial data structure consists of a run-length encoded basic data structure for images, symbolic projections representing the orthogonal relations among objects or subobjects in an image, and rules to derive complex spatial relations from the symbolic projections. Based upon this approach, a knowledge-based image information system can be designed, which supports spatial reasoning, image information retrieval, and image manipulation.

1. Introduction

Image information systems (IIS) are heavily dependent on how images are stored in an image database according to certain pictorial data structures. Many different pictorial data structures have been proposed: some are pixel-oriented [CHOCK84], some utilize quadtrees or their variants [SAMMET84], and some are vector-based [JUNGERT85]. To make an image information system more intelligent and more flexible, it is important that the system be capable of integrating a knowledge-base into its pictorial data structure. The pictorial data structure should also be object-oriented, so that users can easily retrieve, visualize and manipulate objects in the image database.

In this paper, we present a new approach for image information system design, based upon a pictorial data structure using symbolic projections [CHANG86]. The basic data structure for image encoding is the run-length code (RLC) [JUNGERT86]. The technique of symbolic projection is used to generate a description of the RLC-encoded symbolic picture [CHANG86]. When the objects have complex shapes and their minimum enclosing rectangles overlap, orthogonal relations can be found to preserve basic spatial relations. Production rules can then be applied to derive more complex spatial relations from the 2D string representation of symbolic projections. Based upon this approach, a knowledge-based image information system can be designed, which supports spatial reasoning, image information retrieval, and image manipulation.

Figure 1 is the schematic diagram of an image information system with the proposed pictorial data structure. The icon-oriented user interface utilizes the following modules to perform its function:

-simple query processor: retrieves objects based upon their names or coordinates.
-knowledge-based query processor: processes complex queries involving spatial relations.
-spatial operators: for creating new objects or testing certain spatial relations.
-image overlay system: performs image display, window management and image overlay.
-image generator: converts RLC structures into images.

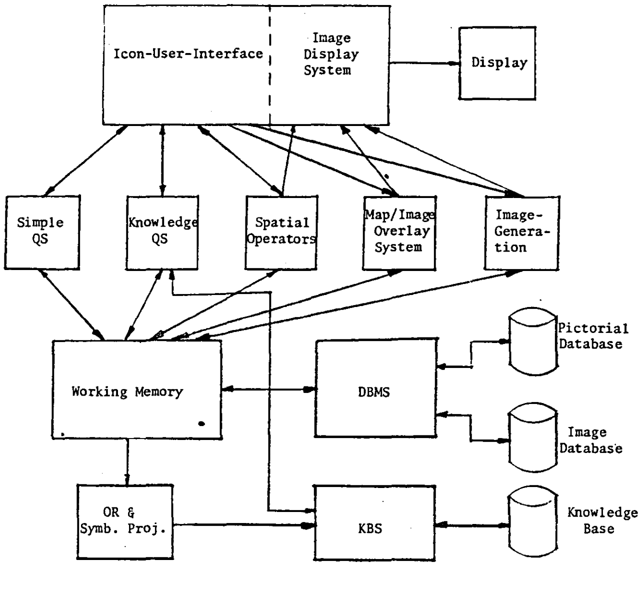


Figure 1 Schematic Diagram of an image information system

Raw images are stored in an image database. The image attributes, the symbolic projections, and the RLC encoded images are

CH2345-7/86/0000/0079$01.00 © 1986 IEEE

stored in a pictorial database managed by the DBMS (database management system). The production rules are stored in the knowledge-base and managed by the KBS (knowledge-base management system). A working memory is used to keep all temporary data, such as newly created objects, extracted orthogonal relations, derived spatial relations, and other kinds of application-dependent data.

The image information system with the proposed data structure is especially suitable for geographic information systems, but it will also be suitable for many other application areas.

2. Basic Data Structure using Run-Length Codes

Run length code has been used primarily for compacting image data. It can be used as a basic object-oriented data structure in image information systems. The principle of the data structure is illustrated in Figure 2. In this example, map overlays are used. A map overlay is an image used for map production and contains normally just one single object type, e.g., lakes or forests. Figure 2 shows that contrary to the general RLC, the information outside the objects is not saved. Only the object information or the object lines that belong to an object are saved. However, so far we cannot talk about objects, just about lines belonging to an object of some type. For each line we keep the coordinates of its starting point, the length of the line and the type of the object to which the line belongs. This structure is well adapted to a relational database. The relational scheme of the run-length coded lines is:

RLC - C (y, x, length, type, nc)

where the identifying key is underlined.

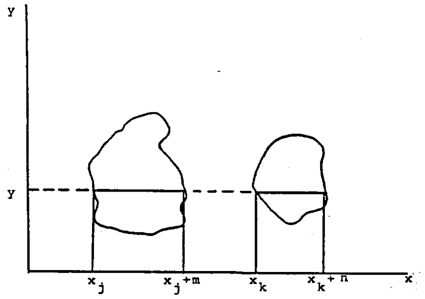


Figure 2 Run-length encoding of image

To identify each single line, only the coordinates of the starting point of each line are needed. As will be seen later the order of the coordinates are of importance. The attribute nc, is used for identification of objects and will be discussed further below.

So far we have only discussed object types which are spatially distributed in such a way that it is possible to "walk" around them, i.e., they are closed object types. The RLC encoding is, however, also valid for object types that are linear, e.g. roads, or of point type, e.g. landmarks. For these types, the lines are always one pixel long. The latter types are trivial compared to the closed types. It has to be pointed out that the linear object types are equivalent to vector structures. The relations for the linear and the point types are respectively:

RLC - L (y, x, type, nc)

RLC - P (y, x, type, nc)

In an object-oriented system, all objects must be identified in an uniform way. This can be done either by using the name of the object or by using unique coordinates. Here both methods are used. Names are used because the users are more familiar with them.

In order to simplify retrieval of objects in the database each object in the database will include coordinates that correspond to the minimum enclosing rectangle of the object. Figure 3 illustrates the correspondence between the object and the rectangle.

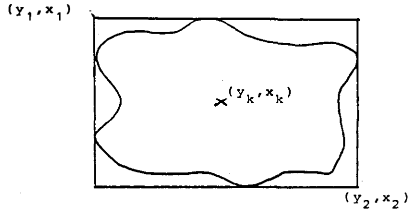


Figure 3 Minimum enclosing rectangle of an object and its key.

The object relations corresponding to the description given above will include not only the given attributes but also application dependent attributes, e.g. the depth or ph-value of a lake. However, such attributes will not be discussed further here. An object relation for closed object types will therefore look like:

OBJ - CR (Name, Yk, Xk, nc, Y1, X1, Y2, X2)

The relations for linear and point object types will be similar although there are no rectangles needed for the point object types.

The nc attribute is a unique integer

attribute that corresponds to the name of the object. Since RLC relations always include this attribute, there is a logical link between each RLC-line and the object in the object relation.

By using run-length encoded data a user can define an image (such as a map) covering a certain area and display it. Furthermore, it is easy to display the data, because compared to e.g. vector data the problem of cutting the image in order to make it fit into the display device is just a question of cutting horizontal run-length lines.

An important advantage in using the RLC data structure is that since the lines are first ordered with respect to their y-coordinates, all lines on a certain y-level can be accessed in sequence. Therefore, all lines within the interval [Ymax, Ymin] can be read in sequence and the lines displayed in the same sequential order. Consequently, the process of displaying and reading image data is always done in sequence and it does not have to be changed when handling the same relation. This process is illustrated in Figure 4.

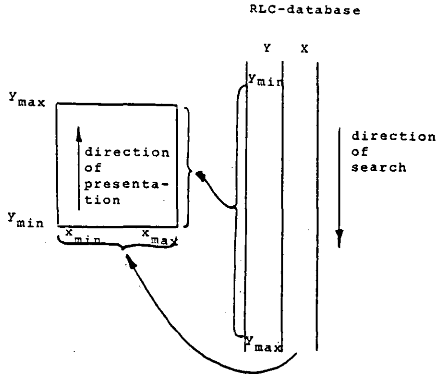


Figure 4 Sequential presentation of image data from RLC.

The RLC structure has some additional advantages for map presentation. First, the method is scale independent for at least some scale intervals. Hence it is fairly simple to implement zooming. Second, there is no need to implement any "fill" operations for the displaying of objects, because filling will be performed automatically when interpreting RLC objects. Similarly, the holes inside an object will be generated automatically when the object is displayed.

3. Spatial Operators on RLC Objects

The creation of new objects from existing RLC objects can be done by set theoretic operations such as union, intersection, exclusive or, etc.

(a) Union: Suppose 01 and 02 are two overlapping objects of closed type (i.e. the boundary is a non-self-intersecting closed curve). As illustrated in Figure 5a, the union of these two closed objects, 01 ∪ 02, can be constructed by combining the RLC for each specific y value as follows (assuming x1 =< x2):

```
     01: y, x1, m
     02: y, x2, n
01 u 02: y, x1, m ; x2, n   (for x1+m<x2)
         y, x1, x2+n-x1     (for x1+m>=x2)
```

(b) Intersection: Suppose 01 and 02 are two overlapping objects of closed type. As illustrated in Figure 5b, the intersection of these two closed objects, 01 ∩ 02, can be constructed by intersecting the RLC for each specific y value as follows (assuming x1 =< x2):

```
     01: y, x1, m
     02: y, x2, n
01 n 02: y, x2, x1+m-x2     (for x1+m>=x2)
         no line            (for x1+m<x2)
```

(c) Exclusive Or: As illustrated in Figure 5c, the exclusive-or of two overlapping closed objects can be constructed as follows (assuming x1 =< x2):

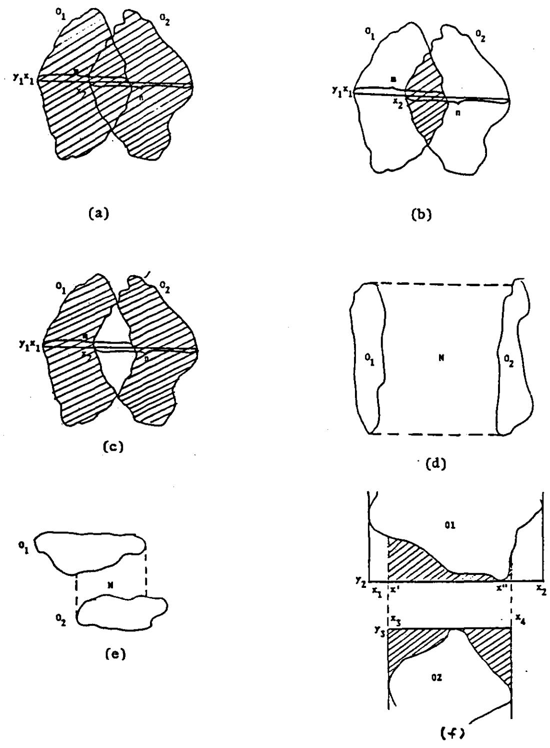


Figure 5 Spatial operators on RLC objects

```
        01: y, x1, m
        02: y, x2, n
 01 xor 02: y, x1, x2-x1; x1+m, x2+n-x1-m
            (for x1+m>=x2)
            y, x1, m; x2, n
            (for x1+m<x2)
```

The above described operators can be used to create new objects from existing objects, or to test the relationships among objects. For example, to decide whether an object 01 is contained in another object 02, it suffices to show that 01 ∪ 02 = 02, or equivalently, 01 ∩ 02 = 01. Using this test, we can find the objects situated inside another closed object, for example, the islands inside a lake. The complement of an object 01 contained in another object 02 can be found by 01 xor 02.

To decide whether a point object (or a linear object) 01 is contained in another object 02, we can check the RLC for each specific y value as follows:

```
     01: y, x1
     02: y, x2, n
   Test: x2 =< x1 =< x2 + n
```

(d) Horizontal Extension: Suppose 01 and 02 are two nonoverlapping closed objects. As illustrated in Figure 5d, the horizontal extension of 01 and 02 can be constructed as follows:

```
     01: y, x1, m
     02: y, x2, n
  01~02: y, min(x1,x2), (max(x1+m,x2+n)
                         -min(x1,x2))
```

In the above, min(x,y) = x if y is undefined, and max(x,y) = x is y is undefined.

(e) Vertical Extension: Similarly, we can define the vertical extension of two nonoverlapping closed objects 01 and 02, as illustrated in Figure 5e. To facilitate computation, the vertical RLC for 01 and 02 should first be found. The vertical extension of 01 and 02 can be constructed as follows:

```
     01: x, y1, m
     02: x, y2, n
  01|02: x, min(y1,y2), (max(y1+m,y2+n)
                         -min(y1,y2))
```

Figure 5f illustrates an alternative way to compute the vertical extension by first finding x' = max(x1,x3) and x" = min(x2,x4), and then taking the union of the two shaded areas and the rectangle (x',x"; y2,y3).

4. Symbolic Projections

We now describe the methodology of symbolic projections [CHANG86]. Let V be a set of symbols, or the vocabulary. Each symbol could represent a pictorial object, a pixel, etc. A 1D string over V is any string x1 x2 ... xn, n >= 0, where the xi's are in V. A 2D string over V, written as (u,v), is a pair of 1D strings.

We can use 2D strings to represent pictures in a natural way. As an example, consider the picture shown in Figure 6.

```
-------------
| d |   |   |
-------------
|   | b | c |
-------------
| a | a |   |
-------------
```

Figure 6   A symbolic picture f

The vocabulary is V = {a, b, c, d}. The 2D string representing the above picture f is,

(u,v) = (a d < a b < c , a a < b c < d)

In the above, the symbol '<' denotes the left-right spatial relation in string u, and the below-above spatial relation in string v. Therefore, the 2D string representation can be seen to be the symbolic projection of picture f along the x- and y- directions.

A symbolic picture f is a mapping M x M -> W, where M = {1, 2, ..., m}, and W is the power set of V (the set of all subsets of V). The empty set {} then denotes a null object. In Figure 6, the "blank slots" can be filled by empty set symbols, or null objects. The above picture is,

f(1,1) = {a}	f(1,2) = {}	f(1,3) = {d}
f(2,1) = {a}	f(2,2) = {b}	f(2,3) = {}
f(3,1) = {}	f(3,2) = {c}	f(3,3) = {}

In [CHANG86], we have shown that given f, we can construct the corresponding 2D string representation (u,v), and vice versa, such that all left-right and below-above spatial relations among the pictorial objects in V are preserved. In other words, let R1 be the set of left-right and below-above spatial relations induced by f. Let R2 be the set of left-right and below-above spatial relations induced by (u,v). Then R1 is identical to R2, for the corresponding f and (u,v).

2D string representation provides a simple approach to perform subpicture matching on 2D strings. The rank of each symbol in a string u, which is defined to be one plus the number of '<' preceding this symbol in u, plays an important role in 2D string matching. We denote the rank of symbol b by r(b). The strings "ad<b<c" and "a<c" have ranks as shown in Table 1:

string v	string u
a d < b < c	a < c
1 1 2 3	1 2

Table 1 Ranks of strings

A substring where all symbols have the same rank is called a local substring.

A string u is contained in a string v, if u is a subsequence of a permutation string of v.

A string u is a type-i 1D subsequence of string v, if (a) u is contained in v, and (b) if a1 w1 b1 is a substring of u, a1 matches a2 in v and b1 matches b2 in v, then

(type-0) r(b2)-r(a2)>=r(b1)-r(a1) or r(b1)-r(a1)=0

(type-1) r(b2)-r(a2)>=r(b1)-r(a1)>0 or r(b2)-r(a2)=r(b1)-r(a1)=0

(type-2) r(b2)-r(a2)=r(b1)-r(a1)

Now we can define the notion of type-i (i=0,1,2) 2D subsequence as follows. Let (u,v) and (u',v') be the 2D string representation of f and f', respectively. (u',v') is a type-i 2D subsequence of (u,v) if (a) u' is type-i 1D subsequence of u, and (b) v' is type-i 1D subsequence of v. We say f' is a type-i sub-picture of f.

In Figure 7, f1, f2 and f3 are all type-0 sub-pictures of f; f1 and f2 are type-1 sub-pictures of f; only f1 is type-2 sub-picture of f. The 2D string representations are:

f	(ad<b<c, a<bc<d)
f1	(a<b, a<b)
f2	(a<c,a<c)
f3	(ab<c, a<bc)

```
-------------
| d |   |   |
-------------
|   | b | c |
-------------
| a |   |   |
-------------

      f

---------       ---------       ---------
|   | b |       |   | c |       | b | c |
---------       ---------       ---------
| a |   |       | a |   |       | a |   |
---------       ---------       ---------

   f1              f2              f3
```

Figure 7 Picture matching examples

Therefore, to determine whether a picture f' is a type-i sub-picture of f, we need only determine whether (u',v') is a type-i 2D subsequence of (u,v). The picture matching problem thus becomes a 2D string matching problem. Efficient 2D string matching algorithms have been developed [CHANG86] and applied to pictorial information retrieval problems.

5. Orthogonal Relations

All run-length encoded objects have the minimum enclosing rectangle available, hence three types of spatial relations between objects can be identified. These are for objects with:

-nonoverlapping rectangles
-partly overlapping rectangles
-completely overlapping rectangles

The case with non-overlapping rectangles is trivial and will never cause any problems because the object relations are simple. The other two might sometimes cause problems, especially when one of the objects partly surrounds the other. Figure 8 demonstrates a problem of this type. The fundamental issue here is to find a method that easily describes the relations between the objects. The method is called orthogonal relations, because it deals with spatial relations that are orthogonal to each other.

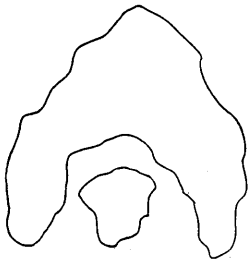


Figure 8 Two objects with overlapping MERs

The basic idea is to regard one of the objects as a "point of view object" (PVO) and then view the other object in four direction (north, east, south, west). Hence, at least one or at most four subparts of the other object can be "seen" from the PVO. The part of the object that actually is "seen" is in the interval where the two rectangles overlap, partly or completely. This is illustrated in Figure 9 and Figure 10. The subobject can be regarded as point objects, i.e. the centroid of the rectangle that enclose each subobject. It is a fairly simple operation to identify and generate these points from the RLC. The next step is then to identify the relation between the objects by using the 2-D projection method described in section 3.

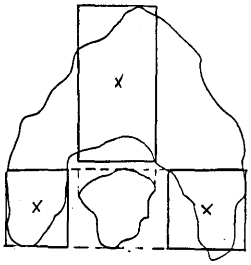


Figure 9 The PVO and its corresponding orthogonal relations

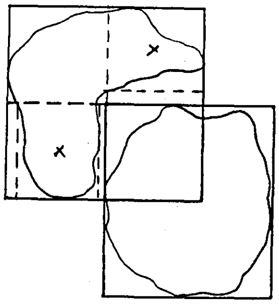


Figure 10 The PVO and its corresponding orthogonal relations for partly overlapping MERs

Each one of the sub-objects constitutes an orthogonal relational object of the original object and since they are regarded as points, a sparse vector description of the original object is generated. From this viewpoint it does not matter whether the original object is of closed or linear type. This makes the methods powerful. However, it is of importance that the subobjects are interpreted correctly. Figure 11a shows a correct interpretation of a north and a west segment while the interpretation of the same element in Figure 11b is erroneous. The natural interpretation is to look clockwise. Hence, nine different combinations can be identified.

2 points: N - E, E - S, S - W, W - N

3 points: N - E - S, E - S - W, S - W - N, W - N - E

4 points: N - E - S - W

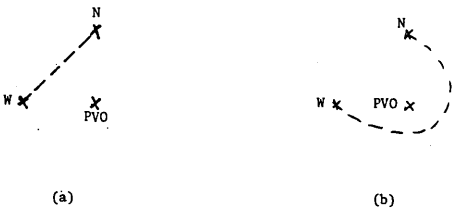


Figure 11 A correct (a) and an erroneous (b) interpretation of orthogonal relations

No other interpretations are allowed. It is also possible to regard the element in between the orthogonal ones. But this is not necessary since enough information is available anyway (see Section 5 for further discussions).

The technique of finding orthogonal relations is described in the following algorithm.

```
Procedure Ortho(x,y)
begin
  /*Ortho(x,y) finds orthogonal relations
    of object x with respect to object y*/

  /*find the minimum enclosing rectangles*/
  find Mer(x); find Mer(y);

  /*find four relational objects of object y
    intersecting the extensions of object x*/
  y-W = W-extension(Mer(x)) ∩ Mer(y);
  y-E = E-extension(Mer(x)) ∩ Mer(y);
  y-N = N-extension(Mer(x)) ∩ Mer(y);
  y-S = S-extension(Mer(x)) ∩ Mer(y);

  return( {y-W,y-E,y-N,y-S} );
end
```

6. A Knowledge-based Approach to Spatial Reasoning

From the 2D string representation, we can derive the spatial relations without any loss. From these spatial relations, we can derive even more complex spatial relations. Therefore, combining the 2D string representation with a knowledge-based system, we can provide flexible means of spatial reasoning, image information retrieval and management.

The knowledge-based approach to spatial reasoning is illustrated by two examples below.

Example 1: An island outside a coast line.

Figure 12 illustrates the orthogonal relations. The 2D symbolic projections are then

U : C1 < C2 i

V : C2 < C1 i

The following rule is now applied

if U : r1 < r2 p and V : r2 < r1 p

then

** facts ** (south r p) (west r p)

** verbalization ** "The object <p> is partly surrounded by the object <r> on its south and west side".

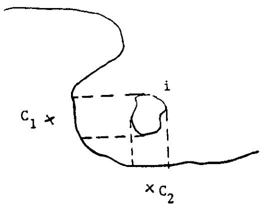


Figure 12 The orthogonal relations between an island and a coastline.

When applying this rule to the example, r will be substituted by C and p by i, i.e., (south C i) and (west C i). These facts are now stored in the fact database. The verbalization is sent to the user.

Example 2: A forest near a lake.

In Figure 13, L1, L2 and L3 illustrates the orthogonal relational objects which are part of the lake. The 2D symbolic projections are:

U : L3 < F L1 < L2

V : L3 F L2 < L1

This is matched with the following rule

if U : r1 < p r2 < r3 and V : r1 p r3 < r2

then

** facts ** (west r p)(north r p)(east r p)

** verbalization ** "The object <p> is partly surrounded by the object <r> on its west, north and east side".

In this example, we will substitute p by the Forest f and r by the lake L.

By successively applying rules that correspond to each one of the basic orthogonal relation types, it is possible to identify all object-to-object relation. For example, the fact identified in Example 2 is (west L F), (north L F) and (east L F).

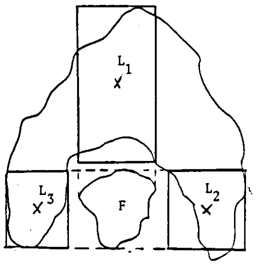


Figure 13 The orthogonal relations between a forest F and a lake L.

7. Discussion

As mentioned in Section 1, an image information system should support both simple query processing and knowledge-based complex query processing. Complex query processing may involve the generation of objects using set-theoretic operators described in Section 3. For example, we can retrieve all type-1 and type-2 objects within a specified area, and generate the union (or intersection) of these objects. As illustrated in Figure 14, the icon-oriented user interface is well-adapted to this task. By searching the knowledge-base and matching against the symbolic projections, we can answer complex queries such as:

"find all objects to the south of X"
"find all spatial relations between objects X and Y"

A pictorial example of such queries using an icon-oriented approach will allow the user to have a friendly and intuitively meaningful way for specifying queries. The pictorial query can then be transformed into symbolic projections and used to retrieve matching pictures in the database.

Generally speaking, expert systems are not particularly good at handling spatial data [WATERMAN86]. Symbolic reasoning is generally not possible because this type of data requires large amount of memory to keep track of the various spatial relations. Moreover, this process is normally slow when conventional methods are used. Orthogonal relations represented as chains of 2D symbolic projections constitute a basis for efficient use of spatial knowledge. For example, another potential application is path finding in a map. The symbolic projections can be used by a planning expert system to generate an approximate route. The approximate route can then be algorithmically refined to a definite path using previously developed algorithms [LOZANO83, WONG86]. The proposed pictorial data structure therefore can be very useful in designing expert systems for spatial reasoning, as well as knowledge-based image information systems.

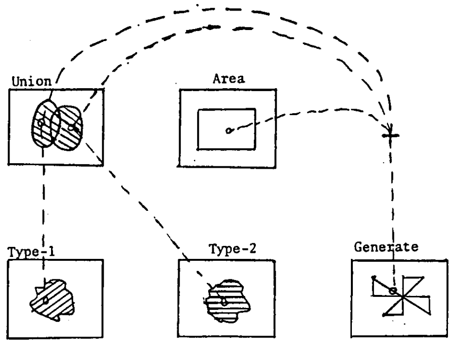


Figure 14 Icon-oriented user interface for image manipulation

References:

[CHANG86] S. K. Chang et. al., "Iconic Indexing by 2D Strings", Proceedings of IEEE Workshop on Visual Languages, Dallas, Texas, June 25-27, 1986.

[CHOCK84] M. Chock, A. F. Cordonas and A. Klinger, "Database Structure and Manipulation Capabilities of the Picture Database Management System (PICDMS)", IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 6, No. 4, July 1984, 484-492.

[JUNGERT85] E. Jungert et. al., "Vega - A Geographical Information System", Proc. of the First Scandinavian Research Conference on Geographical Information Systems, Linkoping, Sweden, June 13-14, 1985.

[JUNGERT86] E. Jungert, "Run Length Code as an Object-Oriented Spatial Data Structure", Proceedings of IEEE Workshop on Languages for Automation, Singapore, August 27-29, 1986.

[LOZANO83] T. Lozano-Perez, "Spatial Planning: a Configuration Space Approach", IEEE Trans. on Computers, Vol. C-32, No. 2, Feb. 1983.

[SAMMET84] H. Sammet et. al., "Processing Geographic Data with Quadtrees", Seventh International Conference on Pattern Recognition, Montreal, Canada 1984.

[WATERMAN86] D. A. Waterman, A Guide to Expert Systems, Addison Wesley, Readings, Mass., 1986.

[WONG86] E. K. Wong and K. S. Fu, "A Hierarchical Orthogonal Space Approach to Three-Dimensional Path Planning", Trans. on Robotics and Automation, Vol. RA-2, No. 1, March 1986, 43-53.

Acknowledgements:

This research was supported in part by NSF Grant DMC-8510804 and NRL Contract N00014-86-C-2038 of U.S.A., and by FFV elektronik AB of Sweden.

*Note: Dr. Erland Jungert is with FFV Elektronik AB, Linkoping, Sweden.

DOCUMENT IMAGE UNDERSTANDING

Sargur N. Srihari

Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260

ABSTRACT

A digital document image is an optically scanned and digitized representation of a printed page that consists of blocks of text, line drawings, half-tone pictures, and icons. Document image understanding is a goal-oriented problem involving detecting and interpreting different blocks and coordinating the interpretations to achieve an end result. We examine several solutions to subproblems in document understanding tasks. The subproblems range from pixel processing issues to symbolic linguistic reasoning to global control.

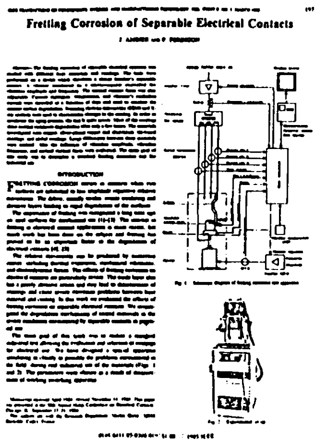
1. INTRODUCTION

A *document image* is a visual representation of a printed page such as a journal article page, a magazine cover, a newspaper page, etc. Typically, it consists of blocks of text, i.e., letters, words, and sentences that are interspersed with half tone pictures, line drawings, and symbolic icons. A digital document image is a two-dimensional array representation of a document image obtained by optically scanning and raster digitizing a hard copy document. It may also be an electronic version that was created in that form, say, for a bit-mapped screen or laser printer. An example of a document image is shown in Figure 1.

Document image *understanding* (or simply, document understanding) is a goal oriented problem that may involve interpreting photographs (which is a *vision* problem), interpreting text (which may involve *natural language understanding*), interpreting line drawings such as graphs, in such a way that the interactions of the different components is accounted for. Some examples of document understanding tasks are: (i) integrate the pictorial content of a newspaper photograph with accompanying narrative title, (ii) determine the paste-on destination address on a colorful magazine cover: it is necessary to locate and orient the most plausible textual block, to read it and verify that it has the correct syntax, and (iii) make inferences from an annotated *x-y* plot.

Document understanding implies the correct recognition of each of its constituent components. The recognition has to be both in terms of high level components such as columns and paragraphs, as well as in terms of low level components such as words and lines in the case of text; curves, rectangles, and solids in case of graphics; and image regions in case of half-tone pictures.

Document image *analysis* is the task of deriving a high level representation of the contents of a document image. The spatial structure of a document can be represented in several ways. Two representations of the image of Figure 1 are shown in Figure 2. The first is an analogical representation that gives the spatial extent of blocks including paragraphs and the second is a propositional representation in the form of a partial semantic network whose nodes represent document entities and arcs represent relationships. Each of them assumes blocks as units where each block can be associated with details like size, shape, number of black pixels and texture information. This information can then be utilized to discriminate between blocks. Methods for deriving the blocks can take advantage of the fact that the structural elements of a document, i.e., columns, paragraphs, titles, figures, lines of text, symbols, etc., are generally laid out in rectangular blocks aligned parallel to the horizontal and vertical axes of the page.



Fretting Corrosion of Separable Electrical Contacts

Figure 1. An example of a document image.

Document Understanding Components

Document understanding involves the use of several types of knowledge including *visual, spatial,* and *linguistic*. In determining objects from the background, say, by thresholding or by edge detection and grouping, the parameters necessary to perform such operations constitute visual knowledge. Labeling regions involves the use of spatial knowledge, e.g., the layout of a typical document. Determining the font and identity of characters also involves spatial knowledge, i.e., the structure of textual symbols, words, and the distinguishing characteristics of each font. Reading words in blurred or poorly registered text, is a process that involves spatial as well as linguistic knowledge, e.g., lexicon of acceptable words. Determining the role of a block of text, e.g., "is this a title?," is a process requiring spatial, syntactic as well as semantic knowledge. Considerable interaction among different types of knowledge may be necessary. For instance, assigning a role to a textual region may require not only knowledge of spatial layout, but also an analysis of its textual syntax and semantics, and an interpretation of neighboring pictorial and iconic regions.

The various processes necessary for document analysis are shown in Fig. 3. The task has been divided into three phases. Phase 1 consists of two steps: optical scanning and digitization and binarization. Phase 2 consists of block segmentation and labeling. The result of phase 2 is a set of regions (possibly overlapping) labeled as text, graphics, half-tone, etc. Phase 3 consists of several parallel operations for processing text, graphics, and half-tone images. Document understanding involves feedback paths not shown in Fig. 3.

CH2345-7/86/0000/0087$01.00 © 1986 IEEE

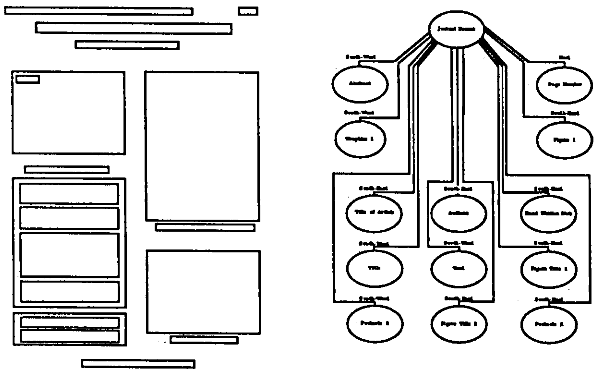


Figure 2. Document image representations.

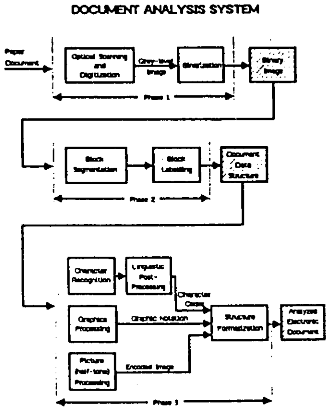


Figure 3. Document Understanding Components.

2. SCANNING AND BINARIZATION

An optically scanned document image is usually an integer (gray-valued) array; it is obtained by a process of spatial sampling and simultaneous conversion of light photons to electric signals. A high-resolution *bit map* is sufficient to capture shades of gray in the eye of the human perceiver (such images are called *half-tone*). An optically scanned document image can be converted into a bit map by a *global* thresholding operation: pixel values below the threshold are deemed to be black (value 1) and those above deemed white (value 0). The threshold itself can either be predetermined or calculated from the image histogram, e.g., the valley of a bimodal histogram.

A global threshold, i.e., one that applies to the entire image, may sometimes be impossible to obtain due to varying background color. One solution is *adaptive* thresholding which is to compare the gray value of the pixel to the average of the gray values in some neighborhood about the pixel. If the pixel is significantly darker than the neighboring pixels, it is called black; otherwise it is white. One such adaptive thresholding algorithm [14] that is appropriate for a printed document considers a 9 x 9 region around a pixel (x, y). The central 3 x 3 region including (x, y) is called A_1 and four 3 x 3 regions 8-connected to A_1 are jointly called A_2. The average pixel value of A_1 is compared to the weighted average pixel value of A_2. The idea in choosing the regions in this manner is that of a black pixel that is part of a line of text should only be compared to the white spaces above and below the line.

Color

A document with significant colored regions is usually scanned as three gray level images with red (R), green (G), and blue (B) filters. A colored image can be converted into a single bit map by approximate methods. Color thresholding is useful to extract regions of a particular color. Thus if the objective were to determine the address label which is known to be white then one can focus only on the white regions and perform further feature tests (e.g., rectangularity, presence of text) [21].

Resolution

Scan resolution is of importance in document understanding. Scanning resolution for text and line drawings can be decided as follows. The width of a *typical* character stroke is about 0.2mm (0.008 inch), with some of the widest strokes up to about 1mm; a 10-point character measures about 0.5mm (0.014 inch) between ascender and descender lines. A sampling rate of 240ppi corresponds to about 0.1mm/pixel which guarantees that at least one pixel will fall totally within the stroke; this resolution corresponds to 33 pixels between the ascender and descender lines. In the case of engineering drawings the minimum line width is approximately 0.3mm. Thus the minimum resolution needed is approximately 100 ppi.

3. BLOCK SEGMENTATION

Approaches for segmenting document image components may be roughly divided into techniques that are top-down or bottom-up. Top-down techniques divide the document into major regions which are further divided into sub- regions, etc. They often use knowledge about document structure, e.g., that the layout is in the form of rectangular blocks. Bottom-up methods progressively refine the data by layered grouping operations. Although no practical system takes a pure approach, every system can be identified as being predominantly aligned with one of the two philosophies.

3.1. Top-Down Approaches

We will describe three top-down approaches: smearing, projection profile cuts, and the Hough transform.

Smearing

The smearing technique [24] involves scanning documents as black/white images A run-length smearing algorithm (RLSA) then operates on the image under which any two black pixels (1's) which are less than a certain threshold t apart are merged into a continuous stream of dark pixels. White pixels (0's) are left unchanged. For example, if the input sequence was 0001100000110010001 and the value of t was 3 then the result of the RLSA on this sequence would be

11111000001111100001.

The RLSA is first applied row-by-row and then column-by-column, yielding two distinct bit maps. The two results are then combined by applying a logical AND to each pixel location. The resulting RLSA image contains a *smear* wherever printed material appears on the original image. The thresholds t_x and t_y in the two directions need not be the same. The segmentation is expected to yield blocks each of which should contain only one type of data (text, graphics, half-tone, etc.).

Projection Profile Cuts

Based on the observation that printed pages are primarily made up of rectangular blocks, a page can be recursively cut into rectangular blocks. Thus the document is represented in the form of a tree of nested rectangular blocks. The application of cuts is based on the configuration of the pixels. A "local" peak detector is applied to horizontal and vertical "profiles" to detect local peaks (corresponding to thick black or white gaps) at which the cuts are placed; it is local in that the width is determined by the nesting level of the recursion, e.g., gaps between paragraphs are thicker than those between lines [11, 25].

A complete hierarchical recursive tiling of a page can be represented as a tree. An *X-Y tree,* is obtained by using horizontal or vertical partitions at alternating levels of the tree. The root represents the entire page and can be considered either a single horizontal or vertical partition. The succeeding nodes, and finally the leaves are obtained by a process of alternating horizontal and vertical cuts. The number of horizontal or vertical cuts placed at each level is variable, so the resulting tree is not binary. A node that represents a horizontal (vertical) partition has a set of descendents that represent a complete vertical (horizontal) tessellation of the parent node.

Hough Transform

A document typically contains several straight lines. Forms and tables contain solid lines. Line drawings, e.g., block diagrams, predominantly have straight lines. Columns of text are separated by straight rivers of white space. Text usually consists of several parallel textured thick lines.

The Hough transform is a technique for detecting parametrically representable forms, e.g., straight lines, in noisy binary images. It involves transforming each black pixel (x, y) in the original image into a curve in the parametric space. In the case of detecting straight lines the transformation used is $K = x \cos T + y \sin T$. For each point (x_i, y_i), T is varied from 0 to J to yield the corresponding curve in the (K, T) space. An array of accumulators is set up by quantizing values of K and T. The accumulators corresponding to values of (K, T) yielded by the transformation are incremented by one. It can easily be shown that if points are collinear along the line specified by parameters (K_0, T_0), then each such point (x_i, y_i) will necessarily increment accumulator (K_0, T_0). Thus each value in the accumulator array corresponds to the strength of evidence for a straight line with the corresponding parameters. For a 512 x 512 image, K values extend from –368 to +368 (This value is arrived at by assuming the origin of the (K, T) space to be at the center of the 512 x 512 image and hence the maximum value of K is 256 x 2).

The accumulator array resulting from applying the Hough transform to a portion of the document image of Figure 1 is shown in Figure 4. The portion corresponds to the "abstract" in the left column. In general the array can be checked for the particular orientation which has the maximum number of transitions to and from a minimum value. Transitions corresponding to text are usually regular and uniform in width and thus are easy to identify. In the case of text containing both upper and lower case, they register a maximum value at the center line of the characters and slightly lower values corresponding to the ascender and descender lines. Textual lines have another property that there are "white" lines perpendicular to the beginning and ending columns and this information can be used to confirm line orientations. First, the number of vertical columns in a page needs to be determined because a multi-column page may not have any (K, T) orientation with low counts. Subsequently, the spacings between the text lines can be identified, on the basis of the average width of significant (K, T) values for a particular T.

The distance histograms along the line and its perpendicular orientation can be used to get the inter-word and inter-line distances. By incorporating these, the thresholds for the run-length smearing algorithms can be derived and big blocks of data can be grouped together [16].

3.2. Bottom-Up Approaches

Bottom-up analysis is based on successive steps of refinement of the input image. The image is first processed to determine the individual connected components. At the lowest level of analysis, there would be individual characters and other large figures. In the case of text the characters are merged into words, words are merged into lines, lines into paragraphs and paragraphs into even larger blocks, if such a merging is possible. In a bottom-up technique it is usually necessary to determine whether a connected component is a part of: text, line drawing, or threshold region of a half-tone picture. Possible features for performing this classification are: size, branching structure, topology, and shape measures.

Size

A commonly used method is to determine the size of a component, by counting the number of pixels, and consider it not to be a character if it lies within a range. A disadvantage of this method is that the size of characters has to be known beforehand, which is not always possible. However, size is a useful first feature.

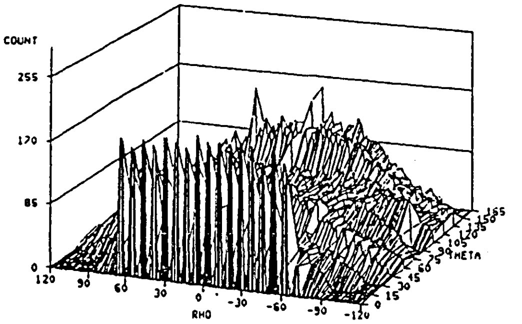


Figure 4. The Hough Transform accumulator array. Text lines appear as a series of peaks in the 0° column.

Branching Structure

The geometrical complexity of an object can be measured in terms of the number of linear components (branches) that the figure is composed of. If the complexity is high then the object is probably part of a line drawing, half- tone picture, or form, i.e., it contains too many branching structures to be a printed character. An algorithm can be designed to determine geometrical complexity of several components while the image is scanned raster fashion. This is done by keeping track of the several components in each line and their connectivity with respect to components in the previous line. A description of each connected component consisting of, say, its size, location, horizontal and vertical extents is maintained. Clusters of structures of similar type can then be identified. An example of this clustering approach is shown in Figure 5. A pass is made on the binary data to determine all connected components, which are individual characters. On the basis of proximity, size and other features the characters are merged into lines [2].

Topology

The topological properties of a component are useful in determining if it is a character. The Euler number of a binary image is defined as the number of components minus the number of holes. For a component which is a character the Euler number is one (1), zero (0), or negative one (–1). A fast algorithm can be designed for computing the Euler number of a binary image. More generally, the adjacency tree of a binary image contains all topological information (Figure 6).

Control for Bottom-Up Grouping

The order of application of different operators can be based on their complexity, e.g., most characters can be quickly determined by their component size. However if characters touch a line, as often in the case of line drawing graphics, the characters have to be segmented from lines. One technique for segmenting characters from line structures is to determine the high neighborhood line density (NLD) areas in the line structures [8]. Such areas are candidates.

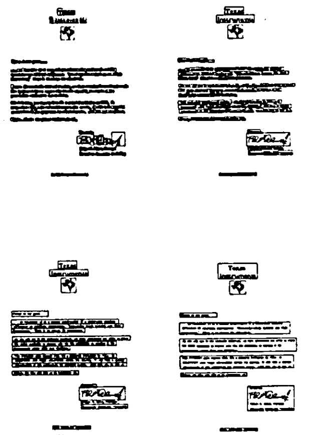


Figure 5. Bottom-up grouping technique. Character blocks are merged into words, words into lines, etc. using a clustering technique.

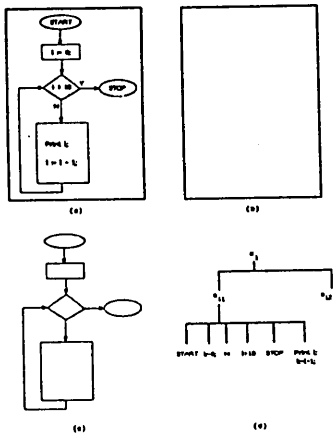


Figure 6. Adjacency tree: (a) input graphics, (b) outermost component a_1, (c) inner component a_{11}, and (d) complete adjacency tree.

The application of different operators for bottom-up grouping can be coordinated by a production system. Examples of specific rules in such a system are:

if connected components [*c*]'s size [*s*] is larger than *t*,
then the component is graphics with likelihood L_1,
if neighborhood line density (NLD) on graphics is high,
then the high NLD area is a character area with likelihood L_2.

A sketch of control flow follows:

1. Size, position, and direction data of every connected component is collected.
2. The components are classified according to size into graphics, characters, and noise and their likelihoods are determined.
3. Character components are grouped together as character areas using vector distance rules.
4. High NLD areas in graphics components are separated out as characters touching lines and their likelihoods are determined.
5. If another character exists near the touching character area its likelihood value is increased.
6. The cut-off part in a faded line is repaired.

4. LABELING

Blocks determined by a segmentation process usually need to be labeled in a subsequent step. There are several techniques for classifying a given block into one of a small set of predetermined document categories.

Statistical Approach

Simultaneously with component coloring, the following measurements are taken: total number of black pixels in the segmented block, minimum *x-y* coordinates of a block and its *x-y* lengths, total number of black pixels in the original image for the block, and number of horizontal white-black transitions in the original image block.

The next step is to classify each block according to content. This is done by computing several features for each block from the above measurements and then using a linear pattern classifier. The features computed are: the height of a block, its eccentricity, the ratio of the number of black pixels to the area of the surrounding rectangle, and the mean horizontal length of the black runs in the original data from the block.

A block is determined to be text if it is a textured stripe of mean height H_m and mean length of black run R_m. The distribution of values in the *R-H* plane derived from sample documents are observed to determine the discriminant function. Low *R* and *H* values represent regions containing text. To determine the threshold values of *R* and *H* that define the text region in the R-H plane, an adaptive method is used. This method estimates R_m, H_m and the standard deviation of *R* and *H*. These values are then used to classify the various blocks by using the following pattern classification scheme that assumes linear separability:

- **Text:**
  if $R < C_{21} \times R_m$ and $H < C_{22} \times H_m$,
- **Horizontal solid black lines:**
  if $R > C_{21} \times R_m$ and $H < C_{22} \times H_m$,
- **Graphic and Half-tone images:**
  if $E > 1 / C_{23}$ and $H > C_{22} \times H_m$, and
- **Vertical solid black lines:**
  if $E < 1 / C_{23}$ and $H > C_{22} \times H_m$.

A Distance Mapping Shape Measure

A shape measure for determining whether a component is a character, line drawing, or thresholded gray-level image is as follows [22]. Given a connected component *S* first a distance *d* is computed for each pixel of *S* as follows. Distance *d* is a function of the two Cartesian coordinates *x*, *y* ($(x, y) \in S$) and an angle φ measured from the *x* axis (0° ≤ φ 180°). It is defined as the length of a line segment B_1B_2 that passes through the point (*x*, *y*) of *S* and makes an angle φ with the horizontal. The two opposite border points B_1, B_2 of *S* are such that the line segment B_1B_2 is entirely inside *S*. Three distance mappings are:

$D_{min}(x, y) = \min_{\varphi}[d(x, y, \varphi)]$, or the length of the shortest chord through (*x*, *y*),

$D_{max}(x, y) = \max_{\varphi}[d(x, y, \varphi)]$, or the length of the longest chord through (*x*, *y*), and $D_{ecc}(x, y) = D_{max}(x, y) / D_{min}(x, y)$.

These mappings reflect geometrical properties of objects within binary images. By averaging D_{min} within *S* the mean minimum border to border distance dmin can be calculated. This provides a fairly good estimate for the mean line thickness of line shaped patterns. To discriminate between line shaped patterns and more compact ones, the average of D_{ecc}, d_{ecc} is a useful measure. Similarly, dmax reflects the mean maximum border to border distance.

Further shape factors are desirable: d_{min} can be used to derive a shape factor which equals one for the most compact pattern, an ideal circle, and is greater than one in all other cases: $f1 = A / d^2_{min}$ with $C_1 = 16 / 9\pi$ in continuous space (or number of pixels in disscrete space). This global measure reflects the "line shapeness" of pattern, i.e., their compactness independent of A more specific characterization can be obtained by means of additional features, e.g., $f_2 = C_2 \times A / d^{-2}_{max}$ with $C_2 = 4 / \pi$ in continuous space (1.2.388 in discrete space) which is invariant with respect to the size of the pattern. Unlike f_1, f_2 is capable of discriminating a straight line from an L-shaped line having about an equal number of pixels and about equal line-thickness.

Handwriting versus Print Discrimination

Given a region that contains either printing or cursive script, a method can be devised to determine its nature by observing regularities of white-to-black and black-to-white transitions. Figure 7 illustrates.

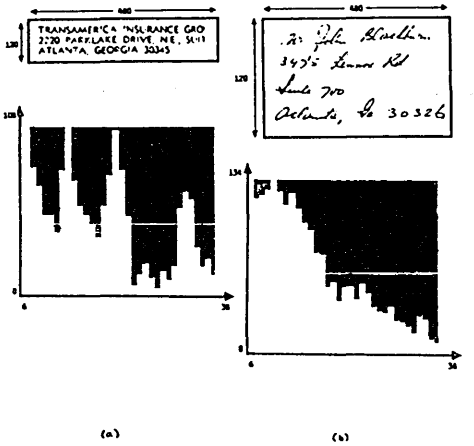


Figure 7. Printed text can be distinguished from cursive script by determining regularities: (a) printed text and histogram of black-white transitions in vertical direction, (b) cursive script and corresponding histogram.

The method involves computing several filter histograms of the test region. One example of a filter is a white-to-black transition separated from another white-to-black transition by n units along the horizontal direction. The histogram is obtained by counting the number of such transitions that occur for each value of n. Different histograms are obtained by considering a white-to-black transition separated from a black-to-white transition in the horizontal/vertical directions. The histograms for printed text tend to have several regular peaks while handwriting tends not to have such peaks. Thus the test becomes one of classifying the histogram as regular/irregular. The classification can be done either by treating the histogram itself as a feature vector or by extracting an appropriate feature vector from the histogram.

Rule-Based Approach

A method of associating labels like line-of-characters, paragraph, title, author, abstract, text columns or text lines, column, photo, etc., to the blocks on the basis of their extent, absolute and relative position is then needed. This can be done by a rule-based expert system that uses a knowledge-base comprising layout and composition rules for specific classes of documents [11]. Using physical (syntax) information, the rule-based system proceeds on its labeling of the blocks (referred to as semantics).. These rules can determine line-of-characters, paragraphs, columns, photos etc. using concepts of width, aspect ratio, number of cuts, number of siblings at a particular stage and other such related information. Rules are of the following type:

- line of characters: sequence of adjacent character blocks of some height, separated by character-segmenting rule,
- paragraph: sequence of blocks of line-of-character of same length, separated by line finding rule,
- column: large blocks with approximately equal frequency of "1" and "0" pixels, and
- photo: sequence of paragraph blocks of same width, separated by paragraph-cutting rule.

For a complete consistent set of rules, the labeling process has worst case complexity equal to the product of the number of blocks and the number of labels.

5. GRAPHICS PROCESSING

Line drawings and tables are commonly encountered in documents. Their analysis involves a raster to vector conversion. The aim of a raster to vector process is to convert a binary pixel representation of line-work into a connected set of segments and nodes. A segment is typically a primitive such as a straight line or a parametric curve. Straight lines in vector form are specified by their start positions, extent, orientation, line width, pattern, etc. There have been several different approaches to vectorization, including: pixel- based thinning, run-length based vectorization, and contour-based axis computation.

Pixel-based thinning algorithms require multiple passes through the data set. During each pass object pixels are deleted based on local neighborhood criteria. Such methods gradually thin thereby thinning the image down to unit width (Figure 8). A single pass algorithm could be obtained by using a set of local 3 x 3 operators and applying them in a quasi-parallel manner to generate a marked skeleton; quasi-parallel application means dividing the image into four disjoint sub-images and applying the operators to each sub-image in turn [5]. These techniques are slow as they operate on individual pixels rather than on groups of pixels. Also a lot of time is wasted in looking at white pixels more than once; only about ten percent of the pixels on any page of text are black and hence this wastage is quite considerable.

Run-length based vectorization [15] differs from pixel-based thinning in that the image does not go through multiple passes to get at a thinned image of unit width. Instead, vectors are drawn with certain approximations resulting in this. The method aims at this by tracking the black segments along scan lines and using aggregates of such segments to operate upon. The algorithm achieves the objective by converting the input run length encoded image into a graph and analyzing the graph. The graph also has information which could be used to provide many more useful features for character recognition. It is found that this method works better for lines perpendicular rather than parallel to scan lines. The complexity of pixel based thinning is of the order of the number pixels. The complexity of run-length based thinning is approximately of the order of the number of run length code segments.

Some problems that need to be specially examined are handling of dotted lines, determining end points of lines, effectiveness of thinning, determination of vector intersection and curve fitting versus piecewise linear approximation. Issues in understanding of graphics are discussed in [3, 4, 10].

6. TEXT RECOGNITION

A text recognition or reading technique takes an image of text and maps it into an ordinal machine representation, e.g., ASCII. Methods of reading text can be divided into two related but distinct categories: *isolated character recognition* and *character recognition in context*. When the types of font are known *a priori*, or when few fonts are encountered, a highly reliable isolated character recognition technique can be designed.

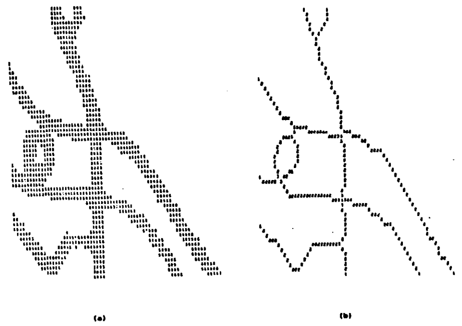


Figure 8. Example of Thinning Line Drawings.

Isolated character recognition involves extracting a set ot predetermined features from the character image. The features used are chosen for their ability to discriminate between classes. For instance, character height in millimeters is useful for distinguishing between lower case letters *with* ascenders, (e.g., b, d, h, k, l, t) and letters without, (e.g., a, c, e, n, o, r, s, u, v, w, x, y, z). Thus the character is regarded as a point x in a multi-dimensional feature space. Classification is accomplished by using discriminant functions which effectively partitions the feature space; typically each class i has an associated discriminant function d_i and x is assigned to the class that maximizes $d_i(x)$. The discriminant functions d_i, or their parameters, are determined statistically from a large set of samples during a training phase. Character recognition techniques quite often use only binary-valued features. since they lead to simpler feature representation and discriminants

6.1. Isolated Character Recognition

Typical of such a method is one [17] that treats a binary character image on a 16 x 16 grid as a 256-element binary feature vector x. The classifier uses second order polynomial discriminant functions derived using least mean squared error considerations. The form of the discriminant function is: $d_i = a_{0i} + a_{1i}y_1 + a_{2i}y_2 + \dots + a_{mi}y_m$, where y_i are either components of x or products of pairs of components. Since the number of such terms is large, (i.e., $256 + {}^{256}C_2$), only a subset is used. Approximately $m = 1200$ is determined where the pairs of pixels are heuristically chosen. In the 16 x 16 raster field, more pairs are chosen at the center than in the periphery. The set of coefficients a_{ji} are represented as a matrix A of size $k \times m$. Thus the computation of the k discriminant functions can be expressed as the matrix multiplication: $d[k \times 1] = A[k \times m] \,.\, y[m \times 1]$. The index of the member of d which is closest to unity is the class to which the input is assigned.

6.2. Character Recognition in Context

In most printed documents characters hardly ever appear in isolation. Several approaches to recognizing characters in context are known [19]. Contextual information is usually in the form of a lexicon of acceptable words. It can also be in the form of *n-grams*, i.e., legal letter types, or letter transition probabilities.

One approach is to recognize characters of words and then post process (correct) any errors by using a lexicon of words and a distance measure between words. Another approach is to use lexical information to limit the number of possibilities in recognizing individual characters to find the nearest word. This is done by weighting the choices for a given character by (i) its occurrence in the lexicon and (ii) its frequency of occurrence in the text. Thus the unlikely choices are eliminated and the resultant is guaranteed to be in the lexicon. Integrating contextual information into the character recognition process leads to better performance than the two step approach of character recognition followed by error correction [20]. The third approach is to extract features from the entire word and attempt to classify it using a lexicon organized by word feature. A simple set of features are used in a first level analysis to select a neighborhood of words and a more detailed analysis discriminates between a small subset of character classes [6].

6.3. Type Font Analysis

A document analysis system needs to distinguish between text of different fonts. In the terminology of typography, a *font* is a collection of upper and lower case letters and special characters of one particular typeface, style, and size. The tvpeface determines the overall design of one character shapes. The style refers to the average stroke width of the characters, boldface versus lightface (normal), and the posture of the body, italic versus roman.

The task of character recognition is considerably simplified if the font is determined ahead of recognition. Variability of characters is due to two sources: the type font and noise. Given the English alphabet in several fonts, the style of any given letter, say "B," is different in each font. However within any one type font certain aspects of the style of the "B" are identical to the stylistic aspects of the other characters in that font, e.g., the lower left sides of "B" and "D." By providing a framework within which this repetition of stylistic aspects in a type font can be expressed explicitly, it can be used subsequently to augment the recognition algorithm, e.g., knowing the font, a decision procedure for that font can be applied.

Determining the stylistic consistency of a given font involves finding elements such as serifs and then finding their relationships to strokes to which they are joined. A set of rules for type font analysis considering only machine printed fonts taken from the Roman and sans-serif families is given in [1].

6.4. Linguistic Analysis

The role of language in the recognition of images of text has been little explored beyond the word level. Here we describe a linguistic approach to recognize whether a given two dimensional block of text is an address. If it is an address, a parse tree is obtained. The method exploits spatial information in its analysis, i.e., the space between words or lines is important.

In determining the destination address on a mail piece image with several two-dimensional blocks of text, it is necessary to syntactically analyze the recognized characters in a candidate block by using spatial information such as character, word and line spacing [21]. A system has been developed to perform this syntax analysis by attempting to parse (label) subelements of the block of text and produce a parse tree representing the labeling of the constituents [13]. This labeling identifies words corresponding to the city, state, zip code, and street name, among others. Several significant changes have been made to a standard Augmented Transition Network (ATN) parser [18] including word abbreviations, spatial feature analysis, and heuristic parsing.

This module takes as input a sequence of lines of sentences (subsequently called a *sentence block*). This input, in the form of ACSII characters, is then preprocessed to record the number of horizontal blocks before each word and record this spatial information as lexical features of the input word. A similar preprocessing operation is performed to record the vertical spacing between lines in the block but with the spatial separation recorded as a special *word* inserted directly between the line of the input and later handled as another word in the input stream. This provides the parser with spatial information needed to check if a word or line is "too far away" from the remaining words to be considered.

This preprocessed sentence block will be *accepted* by the parser if and only if the sentence block has the syntax of an address block. If the input sentence block is acceptable in terms of the grammar and lexicon, then the structure (parse tree) of the region will be passed to the generalized rule-based region labeler for use in the determination of the correct sending address as opposed to the return address.

The address recognition module incorporates the four assumptions described below:

1. When a block in initially segmented and passed to this address recognition unit, extraneous printed material may be present in the block. This is dealt with by adding heuristic arcs (described below) to the grammar which will allow crude matching of the beginning of each line and result in the skipping of the remainder of the text line suing the generalized TO arc [18].
2. If the text block contains an ambiguous address, then the output will be a parse tree representing all possible interpretations of the block.
3. This unit will not perform context matching between the city, state, and street name with the five of nine digit zip code. Context checking is also not performed between other components of the address since it is not a necessary condition for a text block to be an address block.

Parser Modifications

The major changes to the standard parser include the capturing of spatial features such as end of line and word spacing, and the addition of abbreviation handling.

Spatial Properties: A major parser modification was the addition of a spatial acquisition mode for capturing the number of spaces before each word and adds this spatial information to each lexical form of the current input word. This information includes the number of blank spaces prior to a word, the distance to the left, top, and right margins and also the length of the word. Below is a partial input to the parser after the initial conversion of physical spaces in the input to lexical features associated with the words.

```
(buffalo(((root . buffalo)
(ctgy . city)
      (parts . 1)
      (head-space 3)
      (left-margin-space 3)
      (top-margin space 0)
      (word-length 7)
      (right-margin-space 29)
      (line-length 32)))

...)
```

If vertical spacing is present in the input, then the block will be linearized with the insertion of a new line *word* of the form: @*nnnn* where @ represent the start of a new line, and the *nnnn* represents the number of units (lines) in the vertical spacing.

Abbreviations: Since many of the words in an address block are abbreviations such as Ave, St, NY, PA, S, N, W, a method was developed to associate these abbreviations with their expanded forms. To facilitate the lexical look-up of abbreviations and avoid the repetition of lexical words with similar meaning, a new property was used to equate a word with a series of other words.

A grammar corresponding to the bottom two lines of a standard address has includes the structures of the PO box, street, city, and state fields as well as the five (or nine) digit zip code. This grammar is complicated by the possible presence of punctuation between words and the spatial information mentioned above. Sometimes the punctuation and spatial information is important. For instance, if a comma is present, then it is a separator between the city and state name. Also, if a new line word is present, then it is most likely the termination of a multiple word phrase such as a city, state, or street name.

The grammar, in addition to containing arcs representing the bottom two lines of a standard address also contains heuristic rules for detecting and skipping other possible input lines. That is, if the first word on a line is a personal title (Dr., Mr., Ms., Miss, or Mrs.), or a personal first name, then the remainder of the line would be ignored. Another skipping rule is for the detection of a standard presort codes usually present on the top line of an address block.

For a sentence block to be parsed as an address region, several constituents must be identified. For instance, if more than one line is present in the address region, then the city/state/sip line must start on a new line. Another necessary element for a block to be parsed as an address block is the presence of a state name followed by a zip code (either on the same input line of on the following input line).

The grammar, initially, makes one pass through the input, attempting to locate a possible state name. If a state name is not found, then the parse will fail. If, however, a state name is found, then the state name and line number are recorded for later processing and a second pass is made through the input utilizing a more robust grammar for detection of other constituents. This preliminary detection of the state name provides an efficient means for detecting a necessary component of the address and reduces the computational cost of parsing the entire address region.

Lexicon

The lexicon contains generally five different types of entries corresponding to the street, city, and state names, street keywords (such as avenue, street, circle, parkway), and other keywords most often present in the address region of mail pieces such as personal names and titles. As was mentioned previously, the lexicon need not contain all possible street and city names but parses of an input sentence blocks containing these "missing" cities of street names would result in weak parses.

The grammar entails 106 states, 178 arcs, and a small text lexicon which contains 188 unique words with entries for all 50 states (and their abbreviations), 32 cities, 20 street names, and 20 street keywords. To judge the effectiveness of the keyword approach in guessing a city name, the Directory of Post Offices (DOPO) data base was used. A total of 25,630 on word city names were encountered with over thirty percent (7,843) of these city names ended in the above mentioned letters. For two word city names. 7.437 cities were encountered with over seventy-four percent (5,522) of these having first or last words as those mentioned above for two word city names. This shows that these patterns are a good indication of a city name, however, testing was also done to compare the number of English words (advertising material) that would be wrongly mapped into a city name based on these rules. Using the Brown Corpus containing 43,264 English words, a total of only 79 (0.18%) English words matched the one word patterns. A comparison of the two word pattern was not possible with the lack of contextual information in the corpus.

The parses of an input sentence which has been accepted using the grammar and lexicon is shown below, where the preprocessed input sentences are also stored in the parse tree as the first element using the form mentioned above.

```
(address(input-sentence
    (|1241| main @2 n |.| java n |.| y |.|152102| |;|4128|)
    (line starts (00)))
    (spatial (word (0 W4 1 W4 0 W1 0 W1 1 W4 1 W1 0 W1
              0 W1 0 W1 1 W5 0 W1 0 W4))
              (line (L9 2 L23))))
    (prepass (states((state (state-name (new york))
            (puncts (|.| |.|))))
    (level (01)))
    ((street-line (street-number |1241|) (direction nil)
                  (street-name main) (street-keyword nil)
                  (apartment nil) (puncts nil) (street-guess nil))
     (bottom-line (city (city name (north java))
                  (parts 2 (puncts (|.| nil)) (guess t))
                  (state (state-name (new york)) (puncts (|.| |.|)))
                  (zip (zip-location after-state)
                       (ninezip |151024128|) (fivezip nil))))).
```

7. PICTURE PROCESSING

A document image understanding system needs to integrate the information contained in photographs with the information contained in other fields such as with the text and line drawings. Often an understanding of the photographs is critical to understanding the narrative. Deriving descriptions of the contents of photographs is the subject of many computer vision projects today.

8. CONTROL STRUCTURES

The coordination of several processes is of central importance in a document understanding system. The control structure allows the application of the appropriate process. For instance, there may be more than one way of binarization. The method to be applied is determined by the control structure.

Several efforts towards designing control structures for document understanding are described [8, 9, 12, 23]. A method based on production systems and blackboard communication first proposed in the context of speech understanding holds promise for document understanding.

The mechanism described in [8] is based on a production system. When the production system needs data the mechanism causes the appropriate module to start collecting data. In this mechanism each connected component is treated as a data unit. An array, which is a specialized *blackboard* is used for the read/write area in both the production system and program modules. The blackboard is made up from three planes of identically sized arrays. Data, indicators, and program modules names and arguments are respectively stored in these planes. Data concerning positions, size, likelihood, and attributes for connect components are stored in the first data plane. The indicator plane indicates whether or not the value in the corresponding position in the first plane has already been collected. When the production system requires some data for matching, the indicator is checked first. If the desired data has already been collected, the values in the data plane are transferred to the production system. In the case where the data has not yet been collected, the program module stored in the third plane is activated and its function is carried out. The program module function determines the data and forwards the values to the data array.

Advantages of using production systems in document understanding are:

- easy to apply either an additional technique to an object that is hard to interpret with only one technique, or a retry process having modified parameters,
- processing knowledge is expressed as rules, thus software maintenance (e.g., modifying or patching of a program) becomes easier, and
- as the processing is not carried out over the entire picture uniformly, but only in necessary segments, high efficiency is obtained.

An Expert System

A production system that is organized with different levels of production rules that perform an analysis of a document image, and interpret and classify the various regions of printed matter on the document is given in [12]. The control flow is as follows: the document is first digitized and the resulting digital image is segmented to obtain data about the various printed regions in the document. This data includes the intrinsic properties (e.g., shape, size, aspect ratio, etc.) of each of the identified regions, as well as the spatial relationships between the various identified regions in the document image. The control structure then uses the knowledge base to examine this data, and attempts to arrive at a consistent classification for each of the identified regions, or blocks. The system consists of three levels of rules.

If the data from the initial segmentation of the image is not sufficient for an unambiguous interpretation of the document image, then the system decides to obtain more data from the given image. Thus, any further image processing operations that are required are progressively invoked under the supervision of the inference engine. These operations could include further segmentation of the image, color filtering, text reading, etc.

A goal-driven (top-down) approach is used by this system, which uses a hypothesize-and-test strategy for arriving at its conclusions. Thus, the system makes hypotheses about different intermediate conclusions and chains backwards through the rules in order to test the hypotheses. In trying to satisfy a hypothesis, some other hypotheses may be generated which must first be tested before the original hypothesis can be considered to be justified. Thus, an entire set of backward-chaining processes are set up, and the system only reaches a satisfactory conclusion when all these processes have run to completion.

The Knowledge Base

The knowledge base consists of a set of rules that embody knowledge about the general characteristics about document images. These rules are expressed in terms of predicates in first order predicate logic. The rules in the knowledge base are *Knowledge rules*. These rules define the general characteristics expected of the usual components of a document image, and the usual relationships between such components in the image. The usual relationships, e.g., the title being above the author names, the abstract being above the first paragraph of text, the footnotes being at the bottom of the page, etc. are generally true of such documents. Intrinsic properties, like the block-to-white pixel ratio for half-tone figures in the image being larger than the corresponding ratio for text, are also true in general for such blocks. From such known facts about these kinds of document images, rules are constructed that can be used by the inference engine to make inferences about the various identified *blocks* on the given document image.

The Control Structure

The control structure for the expert system consists of an inference engine which uses the knowledge base to make unambiguous inferences about the classification of various blocks in a given document image. The inference engine is also rule-based, and contains two levels of rules: *Control Rules* and *Strategy Rules*. These rules regulate the analysis of the document image, and decide when a consistent interpretation of the image has been obtained. The inference engine uses a top-down approach in arriving at its solution, since the solution space is not very large, and a lot of knowledge exists (in the knowledge base) about the domain. A backward-chaining process is used by the control structure.

The rules comprising the inference engine are also coded in terms of predicates in first-order predicate logic. The control structure determines the order in which these rules are executed in order to test various conditions effectively. Control rules can be *focus-of-attention* rules or *meta-rules*. For example, at any given stage of the analysis, control rules can decide that all the relevant knowledge rules for footnotes be executed so as to test whether the given block is a footnote. Strategy rules can guide the search in a more general way, i.e., they can determine what strategy is to be followed at any given time for analyzing the image. This means that the strategy rules determine what the order of execution of the control rules will be.

Representation of Uncertainty

The system has to deal with many situations where a combination of rules, rather than a single rule, lends credence to a particular hypothesis. Thus, the success of each of these rules adds evidence towards that hypothesis. If the total evidence obtained from the successful rules is sufficiently high, then the hypothesis is assumed to be true, and the next stage in the analysis process can then be tackled with the assumption that the given hypothesis has been confirmed. To deal with such a scenario, each Knowledge Rule in the system is given a certain *confidence value* between 0 and 1. When the knowledge rules for testing the characteristics of a certain type of block are executed, the confidence values for all the rules that succeed are added up. The sum thus obtained indicates the *certainty factor* for the conclusion obtained from the control rule which invoked these knowledge rules. This certainty factor is used for the purpose of ordering the conclusions at any given stage so that the more likely conclusions can be examined in further detail *before* the less likely ones. This has the effect of making the search process more efficient, thus reducing execution time in the system.

Discussion

The wisdom of using production rules to represent knowledge has been the topic of discussion among many AI researchers. In the domain document understanding, a rule-based system is extremely elegant because unlike natural scenes, documents are very structured in character, and thus knowledge about features of documents can be very effectively formulated in terms of production rules. There are other advantages to using production rules in document image understanding. First, it is easy to apply either an additional strategy to a region that is hard to interpret with only one strategy, or a retry process having modified parameters. Second, software maintenance becomes easier, since addition/modification of rules is a relatively simple process that does not disrupt the rest of the system. Third, in a production system the processing is not carried out over the entire image uniformly, but only on necessary segments; thus, high efficiency is achieved. All these reasons make production rule-based systems eminently suitable for use in the domain of document understanding.

9. DISCUSSION

Document image understanding is a task that begins with pixel processing and ends with complex symbolic reasoning. Thus it is an area of research that draws upon techniques of image processing, pattern recognition, computer graphics, natural language processing, and artificial intelligence. There is an intense level of activity in this field in Japan and in Europe. Interest in this topic is also growing in the United States.

Several of the components for building a document understanding system are now well-understood, e.g., component detectors, line detectors, single font character recognizers, text parsers, etc. Several other components need to be refined, e.g., techniques for text region determination. Multifont character recognition without operator input will continue to be a challenging problem for the foreseeable future. The problem of raster to vector conversion of line drawings is not as formidable but several problems remain, e.g., handling of dotted lines, global considerations in thinning, etc.

The coordination of component processes is a problem that has been addressed in other domains such as speech understanding. The coordination of document understanding processes will have to be done using similar techniques with access to domain knowledge. Due to advances in several related areas it can be concluded that document analysis is now a task that is well defined and of a moderate level of complexity. Thus the prospects of tangible results are reasonably good.

ACKNOWLEDGEMENT

Several members of the Document Image Understanding Group at Buffalo contributed to this work. In particular, Jon Hull, Debashish Niyogi, Paul Palumbo, Joseph Piazza, Steve Tylock, and Ching-Huei Wang. Gregory Zack of Xerox Webster Research Center provided the impetus for this work and many suggestions. This work was supported in part by the United States Postal Service contracts 104230-84-D0962 and 104230-85-M3349.

10. BIBLIOGRAPHY

[1] C. Cox, B. Blesser and M. Eden, The application of type font analysis to automatic character recognition, *Proc. of Second IJCPR*, Copenhagen, 1974, 226-232.

[2] W. Doster, Different states of a documents content on the way from the Gutenbergian world to the electronic world, *Proc. Seventh ICPR*, Montreal, 2, 1984, 872-874.

[3] M. Ejiri, T. Miyatake, S. Kakumoto, S. Shimada, and H. Matsushima, Automatic recognition of design drawings and maps, *Proc. Seventh ICPR*, Montreal, 2, 1984, 1296-1305.

[4] R.P. Futrelle, A framework for understanding graphics in technical documents, *Proc. IEEE-CS Expert Systems in Government Symposium*, McLean, VA, 1985, 386-390.

[5] J.F. Harris, J. Kittler, B. Llewellyn, and G. Preston, A modular system for interpreting binary pixel representations of line structured data, in J. Kittler, K.S. Fu, and L.F. Paul (eds.), *Pattern Recognition Theory and Applications*, D. Reidel, 1982, 311-351.

[6] J.J. Hull and S.N. Srihari, A computational approach to visual word recognition: hypothesis generation and testing, *Proc. IEEE-CS Conference on CVPR*, Miami Beach, 1986, 156-161.

[7] K. Inanaga, T. Kato, T. Hiroshima, and T. Sakai, MACSYM: A hierarchical parallel image processing system for event- driven pattern understanding of documents, *Pattern Recognition*, 17(1), 1984, 85-108.

[8] K. Kubota, O. Iwaki, and H. Arakawa, Document understanding system, *Proc. Seventh ICPR*, Montreal 1, 1984, 612- 614.

[9] I. Masuda, N. Hagita, T. Akiyama, T. Takahashi, and S. Naito, Approach to a smart document reader system, *Proc. IEEE-CS Conference on CVPR*, San Francisco, 1985, 550-557

[10] F.S. Montalvo, Diagram understanding: the intersection of computer vision and graphics, Mass. Inst. of Technology, AI Memo 873, 1985.

[11] G. Nagy, S.C. Seth, and S.D. Stoddard, Document analysis with an expert system, *Proc. Pattern Recognition in Practice II*, Amsterdam, June 19-21, 1985.

[12] D. Niyogi and S.N. Srihari, A rule-based system for document understanding, *Proc. AAAI-86: Fifth National Artificial Intelligence Conference*, Philadelphia, 1986.

[13] P. Palumbo and S.N. Srihari, Text parsing using spatial information for recognizing addresses in mail pieces, *Proc. Eighth ICPR*, Paris, 1986.

[14] P. Palumbo, P. Swaminathan and S.N. Srihari, Document Image Binarization: comparison of techniques, *Proc. SPIE Symposium on Digital Image Processing*, San Diego, 1986.

[15] T. Pavlidis, A hybrid vectorization algorithm, *Proc. Seventh ICPR*, Montreal, 1, 1984, 490-492.

[16] A. Rastogi and S.N. Srihari, Recognizing textual blocks in document images using the Hough transform, TR 86-01, Dept. of CS, SUNY at Buffalo, 1986.

[17] J. Schurmann, A multifont word recognition system for postal address reading, *IEEE Trans. Computers,* C-27, 8, 1978, 721-732.

[18] S.C. Shapiro, Generalized augmented transition network grammars for generation from semantic networks, *American J. of Computational Linguistics,* 8(1), 1982, 12-25.

[19] S.N. Srihari, *Computer Text Recognition and Error Correction,* IEEE Computer Society Press, Silver Spring, MD, 1984.

[20] S.N. Srihari, J.J. Hull, and R. Choudhari, Integrating diverse knowledge sources in text recognition, *ACM Transactions on Office Information Systems,* 1(1), 1983, 68- 87.

[21] S.N. Srihari, J.J. Hull, P.W. Palumbo, D. Niyogi and C-H Wang, *Address Recognition Techniques in Mail Sorting: Research Directions,* TR85-09, Dept. of CS, SUNY at Buffalo, August 1985.

[22] F.M. Wahl, A new distance mapping and its use for shape measurement on binary patterns, *Computer Vision, Graphics and Image Processing,* 23, 1983, 218-226.

[23] C-H. Wang and S.N. Srihari, Object recognition in structured and random environments: locating address blocks on mail pieces, *Proc. AAAI-86: Fifth National Artificial Intelligence Conference,* Philadelphia, 1986.

[24] K.Y. Wong, R.G. Casey and F.M. Wahl, Document analysis system, *IBM Journal of Research and Development,* 26(6), November 1982, 647-656.

[25] H. Zen and S. Ozawa, Extraction of the fair document from mixed mode manuscript, *Proc. Conference CVPR,* San Francisco, 1985, 544-549.

LIVING IN A DYNAMIC WORLD

R.L. ANDERSSON

AT&T Bell Laboratories
Crawford's Corner Road (Rm. 4B607)
Holmdel, NJ 07733

ABSTRACT

Today's robot systems take an egocentric view of the world, assuming that the world is largely static and changes from state to state only in response to robot actions. To a large extent, this is a consequence of the limited bandwidth of current environmental sensing systems, in particular, doing any kind of vision takes most of a second or more.

We have designed and constructed a vision system based on a VLSI chip that locates objects at the full 60 Hz camera frame rate. Two systems provide a three dimensional description of object motion. In this environment, the robot must be capable of a sense of time: it must consider new data in the context of the old, and it must be aware of the temporal characteristics of its mechanism and processing electronics. We are exploring these concepts by creating a robot ping-pong player.

1. INTRODUCTION

Conventional robots live in a world that changes in very discrete steps, in response to clearly defined causes. Robot actions are interlocked with status lines and control signals to the outside world:

```
move_to input_bin
wait_for part_present
withdraw part
move_to punch
activate punch
wait_for punch_done
withdraw part
wait_for output_empty
move_to output_bin
release part
```

All events of interest are either directly sensed or directly caused by the robot, typically requiring a large number of discrete binary sensors. Sensors like vision can be brought into this framework by suitably restricting their functionality to discrete events (their function is also limited to start with), for example: take a picture now, the part is missing, the part is bad, or a good part is at (10 cm, 15 cm, 45 deg). The program can be represented by a simple finite state machine, and the time variable is effectively suppressed. We will refer to this model of programming as the discrete time approach. Although we can solve many useful tasks this way, it begins to break down as the robot, task, and environment become more sophisticated.

Consider retrieving an object from a conveyor belt. The manipulator must be at the right place at the right time at the right velocity to make a smooth pickup. The conventional approach is to take a single snapshot of an object to find its position, then update its position using an encoder mechanically mounted on the conveyor belt. Paul [10] illustrates servoing to a moving frame of reference such as a conveyor. Pragmatically, this may be fine for many low accuracy applications, but suppose one would like to pick up the object using visual information exclusively. One approach is that of Weiss [11], who considers servoing directly based on visual feedback, though our eventual application precludes this.

An alternative approach which appears similar to how a person might perform the task is to try to lead the object by a distance proportional to the expected time to get there at some reasonable expenditure of energy. As the arm approaches the object, we can refine its target position based on the currently observed object position and the arm's position and velocity. In contrast to the previous discrete time model, this strategy is a continuous time approach.

Continuous time systems require high sensor bandwidth — many sensor data points per second, and low latency — the time from the acquisition of data until it is applied to the control output. The latency is often significantly larger than the reciprocal of the bandwidth due to the use of pipelined parallel processing. In the case of single shot events, latencies include the fixed overhead of initiating and completing an operation.

The system latency is the sum of latencies due to the sensors, the processors, and the actuators. The real world doesn't stop while each latency expires, so in a rapidly changing environment, the sensor data is incorrect by the time the actuator reaches a position based on that data. An object moving at only a meter per second (moderate walking speed) moves a millimeter per millisecond, so a millisecond error will cause most assembly operations to fail, and an error of one robot trajectory generator cycle causes a 2-3 cm error, enough to totally miss an object to be picked up from the conveyor. Since people perform assembly operations at up to 15 m/sec, timing into the tens of microseconds range might be required.

We are interested in creating systems which can operate in situations with tight timing, position, and velocity constraints characteristic of a continuous time environment. The work described here is only part of a larger effort to make more intelligent robot controllers. In this paper we will scrutinize the requirements of a sense of time and outline a hardware and software approach to achieving it. We will begin by discussing our test problem in more detail.

CH2345-7/86/0000/0097$01.00 © 1986 IEEE

2. ROBOT PING-PONG

As a sample problem, we have chosen a robotic form of ping-pong proposed by Billingsley [4,8]. Ping-pong requires low latency in the sensors, actuators, and processing stages, and accurate system timing if the ball is to be directed along the desired trajectory. The robot controller must be capable of satisfying simultaneous position, velocity, and acceleration constraints.

The modified ping-pong table is shown in Figure 1. It has been scaled down from a standard table to make it easier for (stationary) robots. The rules are generally structured to make the game as feasible as possible, and to make it possible for any successful robots to be able to play each other. For example, the ball must pass through each end of the table; no shots off the side are allowed, and both robots and the background must be black.

Ball velocity can approach 10 meters per second, so available response time can range down to 0.2 seconds. A typical value is more like 0.4-0.5 seconds. Required paddle velocity is 1-2 m/sec, tightly controlled as a function of direction. The paddle velocity need not be too high as the outgoing ball velocity is twice the paddle velocity plus the incoming velocity, subject to the elasticity of the paddle/ball.

Complexity is added to the system by the mechanical configuration of the robot. The robot/paddle configuration must be able to cover a large (relative the robot's size) position and orientation set, and able to generate a controlled high speed motion at each point with minimal windup.

To generate the required speed and reach, we use a PUMA 260 robot with the paddle at the end of a roughly 0.5 meter stick perpendicular to joint six. A relatively slow motion of joint six provides large paddle speeds. The robot is hung upside down to keep the robot base from getting in the way. The paddle swings down on the ball rather than up. The robot positioning informally maximizes the usable working volume. Other robots with a larger reach, such as the PUMA 560, lack the necessary speed. SCARA robots are fast enough but don't have six degrees of freedom.

Ping-pong requires only 5 degrees of freedom, but the robot produces six. The free dimension corresponds to rotating the handle of the paddle in the plane of the paddle surface. Since the robot's wrist is half a meter away, the paddle orientation has a drastic effect on reachability and orientability, and picking it well is essential to good performance.

For simplicity, we ignore both spin and drag in trajectory calculations. The paddle velocity may be computed in closed form by requiring the paddle velocity to be normal to the paddle surface. Incoming and outgoing spins will require approximate techniques, perhaps with learning (iterative numerical solutions would be too slow).

2.1 Computer System Architecture

Our system is physically distributed over several processors. Each processor consists of a Pacific Microsystems PM68K processor (SUN-type 68000 based), a SKY Computers floating point board, 1 MB of memory, a network interface, a clock board we will discuss later, and miscellaneous I/O devices. The network is a custom small-area network featuring high bandwidth and low latency [1]. Star connections fan out from a backplane bus to the individual processors.

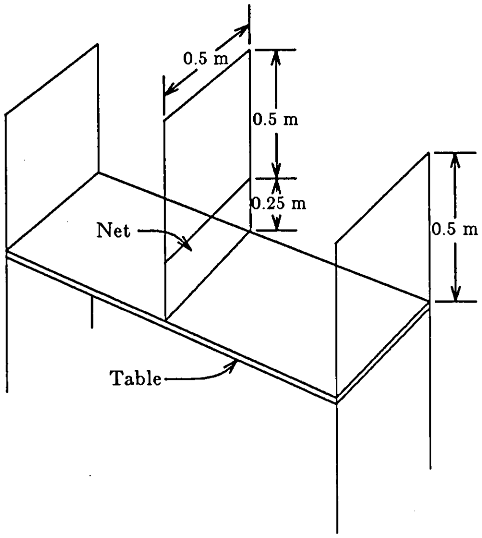


Figure 1. Robot ping-pong table.

The processor runs a multi-tasking operating system designed for real time performance, but with UNIX® (AT&T Bell Laboratories) look-alike calls [5]. The primary intertask (intra- or inter- processor) communications structure is a channel, a form of bi-directional UNIX pipe. A short (say 10 byte) message may be sent from one processor to another in about a millisecond, including all software overhead.

Software is written on a microVAX® (DEC) host running Unix, and downloaded into the 68000's for execution. The 68000s may access files on the host, typically to read calibration data and write debugging output.

2.2 Software Architecture

Although the network is homogenous, we overlay a structure by means of the channels we open, and the allocation of peripheral devices to processors. The allocation is performed "by hand" based on observed execution times for different systems components. The structure for robot ping pong is shown in Figure 2. Rtd is a specialized debugger, "chief" is a sequencer, and the SP2000 is a video tape system, the Kodak/Spin Physics Motion Analysis System, able to record 2000 full frames per second. Debugging tools are an important factor in constructing working systems.

Two vision processors drive two moment generator systems each, such that each processor has a stereo pair. One pair is looking at the far side of the table, one at the near side to achieve the proper field of view. All four cameras are genlocked together, so the vision processors can synchronously output an (x,y,z) triplet or "I don't see anything" at the 60 Hz frame rate. The image being processed is quite simple, consisting solely of a white ball against a black background. However, the ball can be greatly blurred due to its motion relative the camera. The vision system will be described in greater detail in a following section.

The next processor ("tranal") takes the (x,y,z) data and segments and fits it on the fly to create parabolic segments. The trajectory data is predicted forward to find the intersection of the trajectory with a fixed vertical plane at the end of the table, which serves as a common "point of reference." This intersection

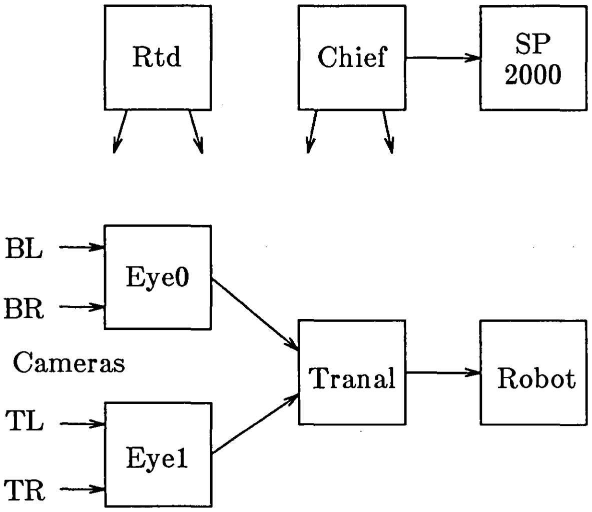


Figure 2. Task Structure.

data is then passed on to the final processor, the robot controller, still at the 60 Hz rate but with some additional latency.

2.3 Robot Controller Architecture

The robot itself is a Unimation PUMA 260. However, the electronics have largely been replaced. The LSI-11 and six 6503s normally controlling the robot have been replaced with a total of four 68020 based machines: one is a PM68K machine with a 68020 daughterboard, the other 3 are custom machines also with 68020 daughterboards (Figure 3). The standard PM68K processor serves as the main processor. Two custom boards serve as joint processors, each controlling three joints. The third slave processor is strictly computational, off-loading the main processor. The memories of the slave processors are accessible to the host but not vice versa, and slaves can interrupt or be interrupted by the host, but not each other.

To execute a typical motion, the program running on the main processor (the "user" program) calls up a motion initiator which performs the initial planning, computing variables for the path interpolation. The data is placed in the computational slave's memory, where the slave uses it to compute way points along the trajectory at a "major cycle" rate, say 16 msec, in response to interrupts generated by the main processor. The way points are sent to the joint processors by the main processor. The joint processors are interrupted at a "minor cycle" rate of 1 msec by an external source, causing the servo function to be executed. One of the joint processors interrupts the main processor every sixteenth minor cycle to create the major cycle.

The purpose of this involved structure is to minimize the amount of work required of the main processor to support ongoing motions, maximizing the time available for planning, while keeping the details visible to the planner. A totally shared memory architecture would even further reduce the nuisance work required of the main processor.

3. REAL TIME VISION WITH MOMENTS

The cornerstone of the research described in this paper is a vision system which operates in "real time." The system provides data with a sufficiently high bandwidth and sufficiently low latency that the data may be regarded as continuous for the class of problems we wish to consider, those involving interaction with a relatively macroscopic robot arm. A real time system is one that does its job in a time determined by external constraints, not just a system which is fast. Such a system can be used to watch moving objects, both in a conventional sense, or in more specialized domains such as aligning holes to be punched, or the apparent motion may be induced by the motion of the camera, when the camera is mounted on a robot arm or positioner. We would like to construct systems capable of extracting information useful for manipulation and inspection from gray scale images of three dimensional scenes in real time.

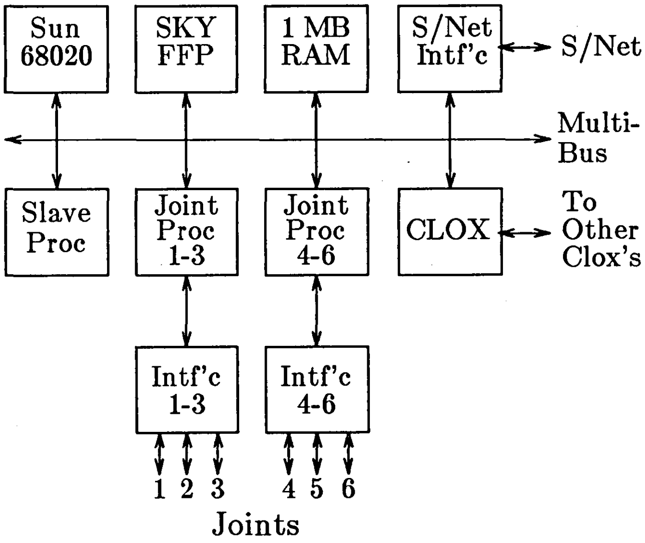


Figure 3. Robot controller architecture.

In general, we need to have full six dimensional information about the scene, three translational degrees of freedom describing object position, and three rotational degrees of freedom describing object orientation. For a vision system watching unknown objects, at least two camera views are required to extract the six degrees of freedom, i.e. stereo (binocular) vision is required. Ping-pong balls are rotationally symmetric (three degenerate degrees of freedom), therefore only the three translational degrees of freedom need be computed. The 3-D vision system must first process each two dimensional monocular image, then combine the results to form the three dimensional analysis. Let us take a brief look at existing vision systems.

Commercial vision systems are monocular, and most are capable of processing only binary images, typically by thresholding and run length compressing the image to reduce the amount of data before storing it in a fairly general purpose processor. Once read in, the data is processed for many tenths of a second before some decision is made. The canonical basis for these schemes is [6]; there are many current commercial imitators. Most of these systems are too slow to even be considered for continuous time robot control, but some systems using specialized hardware are approaching this regime. The extensibility of these binary image schemes into the gray scale three dimensional world is limited, however.

Image processing algorithms used for satellite image processing and television graphics, for example, can sometimes be made to run in real time with appropriate hardware when they operate on only local areas of the image. These algorithms take an image as input, and produce an image as output, such as an edge finding operator that produces a "line drawing." Such algorithms are not directly useful for robotics as they don't reduce the amount of information which needs to be processed, although they may simplify subsequent feature extraction operations. Reducing the amount of data to be processed without eliminating essential image content is fundamental to processing images in real time.

Research systems typically read a gray scale image into a frame buffer before processing it for several seconds, minutes, or even hours. Both two and three dimensional systems are under study. Although these systems are steadily advancing in capability, their low processing rate precludes their use for robot control. As research continues, we expect more of the features of these systems to be integrated into hardware running at real time rates.

A primary objective in constructing the vision system to be described was to build a system capable of processing simplified scenes at the "real time" rate necessary to use the data for robot control, to see what sort of system was necessary, and what would happen when we tried to use it. The overall approach is to make a "feature extractor" capable of extracting certain useful information from a complete video image streaming by at 60 frames per second, without having to store it for later processing. More complicated systems might be built by combining multiple types of feature extractors operating in parallel with multiple image processors connected in series. Once we can perform the necessary monocular processing at the proper rate on a reasonable sized piece of hardware, we can duplicate monocular processors and add further processing to generate the full three dimensional data. We'll start off by describing the monocular processing.

3.1 Moment Generator

Moments have been in use in computer vision for some time [7,9], and their use in physics and statistics goes back much farther. The equation defining the moments $M^{m,n}$ of an intensity array $a_{i,j}$ is:

$$M^{m,n} = \sum_{i,j} a_{i,j} i^m j^n \qquad (1)$$

where $m+n\,(m,n \geqslant 0)$ is the **order** of the moment, i is the column, and j is the row.

The zero through second order moments are sufficient to find the area, center of gravity, angle to major axis, and standard deviation along major and minor axes for an object, approximating the object as an ellipse. Second and higher order moments may be combined to form invariants which are used to characterize an object for purposes of discriminating among members of some set of objects.

The amount of time required to compute gray scale moments has hindered their use. On a VAX 11/780 with floating point accelerator, a direct calculation of the zero through second order moments of a 256*256 image takes 6.5 seconds.

The moment computation has been integrated onto a VLSI chip capable of computing a single zero through second order moment of a gray scale image in real time. Since there are six such moments, the moment processor module contains six chips. A number of techniques have been used to make the chip possible, which will be discussed below.

3.1.1 Power Vector Generation. We consider moment generation as a dot product:

$$M^{m,n} = \sum_t a_t p_t^{m,n} = (\bar{a}, \bar{p}^{m,n}) \qquad (2)$$

where the elements of the vectors are in the same order as a normal TV scan: $t = i + 256j$. The element $p_{i,j}$ of $\bar{p}$ will be referred to interchangably with p_{i+256j}.

The equation defining $\bar{p}^{m,n}$ is

$$p_{i,j}^{m,n} = i^m j^n. \qquad (3)$$

The element $p_t^{0,0}$ is one for all t. The first order moments require a counter for either x or y, depending on the moment. We can write the next value of each second order $\bar{p}$ as a function of the previous one, for example:

$$p_{i+1,j}^{2,0} = p_{i,j}^{2,0} + 2i + 1 \qquad (4)$$

with special cases for top of screen and left margin. We can build an iterative $\bar{p}$ generator composed of a single counter, a shifter, an adder, some "and" gates, and a small control programmable logic array (PLA).

3.1.2 Bit Decomposition. We can decompose $\bar{a}$ as

$$\bar{a} = 2^7 \bar{a}_7 + 2^6 \bar{a}_6 + \cdots + \bar{a}_0 \qquad (5)$$

If we substitute equation (5) into (2) and distribute, we obtain:

$$M^{m,n} = 2^7(\bar{a}_7, \bar{p}^{m,n}) + 2^6(\bar{a}_6, \bar{p}^{m,n}) + \cdots + (\bar{a}_0, \bar{p}^{m,n}) \qquad (6)$$

Computation of the dot products in Equation (6) requires only 1 by n bit multiplication, which may be implemented by n "and" gates, where n is the number of bits in $\bar{p}$.

At the end of each frame, we must compute

$$M^{m,n} = 2^7 F_7 + 2^6 F_6 + \cdots + F_0 \qquad (7)$$

where

$$F_k = (\bar{a}_k, \bar{p}^{m,n}) \qquad (8)$$

Equation (7) can be evaluated only once per frame (60 times per second) using Horner's Method of polynomial evaluation. The calculation is performed by the vision processor which controls the system.

The F_k accumulators are identical, simplifying the layout of the chip. The bitwise decomposition used to obtain fast operation also provides significant flexibility, making possible the computation of moments of different regions at the same time, for example.

3.1.3 Implementation. The techniques described above allow a moment generating I.C. to be constructed (Figure 4). The chip was designed using the MULGA symbolic layout system [12] in a 2.5 micron CMOS process and contains 10,214 transistors.

Six moment generator chips are placed on a Multi-Bus® (Intel) board with associated support logic. A preprocessor works independently on the intensity for each F_k bit, "and"-ing together an intensity map output and a location map output.

The intensity map converts intensity values to the desired precision and alignment. The map may implement binary thresholding, intensity windowing, non-linear response correction, or any combination of the above, for example. The intensity map may be used to correct for scene illumination

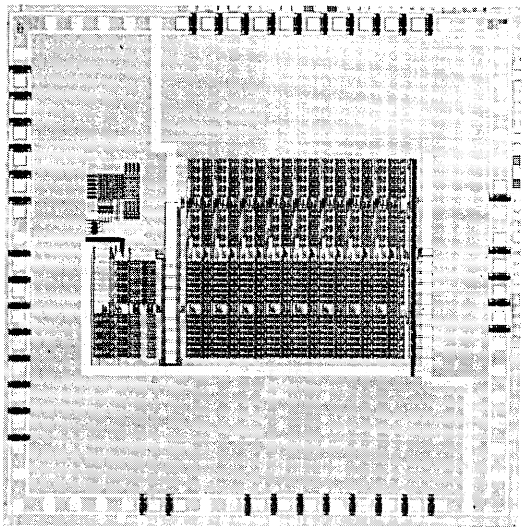


Figure 4. Moment generator chip.

problems or changes.

The location map defines the region of activity of each F_k; a bit is on if that F_k is to be activated at that position on the screen. To reduce the size of the location map, and simplify the host's job, regions are quantized into 8 by 8 pixel blocks.

Additional information on the chip design may be found in [2], and on its use in [3].

3.2 Three Dimensional Processing

Two moment generator systems are capable of processing the images from a pair of TV cameras at the full 60 Hz rate. Moments are a linear operator that commutes with other linear operators, so in theory, the moments of the background could simply be subtracted from the moments of the image, leaving the moments of the ball. In practice, noise from the camera and analog front end dominates the signal from the ball.

Instead, the intensity maps are used to separate object and background as follows. Intensities below a certain threshold are clipped to zero, saying in effect that anything sufficiently black should be ignored, in particular, various highlights off the backgrounds and supporting structures. On the other hand, any intensity above the threshold is considered as a gray scale value (relative the threshold). Importantly, this means that as the image of the ball is smeared towards black by the motion blur, the ball doesn't suddenly vanish, and the gray scale processing effects anti-aliasing, improving the numeric quality of the resulting centroids.

In the thus simplified scenes, once the vision processor has executed Equation 7 several times and performed

$$\bar{x} = \frac{M^{1,0}}{M^{0,0}} \qquad (9)$$

$$\bar{y} = \frac{M^{0,1}}{M^{0,0}}$$

the location of the centroid is known in each image. While the second order moments aren't needed to find the location of ping pong balls, it is interesting to notice that they contain information about the motion blur that can be recovered quantitatively.

Given that we have only a single object in the field of view, the standard problem of stereo vision, finding corresponding points in the two images, is eliminated. Since the orientation of the cameras is hard to control in practice, rather than a disparity based calculation, we represent each camera as a 4×3 matrix, and find the ball location by solving four equations (one for each coordinate of each camera) in three unknowns (x, y, z) using least squares. The calculation can be done in closed form in 3 msec on the 68000/SKY combination.

3.3 Experiment: Catching Balls

The three dimensional vision system and robot controller function, and have been used to conduct a preliminary experiment: catching a hand thrown ping pong ball in a styrofoam coffee cup. The system reliably caught balls on trajectories resulting in viable robot configurations. The program made no attempt to catch balls that were not catchable by its straightforward strategy. Work progresses on replacing the "user" robot program for catching with a much more sophisticated one for hitting the ball, but the other system components remain largely unchanged.

This initial experiment verified the operation of the vision system and robot controller, and served as an impetus for the analysis in the following sections.

4. LOW LEVEL EFFECTS OF TIME

As soon as the rate of change of the environment becomes comparable to the time constants of the components of the robotic system, we have to consider its effect on every part of the system, from sensor to processor to actuator. The next several subsections will detail these effects for each component.

4.1 Sensors

Unless the variables we are sensing are constant over time, any particular sensor output is meaningless unless associated with some particular time. The statement (sensor output) "the ball is 3 feet off the table" is useless, as it is inaccurate as soon as generated. Sensor values must be stamped with the time at which they are taken, as this defines the only time they have any validity.

Furthermore, the design of the sensor must be compatible with rapidly varying inputs, and able to define a precise timestamp for a given sensor reading. For example, sample and holds on the inputs to analog to digital converters prevent incorrect results from being generated, and the sample/hold's control input defines the sampling time very precisely.

Vision systems are no exception to the rule. TV cameras are designed for relatively slowly changing scenes. The electrical output of a camera is some complex function of the time varying light input over the time interval $(-\infty, now)$ depending on the type of camera.

For example, vidicons have a decay function such that the output at any time might be affected by some bright image many frames ago, in effect, the same as the persistence effects seen on CRT displays. Each point of the image is sampled at a different time as the beam sweeps over it, so a vertical bar moving horizontally generates pictures of a diagonal bar. This clearly makes the image interpretation process more complex.

On the other hand, CCD cameras operate as pipelined device, integrating one image while reading out the previous one.

There is no coupling from one image to the next, and no time varying response characteristics. Every pixel is effectively sampled at the same time. Because the electrical output due to a photon is not (strongly) dependent on the interval from its arrival to the end of the sampling interval, the center of gravity of a blurred image represents the average center of gravity during the sampling interval. For objects moving at essentially constant velocity over a sampling interval, we can assert that the object was at the center of gravity of the blurred image at the midpoint (center of gravity) of the sampling interval. The center of gravity is not affected by the motion blur from CCD cameras, so we can further assert that at the middle of the integrating interval of the frame, the object really was at the computed location. In addition to their better accuracy versus vidicons, this is another good reason for using CCD cameras (which we do).

4.2 Actuators

The primary question with actuators is understanding just what the temporal component of a command signal is. For example, consider a simple binary output to close a gripper. The semantics of this signal have an implied "now" component: close it **now**! However, from a planning perspective, for example a program trying to minimize robot cycle time, these semantics are unhelpful, as the gripper may in fact take forever to close. A conservative designer in a discrete time architecture might even put in a microswitch that says "the hand is now closed," which may ensure that the system works, but does nothing to help planning.

What we really would like to be able to assert is that after a certain period of time, a certain state will result. In the gripper example, this means we should be able to assert that after 50 msec perhaps, the gripper will be closed. By making this additional piece of information available to the system, we make possible additional optimizations such as starting to close the gripper before the manipulator has arrived at the part.

Of course, adding more arbitrary time constants to a system isn't desirable, which indicates the utility of systems modeling: we need to have good models of our actuators, whether they are binary actuators or whole manipulators. The gripper-closed limit switch is an aid to making models.

When applied to servo systems, the control signals may similarly be given temporal characteristics, one common example being the position waypoint to which a servo is trying to position the joint. In our system the semantics of the waypoint from the trajectory generator to the servo is that the servo should position the joint at the specified position at exactly the start of the next major cycle, when the next waypoint is specified.

In order for the servo system to be successful at convincing the joint to arrive at the specified destination at the specified time, various perturbing torques must be compensated. The output torque which must be generated to make an action occur must be supplied by the terms found in the servo equation of the joint. If the torque actually required by the manipulator differs from that computed by the servo function with no errors, the torque discrepancy must be "made up" by terms that are present, generally by a position or velocity error as the joint is moving. For example, the velocity damping necessary for stability will cause steady steady state position errors (lags) unless the desired velocity is fed forward into the servo equation [10]. Similar effects arise due to acceleration, friction, gravity, and inertial couplings between joints. Integrators can help, but only under quasi-static conditions, and make performance worse in rapidly changing conditions.

To summarize the section, we must add temporal semantics to the control signals sent to actuators, and we should develop good models of the system's response to the control signals. The better the models we have, the better the system performance.

4.3 Processing

Software events are not dependable in a multiprogramming environment because they are subject to the vagaries of the scheduling of device interrupts, the execution of higher priority tasks, and refresh interrupts. Newer processors are subject to additional factors such as instruction and memory management cache hits which are variable depending on previous events. Accordingly, we can't count on the processor performing the same repetitive action at the same relative time. Events can be well defined only by the hardware, not by software.

The imprecision of software controlled events suggests that we should be careful to build robot peripheral devices such that the application of control signals or latching of sensor readings is performed by hardware clocks, rather than under software control. Rather than having an interrupt request the program to directly read a value, we should have the interrupt latch the value into a register, which is then read by the program. Likewise, outputs should be latched by the peripheral devices under software control, but not applied to the actuators until the next hardware servo clock. As well as protecting against other competitors for CPU cycles, an additional level of latching isolates data dependencies in the control algorithm timing (we may even use different algorithms at different times) from affecting actuator timing. This amounts to pipelining the system. The slight additional cost of a register is compensated by the gain in repeatability of timing. Our robot controller is pipelined both at the major cycle rate and at the minor cycle rate.

The processing system also has significant latencies which must be taken into account. At some point in time, the software must commit to generating a control signal for some specific time. For example, consider changing the target of a robot motion while the arm is moving. To obtain a smooth trajectory, the preliminary setup calculations used to drive the trajectory interpolation must be evaluated based on some assumed initial position and time in the future. The program must know its execution time from the commit point to actually generating the control signal. This time should be minimized to reduce the number of potential interferences.

Once again, the more the program "knows," the better its performance. If the processing can be broken into two sections, an invariant portion that doesn't depend on the time varying variables and can be done first, and a second (minimal) section which does require this information, the program can obtain a timestamp at the completion of the first section reflecting the actual prevailing conditions. This time varying code can make the system hard to debug, but eliminates stored latency times, which must be conservative and tend to become outdated.

A sufficiently well controlled program may also be able to predict that the processor will be occupied by an interrupt for a certain time, and be able to compensate its own latency estimate accordingly. Clock driven interrupt processes can easily be predicted based on the time of their last occurrence and frequency.

4.4 The 'Clox' Board

We need to make accurate timing information available to the processor. Even though most microprocessor systems have free running timers (clocks) that are readable by software (often with substantial overhead), they are not useful for timestamping sensor or actuator data. The presence of software in the measurement path guarantees inaccuracy, because the processor may execute a fairly arbitrary sequence of instructions between the time of occurrence of the event and the time the software reads the timer.

In a real robotic system like ours for ping-pong, the sensors and actuators are distributed across multiple processors. Humans have trouble getting to meetings at the same time because we all have wristwatches with a different time (though this isn't the only factor at work). Likewise, in a distributed microprocessor system each processor has its own time, and times from different processors are not comparable. What we need to maintain a consistent view of time across the system is a "wall clock" accessible to all processors at once.

We implement the wall clock with a specialized "clox" board that resides in each processor, and a specialized clock bus connecting them. A wall clock must have the same value at each instant to each processor on a network. At first glance, this may seem to require that the clox boards be connected by a large number of wires, one for each bit. Since a clock has only two degrees of freedom, only two signals are actually required: one to specify the rate, and one to specify the offset. The simplest implementation uses one wire to carry an actual 1 Mhz hardware clock (square wave), and another to carry pulses one clock long to effect synchronization. One board is selected to generate the clock signal for all of the boards. Its clock frequency may be externally calibrated to ensure that the absolute accuracy is commensurate with the resolution. We are implementing the wall clock by a network of synchronized wristwatches.

Each processor must have a way of determining the occurrence times of events in devices attached to it. In general, this would require that a wire be attached to some suitable signal in the device and to the clox board, but this isn't particularly convenient. As an implementation technique, we can instead monitor activity on the processor's interrupt lines, which are generally driven directly from the same hardware signals we need to monitor. This has the advantage of not requiring odd wires jumping around between boards. An interrupt line must be dedicated to each signal the clock board is to be used to monitor. An event register latches the time of occurrence of the interrupt transition for subsequent reading by the processor when it handles the interrupt.

The clox board contains a software readable clock register which may be used by programs for time variant decision making. In addition, it is useful to be able to obtain software event times for diagnostic purposes: how long does this routine take on average? at most? Because of the simplicity of the access protocol, we need not call it via an operating system trap. The low cost of access means that we may routinely track execution times.

A novel consequence of having a "wall clock" is that software event occurrence times may be compared across processors, directly measuring software latencies across the network, for example. Such numbers may be obtained even for rare single shot situations. A simple example measures the latencies of a processor to processor write. Processor A executes:

```
timel = now ();
write (channel, buffer, count);
time2 = now ();
```

and processor B executes:

```
time3 = now ();
read (channel, buffer, count);
time4 = now ();
```

where **now** is a macro that gets the current time. By examining the times 1-4, we can easily determine: the execution time of the write, the execution time of the read, whether the read or the write began first, and most importantly, the time from when processor A started sending the data to the time processor B had it and could perform further processing.

5. TOWARDS A SENSE OF TIME

In the previous sections, we have taken a detailed look at the microscopic treatment of time. In this section, we'd like to take a broader look at the overall implications and trends.

5.1 Knowledge of Manipulator Capabilities

A recurring theme was the need for the system to know more information about itself: sensor characteristics, actuation delays, processing latencies. The same need is present in higher levels of the system, not only for timing related information, but general knowledge about robot capabilities as well.

Present robot controllers are sadly ignorant of their own capabilities in the temporal sense. Most controllers can compute that they can move to certain position, but not be able to specify the time required to do so, or be able to make the motion occur in a prespecified length of time. The former capability is necessary to evaluate alternative motions or plan intercepts with moving objects, the latter to be able to execute them.

We have these capabilities in our system, but only in a crude sense, treating each joint as decoupled and with only a rough joint performance figure. For maximum accuracy and thus performance, we need to be able to model the robot dynamics, including actual actuator torques, inertias, and couplings. As the specification of the motion becomes more complex, such as straight line motion or a series of continuous motions, it becomes more and more difficult to predict the motion time without actually executing it, at least in simulation.

5.2 Continuous Sensor Integration

New data can be generated by sensors like the moment generator at a 60 Hz rate. If we evaluate each data point independently, we suffer the expense of generating an entirely new plan for each data point, and the risk that the series of plans may be incompatible with one another. A better approach is to evaluate new data in the context of the old.

The processing time required to generate an initial plan is almost certainly much greater than the time required to fine tune an existing one. In robot ping pong, the initial plan requires selecting the desired return, appropriate robot configuration, and a series of trajectory segments. As new data on the ball trajectory becomes available, tuning a few details may suffice to update the plan.

On the other hand, when a sudden discontinuity in input data occurs, for example, after a ball bounces with a lot of spin, we must take a drastically different action. Rather than picking a totally new plan, we should operate within the confines of actions we have already taken, as the arm will be in motion, and this constrains the possible destinations. The decision criteria for making a switch in overall plan should be expressed in terms of the actions, not the input data, so that we avoid needless thrashing.

The process of understanding new data in the context of the old and making incremental refinements in the planned actions seems much more similar to a human's continual perception of time than the discrete time approach of robot controllers.

The work described in the section is still in progress, and is being driven by the observed needs of the task. The natural requirement for self-knowledge, especially about processing capabilities, is an interesting development which bears further observation. As the tasks get more complicated, the amount of self-knowledge must increase. Perhaps this is a step (albeit a small one) in the direction of machine self-awareness.

6. CONCLUSIONS

As we improve robot performance and increase the complexity of robot tasks, we will need to change the way we think about robots from an event driven perspective to a continuous time model. The development of real time sensor systems is accelerating the trend. We can start now by paying careful attention to the sources of timing related effects. Ultimately, robot systems will have to be able to understand the dynamic nature of the world they live in.

7. ACKNOWLEDGEMENTS

Robotics is a multi-disciplinary field; I've been fortunate to be able to stand on the projects of many other researchers. Thanks to Bob Gaglianello and Howard Katseff for Meglos, and Brian Ackland and Neil Weste for MULGA. Thanks also to John Jarvis for helping all this take place, and Richard Paul for letting me tap his formidable experience. Special thanks to Richard Seide and George Whyte for admirable work as construction crew.

REFERENCES

[1] S.R. Ahuja, "S/Net: A High Speed Interconnect for Multiple Computers," IEEE Journal of Selected Areas in Communication, SAC-1, No. 5, November 1983, p. 751-756.

[2] R.L. Andersson, "Real time video moment generator chip," in N. Weste, K. Eshraghian, "Principles of CMOS VLSI Design: A Systems Perspective," Addison-Wesley, 1985, p. 407-424.

[3] R.L. Andersson, "Real Time Gray Scale Video Processing Using a Moment Generating Chip," IEEE Journal of Robotics and Automation, Vol. RA-1, No. 2, June 1985.

[4] J. Billingsley, "Machineroe joins new title fight," Practical Robotics, May/June 1984, p. 14-16.

[5] R.D. Gaglianello, H.P. Katseff, "Meglos: An Operating System for a Multiprocessor Environment," Proceedings of the 5th International Conference on Distributed Computing Systems, May 1985.

[6] G.J. Gleason, G.J. Agin, "A Modular Vision System For Sensor-Controlled Manipulation and Inspection," Proceedings of the 9th International Symposium on Industrial Robots, SME/RIA, p. 57-70, March 1979.

[7] M. Hu, "Visual Pattern Recognition by Moment Invariants," IRE Transactions on Information Theory, IT-8, February 1962, p. 179-187.

[8] D. Loewenstein, "Computer Vision and Ranging Systems for a Ping-Pong Playing Robot," Robotics Age, August 1984, p. 21-25.

[9] A.P. Reeves, A. Rostampour, "Shape Analysis of Segmental Objects Using Moments," Proceedings of the IEEE Computer Society Conference on Pattern Recognition and Image Processing, p. 171-174, August 1981.

[10] Paul, R.P., "Robot Manipulators, Mathematics, Programming, and Control," MIT Press, 1981.

[11] L.E. Weiss, A.C. Sanderson, C.P. Neuman, "Dynamic Visual Servo Control of Robots: An Adaptive Image-Based Approach," IEEE International Conference on Robotics and Automation, March 1985, p. 662-668.

[12] N.H.E. Weste, "Virtual Grid Symbolic Layout," Proceedings of the 18th Design Automation Conference, June 1981, p. 225-233.

CMU Sidewalk Navigation System:

A Blackboard-Based Outdoor Navigation System

Using Sensor Fusion with Colored-Range Images

Y. Goto, K. Matsuzaki
I. Kweon, T. Obatake
Robotics Institute, Carnegie-Mellon University
Pittsburg, PA. 15213

Abstract

We describe the CMU Sidewalk Navigation System, which can drive a vehicle in the outdoor environment of the CMU campus. The system includes all modules necessary for outdoor navigation -- modules for route planning, local path planning, vehicle driving, perception, and map data. The perception module uses sensor fusion with color and rage data to analyze complex outdoor scenes accurately and efficiently.

1. Introduction

The goal of the CMU SCVision group is to create an autonomous mobile robot system capable of operating in outdoor environments.[1] The complexity of the environment requires the system to have a powerful perception ability, capable of analyzing natural objects, and a planning ability which can work in non-uniform conditions. Because this navigation system will be very large, we need mechanisms to combine programs into whole systems and mechanisms for parallelism in computation.

We already have several systems towards the goal: a road following system with color classification [5], road network navigation with a simple map [1], scene analysis with a laser range sensor [2], and the blackboard [4].

The CMU Sidewalk Navigation System is a milestone system toward our goal. In this system, we focus on two points. The first is to build a whole system based on a good system architecture so that the system is both complete (containing every necessary module) and efficient. We achieve that goal by adopting a blackboard-based architecture. The second point is to create perception modules with sensor fusion that work well in our outdoor environment.

The test site for the CMU Sidewalk Navigation System is the CMU campus, containing a network of sidewalks and intersections, along with grass, slopes, and stairs. The system can drive the vehicle through these objects to get to its destination.

2. System Architecture for the Outdoor Navigation System

2.1. Hardware Configuration

The hardware for the CMU sidewalk navigation system consists of three SUN-3 workstations, the vehicle, the color TV camera, and the laser range sensor. The workstations are linked together with Ethernet, and the workstations and the vehicle are linked with radio communication. Figure 1 shows the vehicle called *Terregator.*

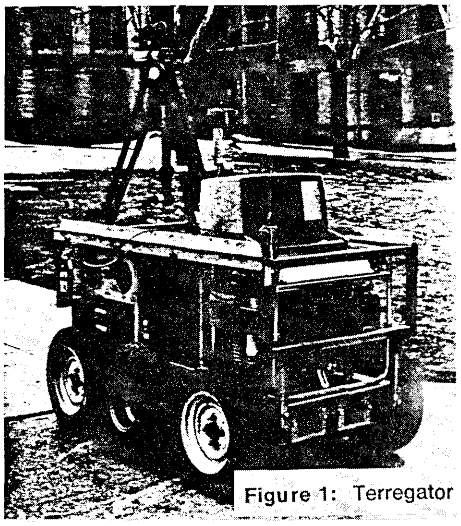


Figure 1: Terregator

2.2. System Architecture

2.2.1. Stages of Navigation

In order to create a reasonable system architecture, we have to start by analyzing outdoor navigation.

If the navigation system uses only one uniform navigation mode, the system architecture issue is not essential. But, in general, outdoor navigation includes several navigation modes. The example shown in Figure 2 illustrates this situation. The vehicle running from the starting point to the destination has to *follow the road*, *turn at the intersection*, *climbing the slope* and *cross the terrain*. *Turning at the intersection* needs a more complex method to drive the vehicle than *following the road*. Perception for *crossing the terrain* is different from perception for *turning at the intersection*. In *following the road* we can use assumption that the ground is flat, which makes perception easier. But *climbing the slope* does not satisfy this assumption. This is one reason why the

[1] This research was sponsored by the Defense Advanced Research Projects Agency, DOD, through ARPA Order No. 5351, and monitored by the U. S. Army Engineer Topographic Laboratories under contract DACA 76-85-C-0003. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the U.S. Government.

CH2345-7/86/0000/0105$01.00 © 1986 IEEE

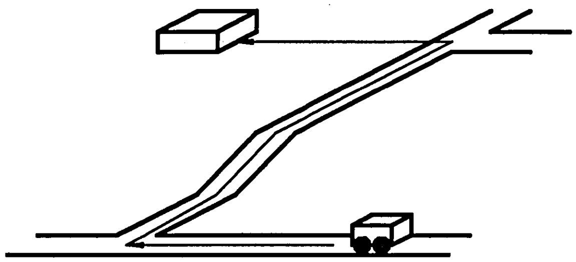


Figure 2: Outdoor Navigation

outdoor navigation system needs good system architecture.

We decompose navigation into two processing stages. The first stage is the *route planning stage*, and the second stage is *local navigation*. In the route planning stage the system selects the best navigation route, from several possible routes to get to the destination from the starting point. The system divides the whole route into a sequence of *route segments*. In each route segment, objects on which the vehicle can run are constant. The navigation system can drive the vehicle using a single uniform driving mode, for example, *following the road* or *turning at the intersection*, and a single perception mode. In this stage, using the map data is essential.

Local navigation is navigation within one route segment. In the local navigation stage, the navigation mode is constant and the main task is to drive the vehicle along the route segment. Local navigation uses perception to find a safe passage for the vehicle, and to determine the actual vehicle driving path.

In contrast, our earlier and simpler navigation system did not have the route planning capability and has only one navigation mode for local navigation.

The system architecture of the CMU sidewalk navigation system is indicated in Figure 3. We decomposed the whole system into several modules. The modules indicated with blocks are separate processes, running independently, and communicating with each other through the BLACKBOARD. In selecting this decomposition of the whole system into these modules, we followed the principle of *information hiding*. The CAPTAIN module and the MAP NAVIGATOR module are responsible for the route planning, and they do not know the result of perception or how to drive the physical vehicle. The PILOT module, the PERCEPTION module and the HELM module are responsible for the local navigation, and they do not know the destination, the whole route, or the sequence of route segments. What they know is limited to only one route segment at one time. We will explain the system architecture in detail in the following sections.

2.2.2. The Blackboard-Based Architecture

Our BLACKBOARD provides modules with communication and synchronization facilities [4]. Using a blackboard-based architecture brings two main advantages to building our navigation system.

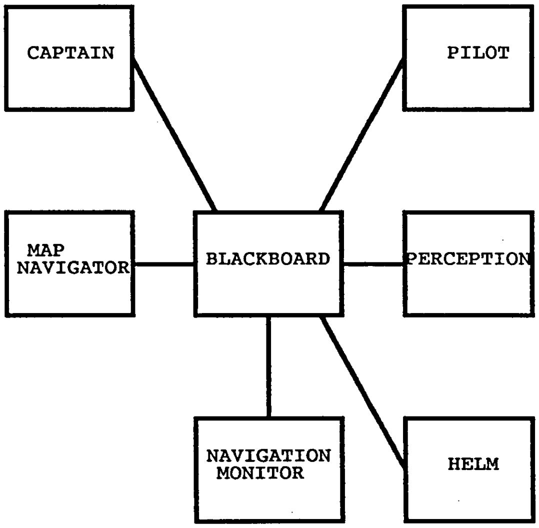


Figure 3: System Architecture

The first advantage is parallelism in execution. We decompose the whole system into several parallel modules. Because the most time consuming operation is perception, it is an independent process, the PERCEPTION module, running on its own machine, and not disturbing other modules. Because the HELM module which drives the physical vehicle needs real-time response, it is another separate process. Communication and synchronization of all modules are handled by mechanisms of the BLACKBOARD.

The second advantage is that using a blackboard makes it easier to combine several programs into a whole system. Our BLACKBOARD provides a good mechanism to connect modules, and limits the interactions among modules. For instance, each module can work in its most natural and convenient coordinate frame, with the BLACKBOARD converting among reference frames. We use the principle of *information hiding* so that the interfaces between modules are small. This keeps communication costs low and allows good modularity. The details of the BLACKBOARD are explained in following sections.

2.3. Module Structure

In this subsection we explain each module.

2.3.1. The CAPTAIN Module and the Mission

At the upper level of the system is the *CAPTAIN* module that receives instructions from the controlling person and oversees the mission. The mission consists of a number of steps, and the CAPTAIN sequences through the steps. For each step, there is a *destination* that tells where to go and one or more *constraints* that tell how to go. For example, "go to intersection D" gives a destination and "keep right" gives a constraint. Each mission step also has a *trigger* condition and an action which will be executed if the trigger condition is satisfied. Triggers can be used to move on to the next mission step when one step is completed.

The CAPTAIN sends the destination and the constraints of each mission step to the MAP NAVIGATOR one step at a time, and gets the result of mission step, *success* or *fail*, from the MAP NAVIGATOR.

2.3.2. The MAP NAVIGATOR and Route Planning

The MAP NAVIGATOR does the route planning based on a destination and a constraint sent from the CAPTAIN, gives the PILOT directions for driving along the route, and reports the result of the mission to the CAPTAIN.

The MAP NAVIGATOR contains two main parts, the ROUTE SELECTOR, and the ROUTE SEGMENT DESCRIBER (see Figure 4). The ROUTE SELECTOR creates the route plan, and decomposes it to a sequence of the route segments so that each route segment has only one *navigation mode*. The current system has several navigation modes: *follow-road*, *turn-at-intersection*, *go-through-intersection*, and *go-through-slope*. Our future system will have another navigation mode, *cross-country*, in order to navigate on open terrain.

The ROUTE SEGMENT DESCRIBER generates the description of the route segment. The purpose of *route segment description* is to provide the PILOT with the information necessary for navigation within the route segment. It includes path objects (e.g., pieces of road, intersections), navigation modes, the conditions to exit from the route segment, the constraints to drive the vehicle, and object descriptions. *Path objects* are the objects on which the vehicle should run. *Object descriptions* describe the location and the shape of the objects (such as landmarks) which the PERCEPTION module can see while running on the route segment. This description is created by copying a part of the Map data, and is used as a prediction for the PERCEPTION module. One important point is that only the MAP NAVIGATOR maintains the Map data.

The route segment description is sent to the BLACKBOARD and forwarded to the PILOT. When the PILOT finishes the route segment, it reports the result. If the result is *success*, the ROUTE SEGMENT DESCRIBER sends next route segment description.

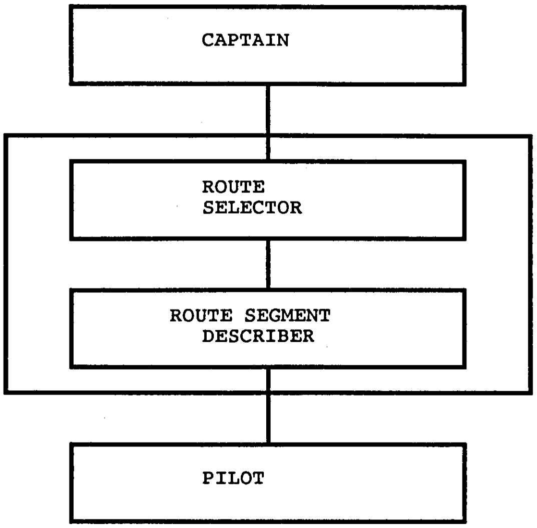


Figure 4: The MAP NAVIGATOR Module

2.3.3. The PILOT and Local Path Planning

The PILOT, the PERCEPTION and the HELM work together for local navigation. The PILOT operates continuously to conduct the navigation within the route segment. The PILOT contains several sub-modules that form a sequence as shown in Figure 5, to process each area to be traversed.

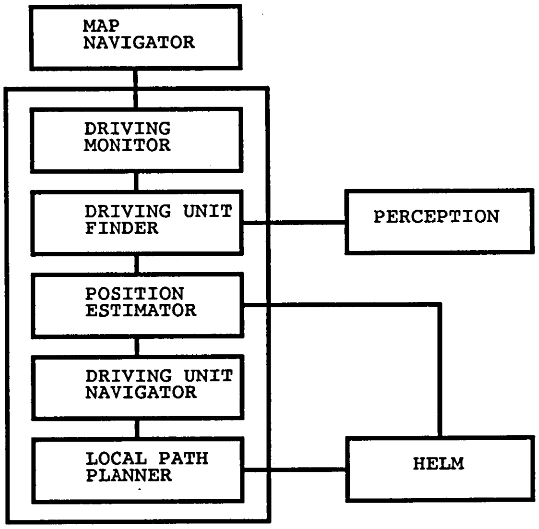


Figure 5: The PILOT module

The DRIVING MONITOR, the top level of the PILOT, receives route segment descriptions whenever a newly created route segment appears in the BLACKBOARD. The DRIVING MONITOR breaks the route segment into pieces called *driving units*, so that the PERCEPTION can detect one driving unit separately (see Figure 6). The DRIVING MONITOR builds a *driving unit description* for each driving unit, describing the location of the driving unit and the objects which PERCEPTION should see. Following the principle of information hiding, the driving unit description tells where and what objects the PERCEPTION should see, but it does not tell how to detect them.

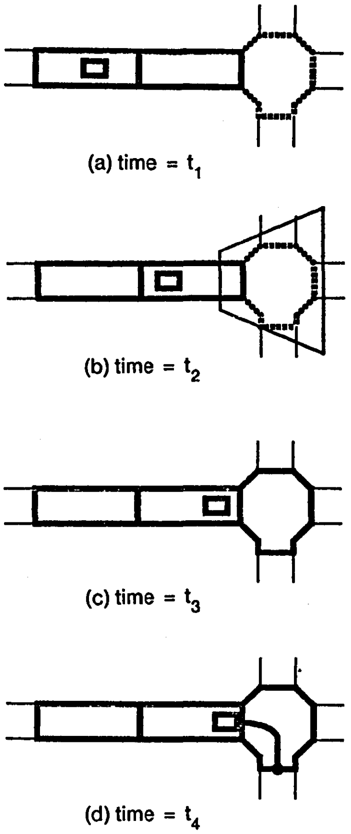
The DRIVING UNIT FINDER works as an interface to the PERCEPTION, getting the newly created driving unit description from the DRIVING MONITOR and sending it to the PERCEPTION through the BLACKBOARD. If the result of perception is written in the driving unit description in the BLACKBOARD, the DRIVING UNIT FINDER retrieves it into the PILOT.

The POSITION ESTIMATOR determines the vehicle position using position estimations generated both by perception and by dead reckoning. When PERCEPTION sees objects which can be landmarks, PERCEPTION can estimate the vehicle position using predicted object locations and shapes. For example, stairs and intersections are good landmarks. Sidewalks are not sufficient as a landmark, because PERCEPTION looking at only a sidewalk cannot tell the location along a sidewalk. It can tell only the distance from the edge of a sidewalk. Therefore, the position estimation by the PERCEPTION is sometimes complete and sometimes is not complete. The position estimation by dead reckoning is given by the HELM. The POSITION ESTIMATOR combines both of them and determines the vehicle position.

The BLACKBOARD stores two kinds of vehicle positions. One is *perceived vehicle position*, and the other one is *moving vehicle position*. Because the perceived vehicle position is estimated by only the POSITION ESTIMATOR once per driving unit, it is discrete. On the other hand, the moving vehicle position is estimated by not only the POSITION ESTIMATOR but also the HELM using dead reckoning. Because the HELM updates the moving vehicle position frequently, it tells the vehicle current position. Because both positions are stored in the BLACKBOARD, all modules can use them. Currently they are used by the PERCEPTION and the LOCAL PATH PLANNER.

The DRIVING UNIT NAVIGATOR plots *local path constraints* in the driving unit using the result of perception and the driving constraints given by the MAP NAVIGATOR.

The LOCAL PATH PLANNER gets the local path constraints and creates the *local path plan* from the vehicle's current position, through all intervening driving units, reaching to the far edge of the newly scanned driving unit. Because the vehicle is not in the newly scanned driving unit, the LOCAL PATH PLANNER keeps the old local path constraints to calculate local path plan, and discards them if they are not necessary. The algorithm for local path planning is based on Lozano-Perez's method [3]. This method generates a sequence of line segments, which the LOCAL PATH PLANNER converts to a smooth path. Therefore, the vehicle turns along a curved line. The generated local path plan is passed to the HELM through the BLACKBOARD.


(a) time = t_1

(b) time = t_2

(c) time = t_3

(d) time = t_4

Figure 6: Process Sequence in the PILOT

Figure 6 illustrates the process sequence in the PILOT. The black painted box is the vehicle. The bold line indicates the driving unit detected by the PERCEPTION already. The dotted line shows the driving unit newly created by the DRIVING MONITOR. At time t_1 the DRIVING MONITOR creates new driving unit description and sends it to the DRIVING UNIT FINDER. The DRIVING UNIT FINDER sends it to the PERCEPTION. When the vehicle reachs the best place to detect the required driving unit, the PERCEPTION inputs image data with the sensors (time t_2). The thin line shows the sensor view frame. At time t_3, the PERCEPTION finishes processing and reports the object shapes and the vehicle position. The DRIVING MONITOR starts creating next driving unit description. And at time t_4 the LOCAL PATH PLANNER generates new local path plan and passes to the HELM.

2.3.4. The HELM and Driving the Vehicle

Whenever a new local path plan appears in the BLACKBOARD, the HELM discards the old path plan and picks up new one. The HELM converts the local path plan, which tells only trajectory, into vehicle driving commands, and feeds them to the vehicle. In addition to driving the vehicle, the HELM is responsible for maintaining the *vehicle moving position* stored in the BLACKBOARD. Because the task of the HELM needs quick response to control the vehicle, it is implemented as an independent process.

2.3.5. The PERCEPTION Module

PERCEPTION picks up a driving unit description from the BLACKBOARD when a new one appears. PERCEPTION has two tasks: detecting navigable passages, and, if possible, estimating vehicle position. The details of PERCEPTION are explained in the next section.

2.3.6. The BLACKBOARD

The BLACKBOARD provides the modules with facitilies for communication and data management. Our BLACKBOARD looks like a traditional blackboard, with several additional properties that make it useful for navigation:

1. **parallel asynchronous execution of modules.** This property makes it possible to execute all modules in parallel.
2. **transparent networking between processors.** This property makes it easier to build interfaces between modules.
3. **no pre-compilation of data retrieval specification.** This property makes it easier to pick up desired data from the BLACKBOARD.
4. **geometric reasoning.** Coordinate transformation and geometric calculations are done by the BLACKBOARD. Data retrieval from the BLACKBOARD with geometry is used in several places.

2.3.7. The MAP

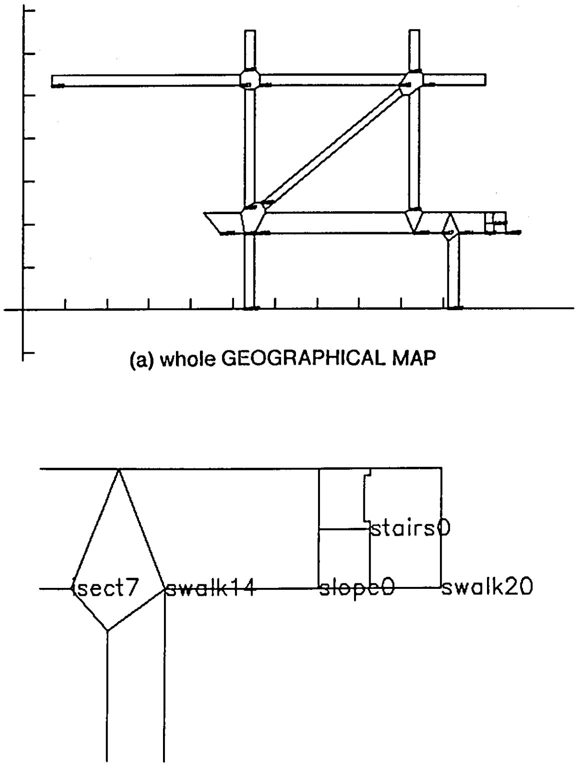
The MAP is the main data base in our navigation system. It consists of two parts, GEOGRAPHICAL MAP and OBJECT DATA BASE.

The GEOGRAPHICAL MAP is similar to usual maps which we use in daily life, and tells object locations with their outlines. The MAP NAVIGATOR uses the GEOGRAPHICAL MAP to pick up objects belonging to the route segment. Figure 7 shows current GEOGRAPHICAL MAP.

The OBJECT DATA BASE stores the object descriptions.

1. **Perception Feature:** Both of the object's three dimensional shape and its color are stored to produce object description for the PERCEPTION. Three dimensional shape is expressed with surfaces.
2. **Role in Navigation:** Some objects can be landmarks, and other objects can be paths. Therefore, the OBJECT DATA BASE indicates roles of objects: *landmark*, *path*, *obstacle* and *no meaning*. Also, navigation costs on objects are stored. These data are useful when the MAP NAVIGATOR performs route planning.

The MAP is stored in the BLACKBOARD and accessed only by the MAP NAVIGATOR. Currently we assume that the MAP has all necessary information and it is correct and complete. But our future work will include "map revising", starting with an incomplete map, and updating it during navigation.



(a) whole GEOGRAPHICAL MAP

(b) stairs and slope

Figure 7: The GEOGRAPHICAL MAP

2.3.8. The NAVIGATION MONITOR

The NAVIGATION MONITOR is a graphics system which displays the current navigation situation. It displays the route segment, the driving unit, the sensor view frame, the vehicle position, and the local path plan. Because the NAVIGATION MONITOR is implemented as an independent process, it does not disturb other modules. Whenever the data to be displayed appear on the BLACKBOARD, the NAVIGATION MONITOR retrieves and displays them.

3. Perception Using Colored-Range Image

3.1. Requirements and Approach

The basic problem for the perception is caused by the complexity of outdoor scenes. Some objects have very complicated shapes and colors from which it is difficult to extract surfaces or edges. This requires powerful sensors and algorithms for object detection. Also, the processing time of the perception is critical, because it eventually constrains vehicle speed. Even if we have a very powerful perception program which can detect complicated objects, it will be computationally expensive. Therefore, having a *single* powerful object detection program is not an adequate solution.

To overcome these difficulties, we take a sensor fusion approach. There are several types of sensor fusion methods [4]. This PERCEPTION module uses two types of sensor fusion.

The first type is low level sensor fusion, doing segmentation using color and range data simultaneously. We call the image which has color information in addition to range values a *colored-range* image. Using this method, we can segment objects with uniform color but varying surface orientation, as well as objects with smooth surfaces and varying colors. This method is used to analyze complicated scenes.

The second type of fusion is a higher level sensor fusion or sensor selection. The PERCEPTION module has both a *colored-range* segmentation program, mentioned above, and a color segmentation program, and uses these programs selectively. The former program can extract segments in complicated scenes, while the later program is adequate for simple flat scenes and uses much less processing time. This type of sensor fusion achieves both powerful perceptual ability and fast processing.

3.2. PERCEPTION Module Architecture for Sensor Fusion

The PERCEPTION receives perceptual requests from the PILOT, and analyzes sensor data to compute its response. The main effort to design the PERCEPTION module is how to combine several types of sensors and sensor data processing modules into one system and make them work efficiently. We designed a hierarchical structure and a monitor module which manages all parts of the hierarchy.

Figure 8 illustrates the structure of the PERCEPTION module. This is composed of the PATCH MAKER, the OBJECT FINDER, the POSITION CALIBRATOR, and the PERCEPTION MONITOR. Sensor data go through in the order of the PATCH MAKER, the OBJECT FINDER, and the POSITION CALIBRATOR.

The data interface between each module is designed to be independent of the algorithms used by each module. This allows each layer in the hierarchy to have several modules based on different algorithms. In the current system, the PATCH MAKER includes two types of segmentation modules, and one object finding module can work on the results of both segmentation modules because of the common data interface. The PERCEPTION MONITOR is a key for this hierarchical processing. We describe it in detail in the next section. Other modules are explained in following paragraphs.

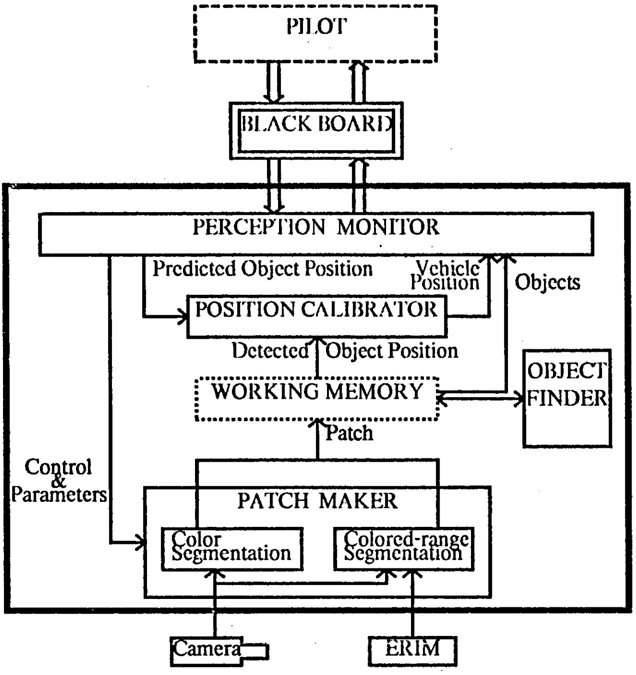


Figure 8: Structure of PERCEPTION Module

3.2.1. The PATCH MAKER

As a segmentation module (PATCH MAKER), this system has a color segmentation module, a range segmentation module, and a *colored-range* segmentation module. The color segmentation, the range image segmentation, and *colored-range* image segmentation are described in Section 3.4.

The data structure which holds patch data is common to both segmentation modules. These data include *color type*, *surface type and normal*, *polygons for boundary shape*, and *relation to neighbor segments*.

3.2.2. The OBJECT FINDER

The OBJECT FINDER identifies each segment as a part of a predicted object. This algorithm is described in rule-base style, with two kinds of rules. One kind identifies detected segments as parts of predicted objects, and the other type of rule finds the actual correspondence between perceived and predicted polygons. In other words, the first set of rules deals with symbolic matching, while the second knows about detailed geometry.

The OBJECT FINDER uses a WORKING MEMORY. The PATCH MAKER assigns the Patch data into the WORKING MEMORY. Also, the PERCEPTION MONITOR assigns predicted object shape and feature data into the WORKING MEMORY. These data include color, surface type, and shape. The OBJECT FINDER uses WORKING MEMORY data to match predicted with detected data.

3.2.3. The POSITION CALIBRATOR

The predicted objects are described in the current coordinate system, but the vehicle coordinate system is used to describe the detected objects. The POSITION CALIBRATOR then computes the vehicle position in the current coordinate system, applying the transformation matrix between two coordinate systems. The problem for this computation is that the predicted object shape and the detected object shape are not same because of imperfections in the MAP and the perception. Therefore, the POSITION CALIBRATOR has to find the most appropriate matching for these two shapes.

To get the best matching point, the POSITION CALIBRATOR calculates the distance between the predicted vertices and the detected vertices of object polygons, and finds the position which minimizes the distance. Sometimes, a scene is composed of only parallel lines (e.g., *sidewalk*) or a point (e.g., *tree*), which are insufficient to decide a matching point. In this case, the POSITION CALIBRATOR derives a line equation on which the vehicle is located instead of a point for vehicle position.

3.3. The PERCEPTION MONITOR

The PERCEPTION MONITOR has two major roles: communication with other modules (the PILOT) and control of internal submodules. As mentioned before, a design principle of this system is to provide a common structure for different sensors and algorithms. This tends to make the module interface rather high level. For example, an image input position is usually decided by an external module using sensor parameters. However, if there are several types of sensors with different view angles, the common interface for those modules will be ***where the PERCEPTION should see*** instead of ***where the PERCEPTION should look from***. This means the PERCEPTION module itself has to decide where the best position is from which to see the requested place. The communication with other modules means doing such kinds of interpretation between the high level module interface commands and actual commands to internal submodules.

Control flow of the perception process is rather simple. It progresses in order of segmentation, object finding, and position calibration. The PERCEPTION MONITOR activates the PATCH MAKER, the OBJECT FINDER, and the POSITION CALIBRATOR in this sequence. The functions for the interpretation of the high level commands from the other planning module (the PILOT) are described in following paragraphs.

3.3.1. Selection of Sensor and Segmentation Modules

The PILOT requests what objects to see, but does not say which sensor should be used. The PERCEPTION MONITOR decides which sensor and segmentation module is the best for the requested objects. The current system has two sensors and segmentation modules. If all requested objects are *sidewalks* or *intersections* on a flat plane, the PERCEPTION MONITOR selects the color segmentation module as a PATCH MAKER. If three-dimensional objects such as stairs, and slopes are included in the requested objects, the PERCEPTION MONITOR selects the *colored-range* segmentation module.

3.3.2. Decision of View Frame and Resolution

The PILOT requests to see objects as quickly as possible within a tolerable error allowance, but does not specify the resolution or view frame of PERCEPTION. The view frame of PERCEPTION depends on a tolerable error allowance for the PERCEPTION. If the error allowance is small, a meaningful view frame will be limited to near the vehicle, even if the view angle of the sensor covers a greater area. Also, the view frame is a function of PERCEPTION resolution. If the resolution is very fine, the view frame can be wider, keeping the same error allowance.

An interesting point is that the processing time of the PERCEPTION is a function of the resolution. Therefore there are two typical strategies for equal error allowance. One is a seeing a closer area with rough resolution in a short time. The other is a seeing a farther area with fine resolution in a longer time. The PERCEPTION MONITOR decides the optimal view frame and the resolution which allows the vehicle to run at maximum speed.

Figure 9 shows the relation between the velocity (V) and the distance the vehicle can run in one cycle (perception distance, D) for several error allowances. We got this result from two relations: the relation of resolution and the maximum perception distance for equal error allowance, and the relation of the resolution and the processing time (T). Velocity is defined as $V=D/T$.

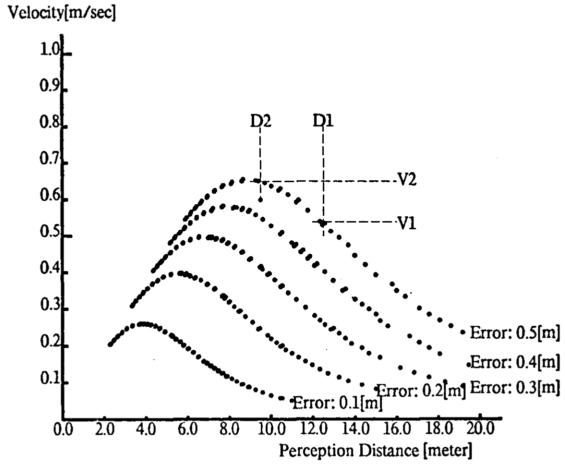


Figure 9: Distance and Velocity by Error Allowance

It is interesting to note that Figure 9 indicates that the maximum speed point of our system is at a rather short distance and low resolution. When the vehicle runs at a speed of V_1, the best perception distance and resolution is at D_1. This is because the PERCEPTION can see the widest area at that point. This increases the chance of finding landmarks or obstacles. Then, if the vehicle trys to run faster (at speed of V_2), the perception distance becomes shorter (D_2). This shows that as speed increases, the vehicle becomes short-sighted.

The PERCEPTION MONITOR has a data table in which are stored the values of optimal resolution and perception distance for the error allowance. Using this table, the PERCEPTION MONITOR provides the optimal resolution and perception distance.

3.3.3. Decision of Image Input Position

The PILOT indicates which region the PERCEPTION should see, but does not indicate when the PERCEPTION should see it. This is because the view frame of the PERCEPTION depends on the sensor used, and the PILOT does not know which sensor will be used. Furthermore, when the PERCEPTION uses a pan and tilt mechanism, only the PERCEPTION can decide image input timing and position.

The position decision algorithm has two steps. First, this module simulates the view frame and the vehicle's future path which is posted in the BLACKBOARD by the LOCAL PATH PLANNER. When the simulated view frame covers the region which the PILOT has requested the PERCEPTION to see, this vehicle position is defined as the image input position. Second, this module monitors current vehicle position by watching the moving vehicle position on the BLACKBOARD. And, when the moving vehicle position reaches the image input position, this module controls sensors to take an image.

3.3.4. Creation of Segmentation Parameters

To have good segmentation results, we need not only appropriate segmentation parameters, but also good algorithms. It is very difficult to know appropriate parameters unless PERCEPTION has scene knowledge. In this system, the PILOT predicts objects for the PERCEPTION, but the description is general and does not indicate segmentation parameters for the objects.

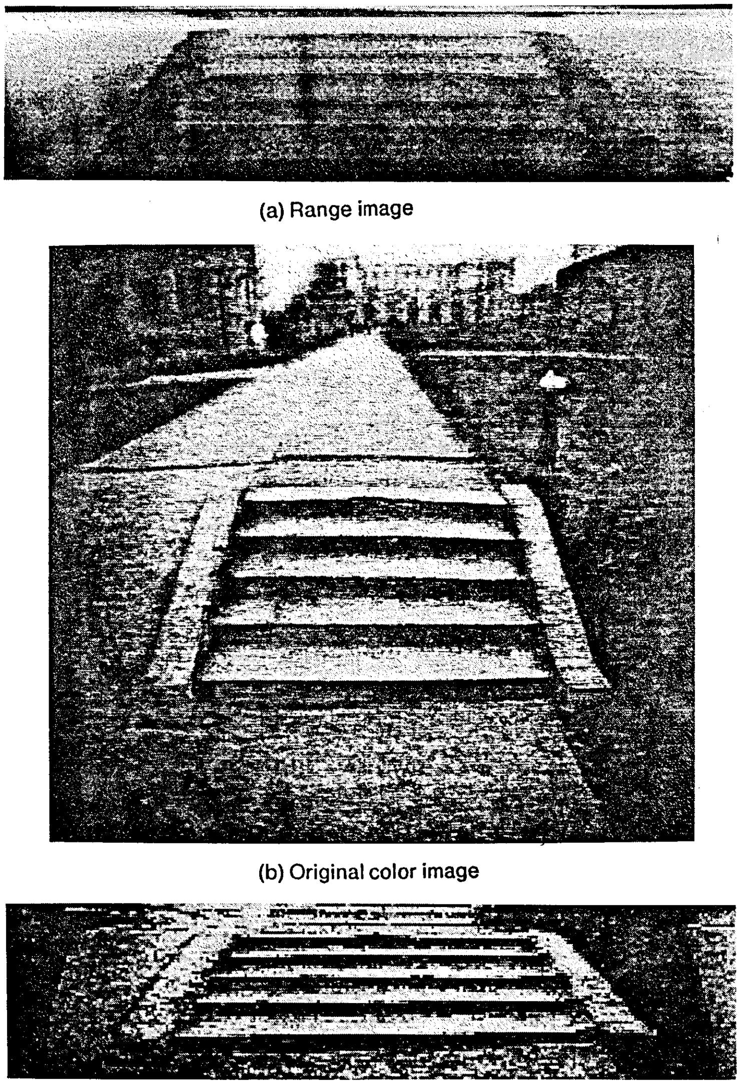
THE PERCEPTION MONITOR creates appropriate parameters, and the segmentation modules use them. Currently, the PERCEPTION MONITOR creates color type and minimum area of segments as segmentation parameters. For example, if the predicted objects are only a *sidewalk* and *grass*, the PERCEPTION MONITOR decides that types of color are *GRAY* and *GREEN*, and reports them to the segmentation module (the PATCH MAKER). Another example is the minimum area for segments. If there are no small predicted objects, the PERCEPTION MONITOR sets a rather large value for minimum segment area. This eliminates the noisy segments, and makes a simple segment list which is easy to analyze in the later object finding phase.

3.4. Colored-Range Image Analysis

It is very difficult to recognize complex objects in outdoor scenes using only one kind of sensor, but several different sensors can provide a lot of clues about the environment. For example, use of both range data and color images provides a very powerful vision system for outdoor scene analysis, because range data provide information about the geometry of a scene, and color images provide an important physical property of objects. In order to use these different sensor data, we must integrate them using sensor fusion techniques. The registration between range data and color images can be a first step to sensor fusion. In the following sections, we describe the registration algorithm for color and range image, the segmentation procedure for range data, the color segmentation algorithm, and how to use a *colored-range* image to recognize stairs and slopes.

3.4.1. Registration

We must know the relative positions and orientations of the color camera with respect to the range scanner to register range data into camera-centered coordinates. These can be computed using a calibration procedure. The calibration step consists of initial value estimation and optimum value finding procedures. In the initial value estimation step, the measured angles are used to simplify the problem, so the position of the camera relative to the range scanner and its focal length are the only unknown parameters. The unknown parameters are computed by solving a least-squares criterion. Once the initial values of the camera parameters are computed, we can obtain more accurate parameters using a modified Newton-Gauss method. In the experiment, the wide-angle lens is used to capture color images, which prevents us from using a linear perspective transformation for projection. Use of a third-order polynomial for projection provides good registration. The projection model is developed through a conventional camera calibration procedure. Since the camera/scanner transformation and the perspective transformation are computed, the range data can be registered with the color image. This is done by applying the transformation to each point of the range image, so the corresponding pixel point in the color image can be computed. The color intensity for the computed pixels in color image then are assigned to the range image. The registration procedure is illustrated in Figure 10.


(a) Range image

(b) Original color image

(c) *Colored-range* image

Figure 10: Colored-range Image Registration

3.4.2. Range Image Segmentation

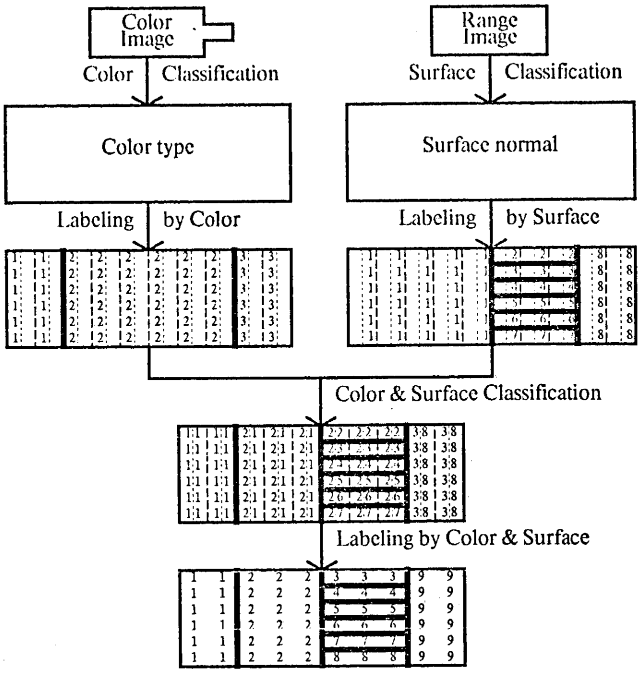
The range segmentation module generates surface segments using three basic attributes at each pixel: jump edges, surface normal, and surface curvature. Each segmented region has a surface normal vector, a surface curvature, 3-D edges, and a label indicating smooth region or rough region. The detailed algorithm for range data analysis can be found in Hebert's recent work [2].

3.4.3. Color Segmentation

The color segmentation module uses Wallace's color classification algorithm [5] which uses a standard quadratic discrimination function for multivariate normal distributions of mean vectors and covariance matrices for the Red, Green, and Blue components of an image. This module cannot detect 3-D position data for segments. However, when we assume all objects are on a flat ground plane, this module can calculate 3-D values for segments using the relation between the perspective image and the scene [1].

3.4.4. Overlap Segmentation

The registration process creates an image in which each pixel has a color value and a range value. The segmentation for the *colored-range* image is done as illustrated in Figure 11. First, the color label and the surface label are obtained by color segmentation and surface segmentation. Then, the color-surface label for each pixel is obtained by an AND operation of the color label and the surface label. Overlap segmentation is done using this label value.



1	1	2	2	2	2	2	2	3	3
1	1	2	2	2	2	2	2	3	3
1	1	2	2	2	2	2	2	3	3
1	1	2	2	2	2	2	2	3	3
1	1	2	2	2	2	2	2	3	3
1	1	2	2	2	2	2	2	3	3

1	1	1	1	1	2	2	2	8	8
1	1	1	1	1	3	3	3	8	8
1	1	1	1	1	4	4	4	8	8
1	1	1	1	1	5	5	5	8	8
1	1	1	1	1	6	6	6	8	8
1	1	1	1	1	7	7	7	8	8

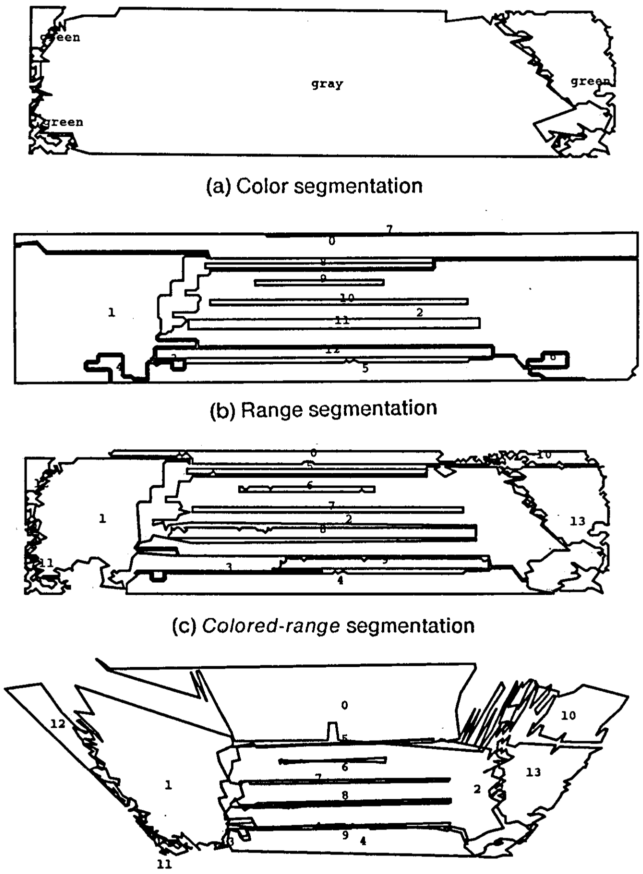
1:1	1:1	2:1	2:1	2:1	2:2	2:2	2:2	3:8	3:8
1:1	1:1	2:1	2:1	2:1	2:3	2:3	2:3	3:8	3:8
1:1	1:1	2:1	2:1	2:1	2:4	2:4	2:4	3:8	3:8
1:1	1:1	2:1	2:1	2:1	2:5	2:5	2:5	3:8	3:8
1:1	1:1	2:1	2:1	2:1	2:6	2:6	2:6	3:8	3:8
1:1	1:1	2:1	2:1	2:1	2:7	2:7	2:7	3:8	3:8

1	1	2	2	2	3	3	3	9	9
1	1	2	2	2	4	4	4	9	9
1	1	2	2	2	5	5	5	9	9
1	1	2	2	2	6	6	6	9	9
1	1	2	2	2	7	7	7	9	9
1	1	2	2	2	8	8	8	9	9

Figure 11: Overlap Segmentation by Color and Range Data

3.4.5. Result of a Real Scene Analysis

One good example to show the effectiveness of the *colored-range* segmentation module is a slope and stairs scene of a campus sidewalk. The slope and the stairs are made of concrete and have the same gray color. The slope and road-side grass are on almost the same plane. Therefore, segmentation using only color can not separate the slope and the stairs, and segmentation using only range can not separate the slope and the road-side grass. Overlap segmentation using *colored-range* image can extract the slope which is only navigable region in this scene. Figure 12 shows the results of color segmentation, range segmentation, and colored-range segmentation drawn with polygons.



(a) Color segmentation

(b) Range segmentation

(c) *Colored-range* segmentation

(d) *Colored-range* segmentation projected on x-y plane

Figure 12: Colored-Range Segmentation

4. Conclusion

We have developed the Sidewalk Navigation System which can drive the vehicle on the test site, the CMU campus. Because our test site involves sidewalks and intersections, slopes, stairs, and grass, the navigation system should have ability to select the best navigation mode depending on the situation. Our architecture works well in this environment, using both route planning and local path planning, selecting vehicle driving mode, and selecting sensors. The scene which includes the stairs, the slope, and the grass is hard to detect with only a TV camera or only a range sensor. But our perception module can analyze even this scene using sensor fusion with color and range data. And sensor selection dependent object prediction saves computation time. Our current system demonstrates the framework of an outdoor robot navigation system.

Our future work will include pipeline processing in the PILOT module, expanding the PERCEPTION module to detect other objects, and map revising. And we will add a capability for *cross-country navigation*.

Acknowledgements

The CMU Sidewalk Navigation System is created as a part of the Strategic Computer Vision Project at CMU. This system includes several programs produced by project members. The BLACKBOARD was produced by Steven Shafer, Anthony Stentz, and Charles Thorpe [4]. The color classification program was based on the code made by Richard Wallace [5]. The range image segmentation program was made by Martial Hebert [2]. The system architecture design is based on the proposal by Steven Shafer, Anthony Stentz and Charles Thorpe [4].

We would like to thank Regis Hoffman for creating the LOCAL PATH PLANNER, Paul Allen for writing the NAVIGATION MONITOR, and Ralph Hyre for useful blackboard facilities. Thanks to Mike Blackwell, Kevin Dowling, and James Moody for hardware, system interface, and video support. The civil engineers Red Whittaker, Chuck Whittaker, Francois Bitz, Steve Berman, and Kai Lee developed and maintain the Terregator vehicle.

References

[1] Wallace, R., Matsuzaki, K., Goto, Y., Crisman, J., Webb, J., Kanade, T.
Progress in Robot Road-Following.
In *Proceedings 1986 IEEE International Conference on Robotics and Automation*, pages 1615-1621. April, 1986.

[2] Hebert, M.
Outdoor Scene Analysis Using Range Data.
In *Proceedings 1986 IEEE International Conference on Robotics and Automation*, pages 1426-1432. April, 1986.

[3] Lozano-Perez.
An Algorith for Planning Collision Free Paths Among Polyhedral Obstacles.
In *Communications of the ACM*. October, 1979.

[4] Stentz, A., Shafer, S., Thorpe, C.
An Architecture for Sensor Fusion in an Autonomous Land Vehicle.
In *Proceedings 1986 IEEE International Conference on Robotics and Automation*, pages 2002-2011. April, 1986.

[5] Wallace, R.
Robot Road Following by Adaptive Color Classification and Shape Tracking.
In *Forthcoming in Proceedings 1986 AAAI*. 1986.

ERROR MODELLING IN STEREO NAVIGATION

Larry Matthies and Steven A. Shafer

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

In stereo navigation, a mobile robot estimates its position by tracking landmarks with on-board cameras. Previous systems for stereo navigation have suffered from poor accuracy, in part because they relied on scalar models of measurement error in triangulation. This paper shows that using 3-D gaussian distributions to model triangulation error leads to much better performance. The paper describes how to compute the error model from image correspondences, estimate robot motion between frames, and update the global positions of the robot and the landmarks over time. Simulations show that compared to scalar error models the 3-D gaussian reduces the variance in robot position estimates and better distinguishes rotational from translational motion. A short indoor run with real images supported these conclusions and computed the final robot position to within 2% of distance and one degree of orientation. These results illustrate the importance of error modelling in stereo vision for this and other applications.

1 Introduction

Consider a robot given the task of going from A to B. At a coarse level its route is planned from a pre-stored map, while at a fine level the route is determined by sensor information gathered along the way. Incremental motion estimates are integrated to keep track of the robot's position in the map, which in turn is used to predict upcoming landmarks, hazards, or arrival at the destination.

To realize this scenario, a robot needs sensors that can measure its position and detect the presence of 3-D objects nearby. Stereo vision can provide both kinds of information. Stereo matching at one point in time provides a local 3-D model for route planning and obstacle avoidance. Selected points in this model become landmarks that are tracked by the stereo system to monitor the robot's progress. Using to stereo in this way, to detect nearby objects and to estimate the motion of the robot, is what we refer to as stereo navigation.

We are interested in stereo in this scenario for a number of reasons. First, other motion sensors can be in error, such as shaft encoders when wheels slip or loose contact with the ground. Second, other sensors, such as sonar and radar, can be inappropriate for reasons of concealment, possible confusion with the broadcasts of other robots nearby, or because color and reflectivity information are important. Lastly, we are interested in stereo *per se* and believe that methods developed for this domain can be transferred to other applications.

Methods for extracting shape and motion information from image sequences can be classified as correspondence-based or flow-based. Correspondence methods [7, 11, 18, 24] track distinct features such as corners and lines through the image sequence and compute 3-D structure by triangulation. Flow-based methods [1, 25] treat the image sequence as function $\mathbf{I(x,y,t)}$ of row, column, and time, restrict the motion between frames to be small, and compute shape and motion in terms of differential changes in $\mathbf{I}$. This paper deals with error modelling issues in the correspondence paradigm.

One of the first systems for correspondence-based stereo navigation was that built by Moravec [18]. This system moved a robot in a stop-go-stop fashion, digitizing and analyzing images at every stop. Features were matched in stereo images to build a world model consisting of 3-D points. After moving and acquiring more images, the points in the world model were matched in the new images to find their coordinates relative to the new robot location. A least squares procedure was applied to the differences between the new and point locations to infer the actual motion of the robot. The contribution of each landmark point to this motion estimate was multiplied by a scalar weight that varied inversely with the distance to the point.

In earlier work with Moravec [17], we found the motion solving part of this system to be somewhat inaccurate and unstable. This has been a common experience with visual motion solving algorithms in general. In the case of correspondence-based algorithms, this can partly be attributed to inadequate modelling of measurement error in triangulation. In triangulation, 3-D structure is computed by finding the intersection of rays projected through corresponding points in two images. Small errors in locating the matching image points changes the direction of these rays and leads to errors in the inferred 3-D location of the point. Scalar weights do not adequately model this error and lead to poor performance in subsequent motion computations based on the point's location. More sophisticated methods have been used in a number of places. In photogrammetry [20], 2-D and 3-D normal distributions are used to model error in image coordinates and 3-D point locations, respectively. Gennery [11] has used 2-D normal

This research was sponsored by the Office of Naval Research under contract N00014-81-K-0503 and by the Defense Advanced Research Projects Agency under contracts DACA 76-85-C-0003 and F33615-84-K-1520. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ONR, DARPA, or of the U.S. Government.

distributions of image coordinates in camera calibration for computer vision. .Hallam [15] used normal error models in conjunction with Kalman filters to track points and estimate robot motion from sonar data. Broida and Chellappa [5] used similar methods to track a known object in monocular image sequences. This paper shows how these methods can be applied to stereo navigation and demonstrates that they lead to markedly better performance.

The system described in this paper is shown in figure 1. The structure is very similar to that used in [18]. The main data structures are a set of 3-D points P_i, called the local model and described in robot-centered coordinates, and the robot's current estimate of its position in some fixed, global reference frame. The points in the local model are obtained by stereo matching are used as landmarks. When a new stereo pair is digitized, points from the local model are matched in the images to determine their current locations Q_i relative to the robot. A motion solving algorithm estimates the rotation and translation (R and T) relating the new and old coordinates. The model updating system transforms the old local model into the current coordinate frame and combines it with the new points to create a new local model. Finally, the motion estimate is used to update the robot's global position. The cycle then repeats with the acquisition of a new pair of images.

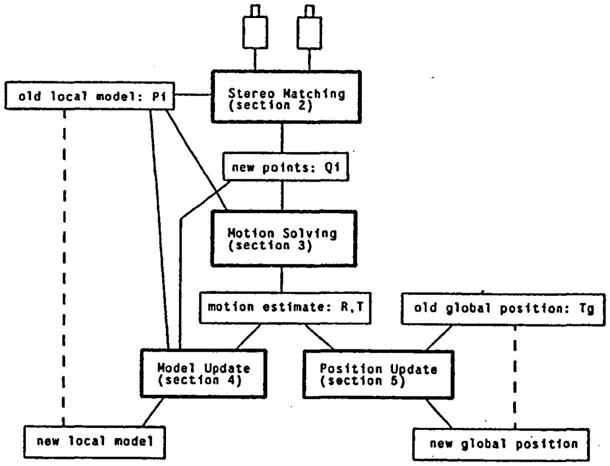


Figure 1: System block diagram

Section 2 shows how to model triangulation error in the stereo matcher with 3-D normal distributions. In section 3 this is incorporated in an algorithm for finding the rotation and translation between successive stereo pairs. The covariance matrix of this transformation is used in section 4 to update the local model with Kalman filters and in section 5 to estimate the robot's global position uncertainty. Simulations described in section 6 show that compared to scalar error models this system reduces the variance of position estimates and better distinguishes rotational motion from translation. An experiment with real images, using 54 stereo pairs covering 5.4 meters and fully automatic feature tracking, supported these conclusions and computed the final robot position to within 2% of distance and one degree of orientation. Conclusions are summarized in section 7.

2 Modelling stereo triangulation error

The geometry of stereo triangulation is shown in figure 2. For the moment we consider just the case of 2-D points projecting onto 1-d images. Two cameras are placed at offsets of ±b from a coordinate system centered between the cameras. Suppose point P projects onto the left image at x_l and the right image at x_r. Because of errors in measurement, the stereo system will determine x_l and x_r with some error. The error can come from many sources, including quantization of the image, photometric and geometric distortion in the camera, and the effects of perspective distortion on the matching algorithm. This error in turn causes the true location of P to be inferred with some error. Figure 2 illustrates this for errors caused by image quantization; because of resolution limits, the estimated location of P can lie anywhere in the shaded region surrounding the true location [22]. Additional random effects will cause this region to have less sharp boundaries, but the general shape will be similar. We want to take this uncertainty into account in any reasoning based on measurements of P.

Three approaches to modelling such uncertainty are discrete tolerance limits, scalar weights, and multi-dimensional probability distributions. Tolerance regions are often used. in object recognition and motion planning [4, 6, 14]; however, they are inappropriate for our application because of the combinatorial nature of the algorithms they require, the stochastic nature of matching errors, and the need to filter time sequences of data.

The idea behind scalar weights is that uncertainty grows with distance, so it can be modelled by weighting points inversely with distance [18]. However, as figure 2 shows, the uncertainty induced by triangulation is not a simple scalar function of distance to the point; it is also skewed and oriented. Nearby points have a fairly compact uncertainty, whereas distant points have a more elongated uncertainty that is roughly aligned with the line of sight to the point. Scalar error measures do not capture these distinctions in shape.

Normal distributions are commonly used in photogrammetry [20] and navigation [10, 26] to model uncertainty in two and three dimensional data. In computer vision, they have been used to model error in coordinates of image correspondences [11], monocular object tracking [5, 12], navigation and tracking with sonar [15], and recently in stereo work similar to ours [9]. To model triangulation error, we begin by treating image coordinates as corrupted by 2-D, normally distributed (ie. gaussian) noise and derive from this a distribution of the error in the inferred 3-D coordinates. Because triangulation is a non-linear operation, the true 3-D distribution will be non-gaussian. We approximate this as gaussian because it is simpler and gives an adequate approximation when the distance to points is not extreme. We will discuss shortly the cases where this breaks down.

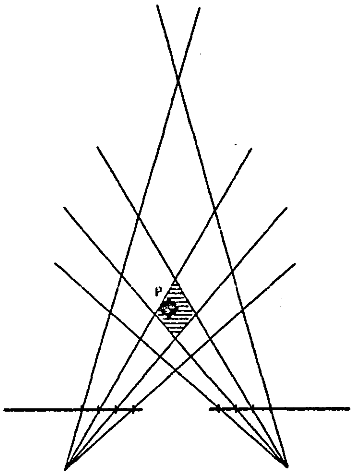


Figure 2: Stereo geometry showing triangulation uncertainty

We will now show the details of the triangulation and error model calculation for the general case of 3-D points projecting onto 2-D images. Let the image coordinates be given by $l = [x_l, y_l]$ and $r = [x_r, y_r]$ in the left and right image, respectively. Consider these as normally distributed random vectors with means μ_l and μ_r and covariance matrices V_l and V_r, respectively. From l and r we need to estimate the coordinates $[X, Y, Z]^T$ of the 3-D point P. We take the simple approach of using the ideal, noise-free triangulation equations $P = [X, Y, Z]^T = f(l, r)$ or

$$X = b(x_l + x_r)/(x_l - x_r)$$

$$Y = b(x_l + x_r)/(x_l - x_r) \qquad (1)$$

$$Z = 2b/(x_l - x_r)$$

(assuming a unit focal length) and inferring the distributions of X, Y, and Z as functions of random vectors l and r. If equation ((1)) was linear, P would be normal [8] with mean $\mu_p = f(\mu_l, \mu_r)$ and covariance

$$V_p = J \begin{bmatrix} V_l & 0 \\ 0 & V_r \end{bmatrix} J^T \qquad (2)$$

where J is the matrix of first partial derivatives of f or the Jacobian. Since f is nonlinear these expressions do not hold exactly, but we use them as satisfactory approximations.

The true values of the means and covariances of the image coordinates needed to plug into (1) and (2) are unknown. We approximate the means with the coordinates returned by the stereo matcher and the covariances with identity matrices. This is equivalent to treating the image coordinates as uncorrelated with variances of one pixel. Better covariance approximations can be obtained by several methods [2, 11].

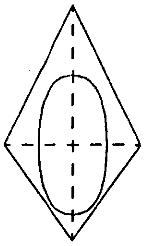


Figure 3: Quantization error with normal approximation

What does this error model mean geometrically? Constant probability contours of the distribution of P describe ellipsoids about the nominal mean that approximate the true error distribution. This is illustrated in figure 3, where the ellipse represents the contour of the error model and the diamond represents quantization error of figure 2 For nearby points the contours will be close to spherical; the farther the points the more eccentric they become. A covariance matrix with structure $V = wI$, equal to a scalar times the identity matrix, describes only spherical contours. This is the difference between attaching scalar weights to 3-D coordinate vectors and using the full 3-D distribution; that is, scalar weights are equivalent to spherical covariances whereas the full distribution permits ellipsoidal covariances. In the balance of the paper we will often refer to scalar weights as a spherical error model and the full distribution as an ellipsoidal error model.

Where the gaussian approximation breaks down is in failing to represent the longer tails of the true error distribution. The true distribution is skewed not unlike the diamond in figure 3, whereas normal distributions are symmetric. The skew is not significant when points are close, but becomes more pronounced the more distant the points. A possible consequence is biased estimation of point locations, which may lead to biased motion estimates. We will return to this issues section 6.

3 Solving for robot motion

The previous section showed how to model measurement error in stereo triangulation. In this section we show how to incorporate the error model into an algorithm for estimating the motion between successive stereo pairs. We will begin by showing how motion is computed with scalar weights, then derive an algorithm based on the 3-D gaussian error model, and finally give this algorithm a geometric interpretation.

Refering back to figure 1, at this stage in the cycle the robot has two sets of 3-D points that have been obtained by stereo matching: a local model of points P_i defined relative to its previous position and the coordinates Q_i of these points relative to its current position. The correspondences between P_i and Q_i are known, but the motion between them is not. Parameterizing the rotation in terms of Euler angles, we have a set of equations

$$Q_i = RP_i + T$$

in which P_i and Q_i are known point vectors, R is the matrix of the unknown rotation, and T is the unknown translation.

Using scalar weights, one finds R and T by expressing the errors of fit by

$$\varepsilon_i = Q_i - RP_i - T$$

minimizing the weighted sum of squares

$$\sum_{i=1}^{n} w_i \varepsilon_i^T \varepsilon_i \qquad (3)$$

where w_i are the weights. Although the rotation makes this optimization problem nonlinear, it has a closed form solution [19]. A solution for case where the rotation is parameterized by quaternions is given in [16].

As will be shown in section 6, the scalar model of uncertainty embodied in equation (3) leads to poor performance. Using the 3-D gaussian error model the solution takes a similar, but more complicated form. For simplicity we begin with the case of translational motion. The simplified motion equation is

$$Q_i = P_i + T$$

which we may rewrite as

$$Q_i - P_i = M_i = T$$

to emphasize the role of $M_i = Q_i - P_i$ as measurements of T. From section 2, P_i and Q_i are modelled as normally distributed random vectors with covariances U_i and V_i, respectively. Therefore, M_i will also be normally distributed with covariance $U_i + V_i$. Now if we consider M_i to be a sequence of noisy measurements of T, each corrupted by noise with zero mean and covariance $U_i + V_i$, application of the maximum likelihood method leads to minimizing the following expression over possible values of T [8]:

$$\sum_{i=1}^{n} \varepsilon_i^T W_i \varepsilon_i \qquad (4)$$

where $\varepsilon_i = M_i - T$ and $W_i = (U_i + V_i)^{-1}$. The solution to this is

$$T = (\Sigma_{i=1}^{n} W_i)^{-1} \Sigma_{i=1}^{n} W_i M_i$$

and the covariance matrix of the estimation errors is

$$V_T = (\Sigma_{i=1}^{n} W_i)^{-1}$$

The covariance matrix can be analyzed to assess the quality of the motion estimate. It is also used later in modelling the uncertainty of the robot's global position estimate.

An intuitive interpretation of equation ((4)) is shown in figure 4. The weight matrices W_i function as norms that measure distance differently for each point. Error vectors making equal contributions to the total error of fit lie on ellipsoidal contours. For example, in figure 4, residuals ε_a and ε_b contribute equally to the total error but ε_c contributes more because $\varepsilon_a^T W \varepsilon_a = \varepsilon_b^T W \varepsilon_b < \varepsilon_c^T W \varepsilon_c$. This effectively gives more weight to errors perpendicular to the line of sight than parallel to it, which, given the nature of stereo, is what we would like to do. The "spherical" error model obtained by using the scalar weights of equation (3) has the obvious mnemonic meaning that residual vectors making equal contributions to the total error lie on spherical contours. This distinction is what gives the ellipsoid model its power.

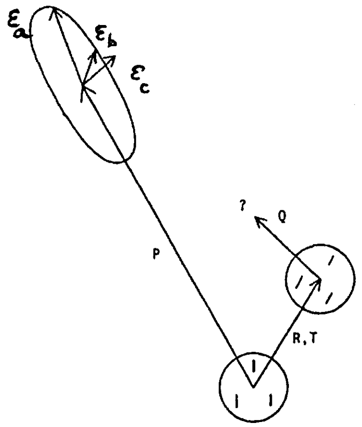


Figure 4: Interpretation of weight matrix

Generalizing this method to handle rotation is complicated by the fact that the equations become nonlinear. The function to be optimized takes the form

$$\sum_{i=1}^{n} \varepsilon_i^T W_i \varepsilon_i \qquad (5)$$

$$\textit{with } \varepsilon_i = Q_i - R P_i - T$$
$$\textit{and } W_i = (R U_i R^T + V_i)^{-1}$$

We have not been able to find direct solutions to this problem or even to approximations in which W_i is not a function of R. Our approach has been to use the direct solution for scalar weights to get an initial approximation, then to iterate on linearizations of equation (5). Linearization methods for solving least squares problems are described in [13].

To recap, this section incorporated the error model of section 2 in an algorithm for finding the rotation and translation between two 3-D points sets. The algorithm replaces the scalar weights of equation (3) with weight matrices based on the covariances of corresponding points. When the motion is purely translational, the problem is linear and has a direct solution, but when the motion involves rotation we resort to an iterative solution. The error covariance of the motion solution will be used in the following two sections in updating the robot's local model and global position estimate.

4 Updating the local model

So far we have described how to model error in triangulation and how to solve for the motion between two successive stereo pairs. This section deals with how to process a long sequence of stereo pairs. At issue is how to average information from successive images to achieve more accurate landmark localization and consequently more accurate estimates of robot position.

An appropriate tool for this is the Kalman filter [10]. In filtering terminology the quantity to be estimated is called the "state", and when a measurement is taken the filter updates the current estimate of the state. Kalman filters incorporate known statistical properties of the measurements into the update process and produce error covariances for the state estimate. They are widely used in terrestrial and aerospace navigation and guidance applications [10, 26]. In computer vision they have been used in object recognition [3], tracking of known objects with monocular image sequences [5, 12], and for robot navigation and object tracking with sonar data [15].

In our application, the state consists of the locations of the landmark points in the local model. A question arises as to whether the landmarks should be represented in a global, stationary frame of reference or in a local, moving, robot-centered frame. In either case, the update involves transforming coordinates from one frame to the other and applying the filter. If a fixed number of landmarks are being tracked, there is no difference in cost between the two. There will be a difference in the uncertainty of the resulting model; this difference depends on the relative uncertainties of the old model, the new measurements, and the intervening motion. We have not completed an analysis of this situation, but are currently keeping the landmark model in robot-centered coordinates.

The update involves transforming the old local model to the current coordinate frame, inflating its uncertainty to account for the uncertainty of the transformation, and filtering the old model with the new measurements to create the updated model. Let P_{t-1} be the coordinate vector of a single point in the old local model at time $(t-1)$ and let V_{t-1} be its covariance. For purely translational motion, P_{t-1} is transformed to the current frame by

$$\mathcal{P}_{t-1} = P_{t-1} + T \qquad (6)$$

where T is the translation from time $(t-1)$ to time t. The translation has an error covariance matrix V_T, so the transformed point has covariance

$$\mathcal{V}_{t-1} = V_{t-1} + V_T \qquad (7)$$

where V_T is the error covariance of the translation. Equation (6) introduces some correlation between points that is not accounted for in (7), but we assume this is small enough to ignore. To extend this to rotation, we rewrite equation (6) as

$$\mathcal{P}_{t-1} = R P_{t-1} + T \tag{8}$$

This is nonlinear, so to compute $\mathcal{V}_{t-1}$ we proceed by analogy to equation (2); that is, we pre-multiply the covariance of R, T, and P_{t-1} by the Jacobian of the transformation and post-multiply by the Jacobian transposed. Since we treat P_{t-1} as uncorrelated with R and T, this leads to

$$\mathcal{V}_{t-1} = J_m V_m J_m^{\ T} + R V_{t-1} R^T$$

where J_m contains the derivatives of (8) with respect to the motion parameters and V_m is the covariance of the motion parameters.

Now let Q_t be the measurement of the same point at time t and let U_t be the covariance of this measurement. Some manipulation of the basic Kalman filter equations leads to the following estimates of the updated point location and covariance:

$$V_t = (\mathcal{V}_{t-1}^{\ -1} + U_t^{-1})^{-1} \tag{9}$$

$$P_t = \mathcal{P}_{t-1} + V_t U_t^{-1}(Q_t - \mathcal{P}_{t-1}) \tag{10}$$

The intuition behind equation (10) is as follows. The second term takes the difference $(Q_t - \mathcal{P}_{t-1})$ of the new measurement from the old estimate, weights the difference by $V_t U_t^{-1}$, and applies the result as an update to the old estimate $\mathcal{P}_{t-1}$. Matrix U_t^{-1} will be "larger" the more precise the new measurement, giving it more weight in the update, and smaller the less precise the measurement, giving it less weight. Conversely, V_t will be small if the old estimate is precise and large otherwise. Hence if the old estimate is already good, the new measurement receives little weight; if it is poor, the new measurement receives more weight.

The procedure we have described assumes that the error in the motion estimate is uncorrelated with the error in the landmark points. When the motion estimate is obtained by using the methods of the previous section this will not be true, although if other sensors are also contributing to the motion estimate it will be approximately true. This is an issue we are investigating.

5 Updating the global robot position

By using the modules discussed in the previous sections, the robot computes estimates of its motion between successive stereo pairs. Combining these to estimate its global position is a simple matter of concatenating the transformation matrices. It may also be desirable to estimate the uncertainty of the global position, which can be done by propagating the covariance matrices of the incremental motions into a covariance of the global position. For translation this is also very simple. If the the global position at time $(t-1)$ is $T_{g_{t-1}}$ and the next incremental translation is T_t, then the next global position is

$$T_{g_t} = T_{g_{t-1}} + T_t \tag{11}$$

Since this is linear, if the incremental translation estimates have uncorrelated, zero mean gaussian errors, then T_{g_t} will also have zero mean, gaussian error with covariance given by

$$V_{g_t} = V_{g_{t-1}} + U_t$$

where $V_{g_{t-1}}$ and U_t the covariances of $T_{g_{t-1}}$ and T_t, respectively. The case of motion in the plane, there are two parameters for translation and one for rotation, has been dealt with by Smith and Cheeseman [21]. In summary, one obtains an equation analogous to (11) in which the three parameters of the global position are expressed as functions of the previous position and the incremental motion. These are nonlinear and error propagation is done by linearization. For general motion in three dimensions, this is not straightforward with the Euler angle representation of rotation we have used here. In this case other parameterizations of rotation, such as quaternions, may be preferable [9, 26]. We are exploring this further.

6 Performance

Our evaluation to date has concentrated on comparing the use of the spherical and ellipsoidal error models in the motion solving methods of section 3. Results of tests with simulated and real data are described below.

6.1 Simulations

Three sets of simulation data will be presented. The first set is a base case that compares the standard deviations of position estimates obtained with each error model for a single step of vehicle motion. That is, it considers motion between only two consecutive stereo pairs. It illustrates the difference in the variability of position estimates with each model and reveals different amounts of coupling between translation and rotation with each error model. The second set of data also considers only two consecutive stereo pairs and tests limiting performance by tracking progressively more distant points. The last set examines both long range performance over many images and the effect on performance of different stereo baselines.

The simulations were generated as follows. The "scene" consisted of random points uniformly distributed in a 3-D volume in front of the simulated cameras. Typically this volume extended 5 meters to either side of the cameras, 5 meters above and below the cameras, and from 2 to 10 meters in front of the cameras. The cameras themselves were simulated as having 512x512 pixels and a field of view of 53 degrees. The baseline for most simulations was 0.5 meters. Image coordinates were obtained by projecting the points onto the images, adding gaussian noise to the floating point image coordinates, and rounding to the nearest pixel. These coordinates were input to the triangulation and motion solving algorithms. For the ellipsoidal error model, covariance matrices were computed as described in section 2. In the scalar case, weights were derived by taking the Z variance from the covariance matrix. Scalars obtained by several other methods were tried and found to give very similar results. These include the volume and length of the major axis of the standard error ellipsoid and Moravec's half-pixel shift rule [18].

The first set of simulations determined the standard deviation of the estimated motion between two consecutive stereo pairs when the true motion was one meter. The results are shown in figures 5 and 6 plotted against the number of points used to compute the motion estimate. In both figures, the top three curves were obtained with spherical modelling and the bottom three with ellipsoid. Tilt implies rotation of the camera up or down, pan is the rotation about the vertical axis, and roll the rotation about the camera axis. The standard deviations obtained with the ellipsoidal model are considerably less than those with the spherical model. The size of this difference will vary; it will be larger when the 3-D points are further from the cameras and smaller when they are closer. This is because the spherical error model is a reasonable approximation to the triangulation error when points are close, but not when they are distant. Another thing to note is that with the spherical model roll and forward translation are estimated better

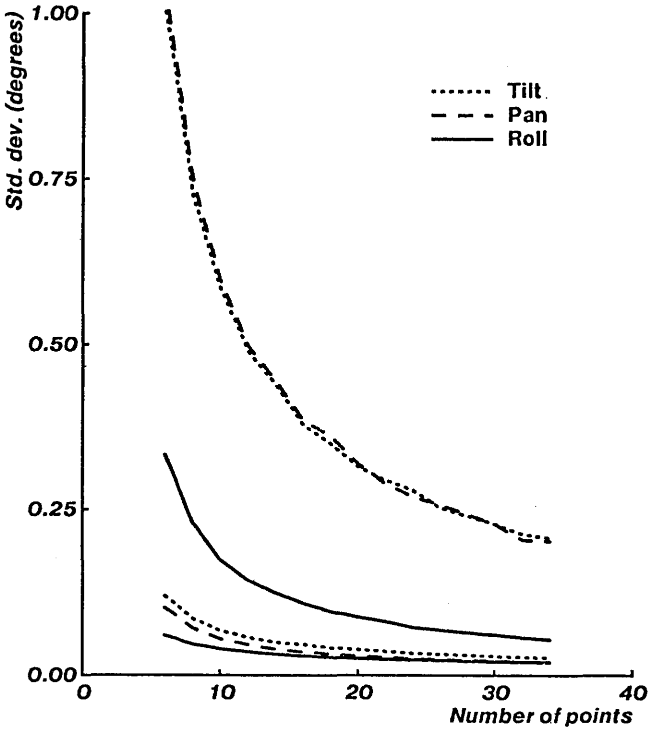


Figure 5: Simulation 1, rotations

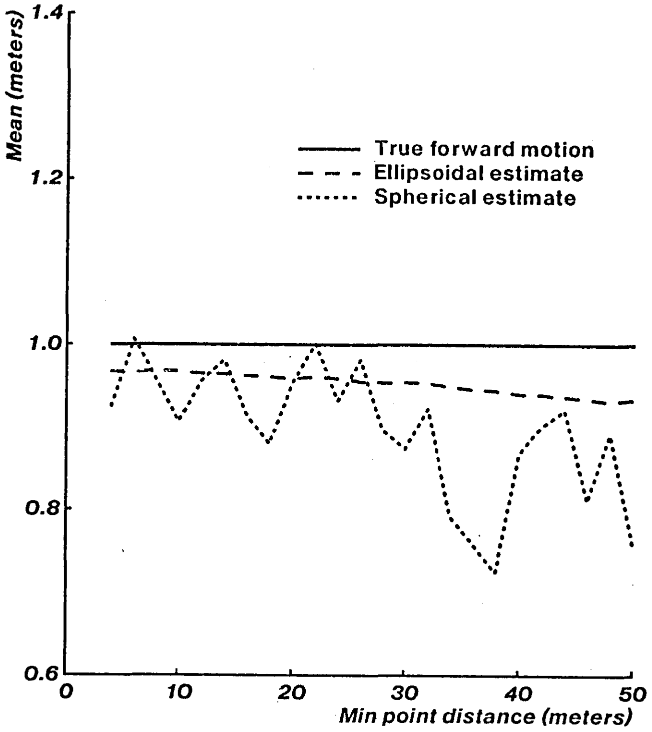


Figure 7: Simulation 2, bias

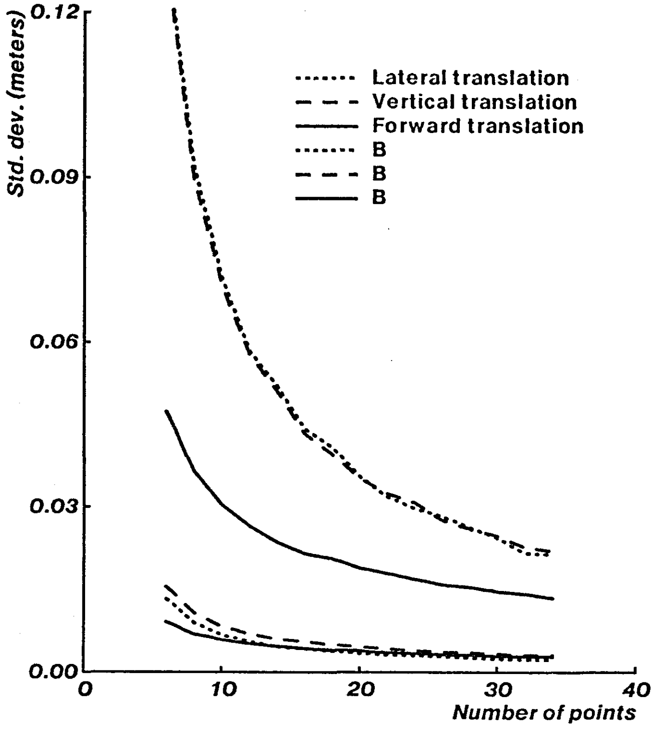


Figure 6: Simulation 1, translations

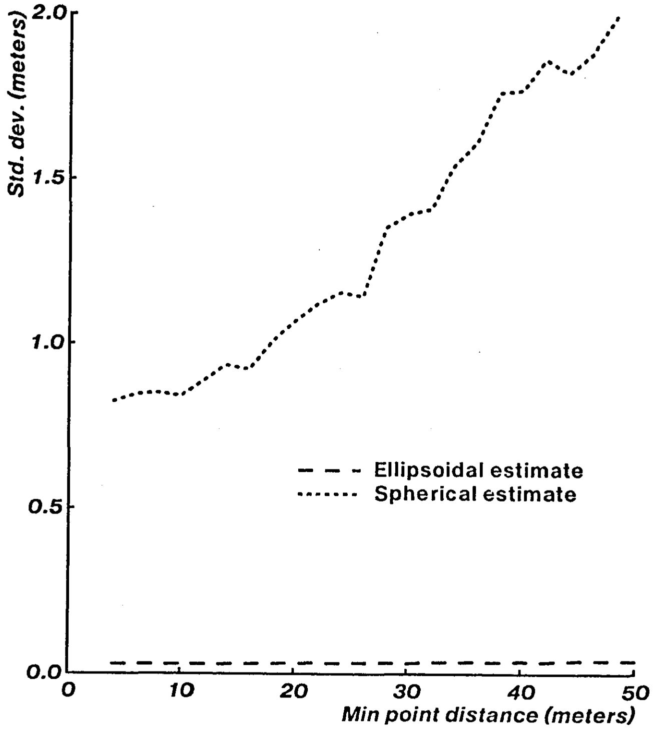


Figure 8: Simulation 2, standard deviation

than the other parameters, but with the ellipsoidal model all parameters are estimated equally well. This is because lateral translations and panning rotations have coupled effects on the errors of fit, as do vertical translations and tilting rotations. Using an ellipsoidal error model appears to reduce this coupling. Lastly, note that for a given level of performance fewer points are needed with the ellipsoidal model than the spherical, offseting the greater expense of the iterative motion solution needed in the ellipsoidal case. The exact relationship will depend on the camera configuration.

The second set of simulations also dealt with the estimated motion between just two stereo pairs. It examined the effect of increasing the distance to points in the scene, or equivalently to reducing the maximum disparity in the image. Figures 7 and 8 illustrate the results. Twenty points were generated in a volume spanning 4 to 50 meters in front of the cameras, giving disparities ranging from 2 to 32 pixels or 0.5% to 6% of image width. The volume was gradually shrunk by moving the near limit from 4 meters back until all points were 50 meters away, so that all disparities were on the order of 2 or 3 pixels. Figure 7 shows the mean value of the forward translation estimate as a function of the minimum distance to the points and figure 8 the standard deviation. The true forward motion was one meter. Looking at the means, with the ellipsoidal error model there is a consistent underestimation of the true motion that gets worse as the disparity shrinks. With the spherical error model the behavior is erratic. The jagged nature of the curve for the spherical model is due to the contribution of image quantization to the noise in the image coordinates. As a 3-D point moves smoothly away from the cameras, image quantization will lead the triangulation to alternately under- and overestimate the true distance to the point (see [22] for a good illustration). This in turn affects motion estimates based on tracking the point. Apparently the ellipsoidal model smooths out this effect. Figure 8 shows that the standard deviation of the motion estimates increases quite rapidly with shrinking disparity in the spherical case, but much less rapidly in the ellipsoidal case. On the whole, the breakdown with distance shown by the spherical error model is consistent with common experience in computer vision; this makes the stability shown with the ellipsoidal model come as quite a surprise.

Whereas the first two sets of simulations looked at motion estimates between only two consecutive stereo pairs, the last set looked at motion over a long sequence of images. There were two purposes for these simulations. The first was simply to confirm the results of the single-step simulations. The second was to test a hypothesis suggested by the previous simulation: that for equivalent performance, the ellipsoidal model may permit the use of a shorter stereo baseline than the spherical. This is an important consideration, because length of the baseline directly affects the difficulty of stereo matching. Figure 9 shows the standard deviation of the estimated distance as a function of the true distance. Here the simulated travel between images was 0.64 meters, so the figure represents about 90 simulated images. It shows curves for a 0.5 meter baseline with the spherical model and 0.125, 0.25, and 0.5 meter baselines for the ellipsoidal model. Comparing the curves for 0.5 meter baselines, the ellipsoidal model does outperform the spherical. It appears that the curves may eventually run parallel, so that the difference between the methods would be an additive constant rather than multiplicative. Looking at the effects of different baselines, results with the ellipsoidal model are still better than the spherical model with a 0.25 meter baseline, though not with 0.125 meters. Based on standard deviations of position, it does appear possible to use a shorter baseline. However, another factor involved is bias of the motion estimates. As seen in figure 7, increasing the ratio of object distance to baseline tends to cause motion estimates with both error models to underestimate the true distance. In general we have found that the narrower the baseline, the more motion is underestimated. The same occurs when we increase the variance of the simulated noise in the image coordinates. This appears to result from a net underestimation of the distance to points in space. Simple compensation schemes appear to work when only the only error in image coordinates comes from quantization, but are less adequate as the noise variance grows. This requires further investigation. For the moment we just note that bias can be a problem with short baselines or non-trivial noise levels.

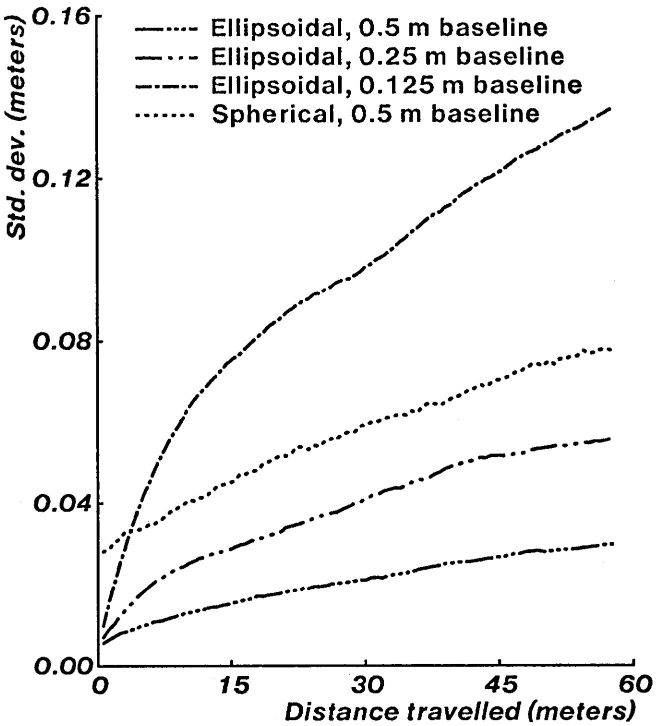


Figure 9: Simulation 3, effect of baseline

6.2 Real images

In order to verify the simulations on real images, we used both error models to estimate the position of a stereo-equipped robot travelling across the floor of our lab. The scene is pictured in figure 10. The robot was driven straight forward in 54 steps of slightly less than 10 centimeters each. The cameras were on a 20 centimeter baseline and had a 36-degree field of view. The FIDO feature-tracking system [23] was used to track points through the image sequence and the resulting set of matched image coordinates were input to the algorithms described earlier to estimte the robot's position at each step. We will briefly describe the operation of FIDO before discussing the results of the experiment.

FIDO uses the Moravec interest operator and coarse-to-fine correlation algorithm to pick and match point features in stereo pairs. The interest operator is applied to one image of a stereo pair to pick points in where intensity varies in all directions; typically these are sharp corners or intersections of lines. The correlator finds these points in the other image of the stereo pair. To find the same points in subsequent stereo pairs, an *a priori* motion estimate is used to predict the location of the point into the new images, a

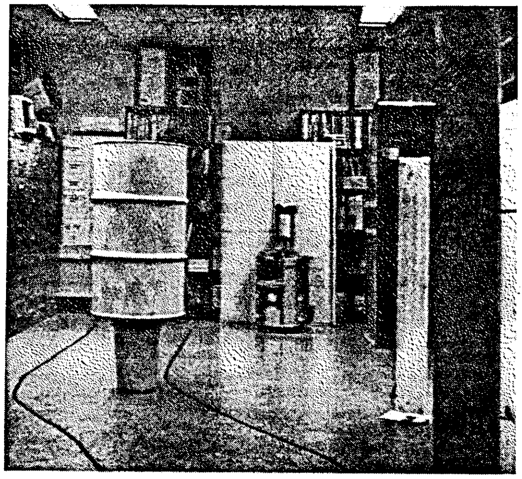


Figure 10: One image from lab sequence.

constraint window is defined around the predicted location based on the the uncertainty of the motion estimate, and the correlator is applied to find the position of best match within the constraint window. Incorrect matches are culled with a threshold on the correlation coefficient and with a 3-D error heuristic called the "3-D prune" stage. This heuristic uses the fact that under rigid motion the distance between two 3-D points does not change over time. Points which appear to violate this condition are discarded. The advantage of this test is that it does not require knowledge of the motion between stereo pairs. Points that survive this test become input to the motion solving algorithms. In the experiments to follow, between 30 and 40 points usually remained.

Figure 11 compares the true motion to the position estimates obtained with the spherical and ellipsoidal error models. For this figure a "planar" motion solver was used that solved only for the parameters of motion in the plane, that is two degrees of translation and one of rotation. The line of heavy dots shows the true position at every step, the path marked with circles shows the positions estimated with the spherical model, and the path marked with diamonds shows the same for the ellipsoidal model. The final position estimated with the ellipsoidal model was correct to within 2% of the distance and one degree of orientation. With the spherical model the corresponding figures are 8% and seven degrees.

In order to gauge the effect of noisier image matches, we adjusted the threshold of the prune stage so that progressivly fewer points were discarded. The general effect was to increasingly underestimate the distance travelled, which is consistent with the results of increasing the random noise level in the simulations. Figure 12 shows what happened when the prune stage was entirely disabled, leaving only the correlation threshold to detect matching errors. Estimates with the spherical model are initially very bad. We attribute this to matching errors caused by large depth discontinuities around the foreground objects. When these objects fall out of view, the estimates are better behaved. The behavior with the ellipsoidal model is much less erratic, though biased.

Finally, we repeated the first experiment (ie. clean data) with the algorithm that computes all six degrees of freedom of motion. The results were in accord with the planar case, with roughly the same levels of error in the final position estimate. It was notable that with the spherical model the error in roll was less than a degree, while in the other rotations it was between five and twelve degrees. This is consistent with the observation made from the first simulation about coupled rotation and translation.

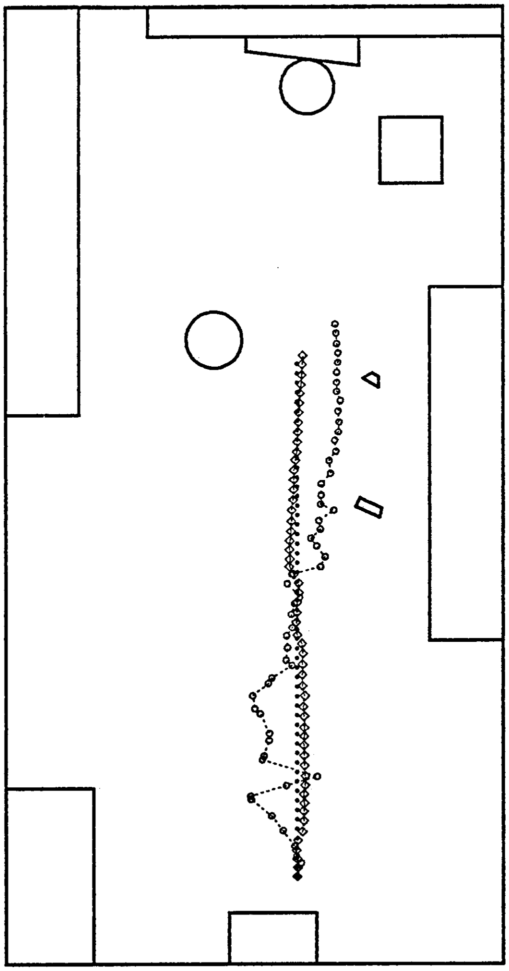


Figure 11: Results with clean data

7 Conclusions

Comparing motion estimates obtained with the spherical (scalar) and ellipsoidal (3-D gaussian) error models, under the relatively long object distance to baseline ratios we examined there is no question that the ellipsoidal model is preferred. Simulations showed that position estimates with the ellipsoidal model had less variance and live trials confirmed that they were more accurate and less influenced by matching errors. With short object distance to baseline ratios this distinction will diminish, and applications that can engineer this situation may be able to obtain satisfactory performance with the cheaper scalar error model. We suspect that many applications will be such that the 3-D gaussian model will be valuable.

A caveat to the results we have described is the possibility of bias leading to underestimation of position. This results from high noise levels in image match coordinates and from large distances to objects. We attribute this effect to the non-gaussian nature of the true error distribution in these situations. This can be dealt with either by ensuring that the noise level is low or by explicitly modelling the non-gaussian error. The low noise level can probably be achieved in many situations with the use of matching constraints, calculating match coordinates to sub-pixel resolution, and effective error detection methods. Where this cannot be achieved, better modelling is an area for further research.

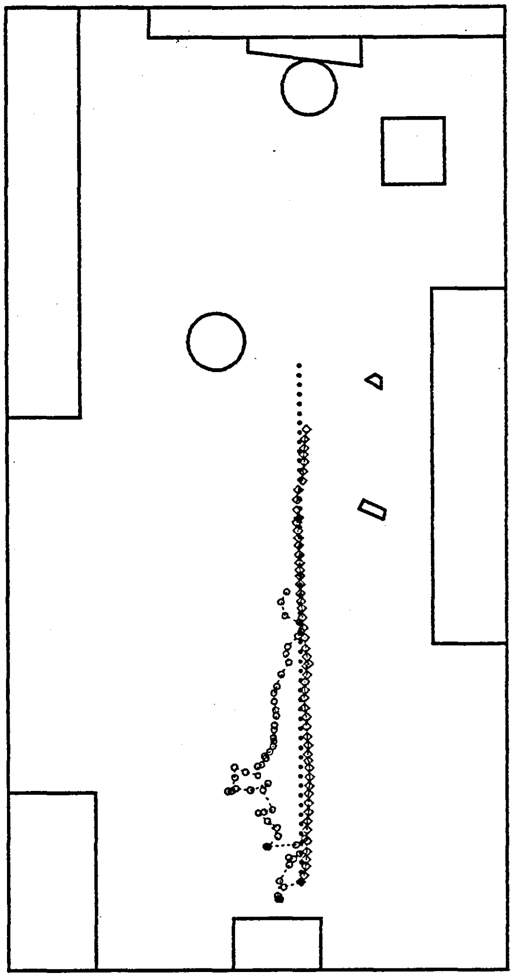


Figure 12: Results with noisy data.

There remains the question of whether use of ellipsoidal error models tolerates shorter baselines than use of spherical error models. To date we have only tested this in simulation. Based on variance of the position estimates, a shorter baseline is possible. However, the bias issue is unresolved.

Perhaps the most valuable result is demonstrating that accurate position estimates can be achieved in a fully automatic system when an adequate error model is used. The true motion in the examples we showed was pure translation, but we believe that the results will hold for general motion and preliminary simulations bear this out. With matching to sub-pixel resolution, matching of extended features instead of points, and more sophisticated error detection, it may be possible to obtain much better performance than that quoted here. Another interpretation of our results is that they show the importance of error modelling in stereo and probably other aspects of vision. One area we plan to explore this is in shape from stereo, beginning with the local update paradigm of section 5.

Acknowledgments

We are indebted to Hans Moravec for making us aware of Kalman filters, Peter Highnam for introducing us to Schonemann's algorithm, and Takeo Kanade for pointing out the possibility of using a shorter baseline.

References

[1] G. Adiv.
Determining three-dimensional motion and structure from optical flow generated by several moving objects.
IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-7(4):384-401, July, 1985.

[2] P. Anandan and R. Weiss.
Introducing a smoothness constraint in a matching approach for the computation of displacement fields.
In *Proc. of ARPA IUS Workshop*, pages 186-197. SAIC, December, 1985.

[3] N. Ayache and O.D. Faugeras.
HYPER: A new approach for the recognition and positioning of two-dimensional objects.
IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-8(1):44-54, January, 1986.

[4] H.S. Baird.
Model-based Image Matching Using Location.
MIT Press, Cambridge, Mass., 1985.

[5] T.J. Broida and R. Chellappa.
Estimation of motion parameters from noisy images.
IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-6(1):90-99, January, 1986.

[6] R.A. Brooks.
Symbolic reasoning among 3-D models and 2-d images.
Artifical Intelligence 17:285-348, 1981.

[7] L. Dreschler and H.-H. Nagel.
Volumetric model and 3D trajectory of a moving car derived from monocular TV frame sequences of a street scene.
Computer Graphics and Image Processing 20:199-228, 1982.

[8] T.F. Elbert.
Estimation and Control of Systems.
Van Nostrand Reinhold Co., New York, NY, 1984.

[9] O.D. Faugeras, N. Ayache, B. Faverjon, F. Lustman.
Building visual maps by combining noisy stereo measurements.
In *IEEE Int'l Conf. on Robotics and Automation*, pages 1433-1438. IEEE, April, 1986.

[10] A. Gelb (editor).
Applied Optimal Estimation.
MIT Press, Cambridge, MA, 1974.

[11] D.B. Gennery.
Modelling the environment of an exploring vehicle by means of stereo vision.
PhD thesis, Stanford University, June, 1980.

[12] D.B. Gennery.
Tracking known three-dimensional objects.
In *Proc. of AAAI*, pages 13-17. AAAI, 1982.

[13] P.E. Gill, W. Murray, and M.H. Wright.
Practical Optimization.
Academic Press, 1981.

[14] W.E.L. Grimson and T. Lozano-Perez.
Model-based recognition and localization from sparse range or tactile data.
International Journal of Robotics Research 3(3):3-35, Fall, 1984.

[15] J. Hallam.
Resolving observer motion by object tracking.
In *Int'l Joint Conf. on Artificial Intelligence*. 1983.

[16] M. Hebert.
Reconnaissance de formes tridimensionelles.
PhD thesis, L'Universite de Paris-Sud, Centre d'Orsay, September, 1983.

[17] L.H. Matthies and C.E. Thorpe.
Experience with visual robot navigation.
In *Proc. of IEEE Oceans'84 Conf.*. IEEE, Washington, D.C., August, 1984.

[18] H.P. Moravec.
Obstacle avoidance and navigation in the real world by a seeing robot rover.
PhD thesis, Stanford University, September, 1980.

[19] P.H. Schonemann and R.M. Carroll.
Fitting one matrix to another under choice of a central dilation and a rigid motion.
Psychometrika 35(2):245-255, June, 1970.

[20] C.C. Slama (editor-in-chief).
Manual of Photogrammetry.
American Society of Photogrammetry, Falls Church, Va., 1980.

[21] R.C. Smith and P. Cheeseman.
On the representation and estimation of spatial uncertainty.
Technical Report (draft), SRI International, 1985.

[22] F. Solina.
Errors in stereo due to quantization.
Technical Report MS-CIS-85-34, U. Pennsylvania, September, 1985.

[23] C.E. Thorpe.
Vision and navigation for a robot rover.
PhD thesis, Carnegie-Mellon University, December, 1984.

[24] R.Y. Tsai and T.S. Huang.
Uniqueness and estimation of three-dimensional motion parameters of rigid objects with curved surfaces.
IEEE Trans. on Pattern Analysis and Machine Intelligence 6(1):13-26, January, 1984.

[25] A.M. Waxman and J.J. Duncan.
Binocular image flows.
In IEEE (editor), *Proc. of Workshop on Motion: Representation and Analysis*, pages 31-38. May, 1986, May, 1986.

[26] J.R. Wertz (ed).
Spacecraft Attitude Determination and Control.
D. Reidel Publishing Company, 1978.

AUTOMATIC GRASP PLANNING: AN OPERATION SPACE APPROACH

Matthew T. Mason
Randy C. Brost

Computer Science Department
Carnegie-Mellon University

ABSTRACT

During grasping motions, frictional forces and geometric constraint can combine to eliminate uncertainty in the location of the object. This paper incorporates the mechanics of friction and constraint in automatic planning of grasping motions. Our approach to automatic planning is centered on the use of *operation space*, which can be constructed for any parameterized class of operations. This approach was applied by Brost [1986] to plan planar parallel-jaw grasping motions. We extend Brost's work to include the finite dimensions of the fingers and the finite lengths of the motions. We explore the conditions for a successful grasp, and derive constraints in the operation space. An operation satisfying all of the constraints should produce the required result.

I. INTRODUCTION

An important aspect of a grasping operation is how it deals with uncertainty in the initial location of the object. In many applications, the orientation and position of an object will vary slightly from predicted values. At the least, a grasping operation should succeed despite such uncertainty. At best, we may be able to construct a grasping operation that eliminates the uncertainty. This paper focuses on grasping operations that eliminate the uncertainty mechanically, by the aligning and centering motions that occur when an object is pushed and squeezed between two fingers.

To plan grasping motions, we adopt the operation–space approach of Brost [1986], and extend it to address a broader set of issues. Brost explored the conditions for stability and convergence of object orientation, assuming that the fingers are wide enough, and approach from a great enough distance. To address the finite width and stroke of the fingers, we must construct bounds on the translations of the object, and incorporate these bounds in the planning process.

The central planning mechanism is the *operation space*. For any parameterized class of operations, we can define the operation space to be the space whose dimensions correspond to the parameters of the class. Each point in the space corresponds to a particular operation. A typical operation space constructed by Brost is shown in figure 1. For this diagram, the fingers are modeled as infinite half–planes, and the duration and speed of motions are not addressed. A two–parameter operation results, which can be viewed as a mapping from parameter values to final object configurations. This mapping is represented explicitly in the operation space, simplifying the choice of an appropriate pair of parameter values to obtain a desired outcome.

The operation–space approach involves three steps:

1) Define a parameterized operation. Each parameter defines a dimension of the operation space.

2) Develop the mechanics of the operation, combining models of the task and the operation.

3) Express the results as a mapping from the operation space to task outcomes. For any desired outcome, select a parameter value–set from the region that maps to that outcome.

Some forms of uncertainty are easily incorporated. For example, if uncertainty in one of the parameters is present, we can simply select a point that is not too close to the limits of the feasible region. This is equivalent to shrinking the feasible region before selecting a point. Other forms of uncertainty must be included in the mechanics of the operation.

Brost modeled the fingers as infinitely wide half–planes, approaching from infinity—planning the initial position was not an issue. In this paper, we assume that the two fingers have a finite width, and an initial position must be planned. Figure 2 shows part of the resulting four–dimensional operation space. As we shall see, the constraining relations among the parameters are not as simple as Brost's, but they are still fairly simple.

CH2345-7/86/0000/0124$01.00 © 1986 IEEE

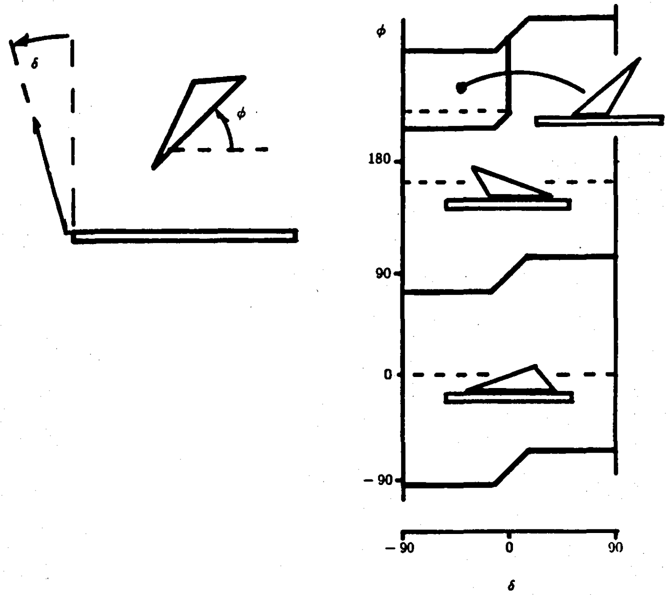


Figure 1. A two-parameter operation space for pushing, from Brost [1986]. The diagram shows the mapping from combinations of finger angle (ϕ) and pushing direction (δ) to resultant configurations of a triangle.

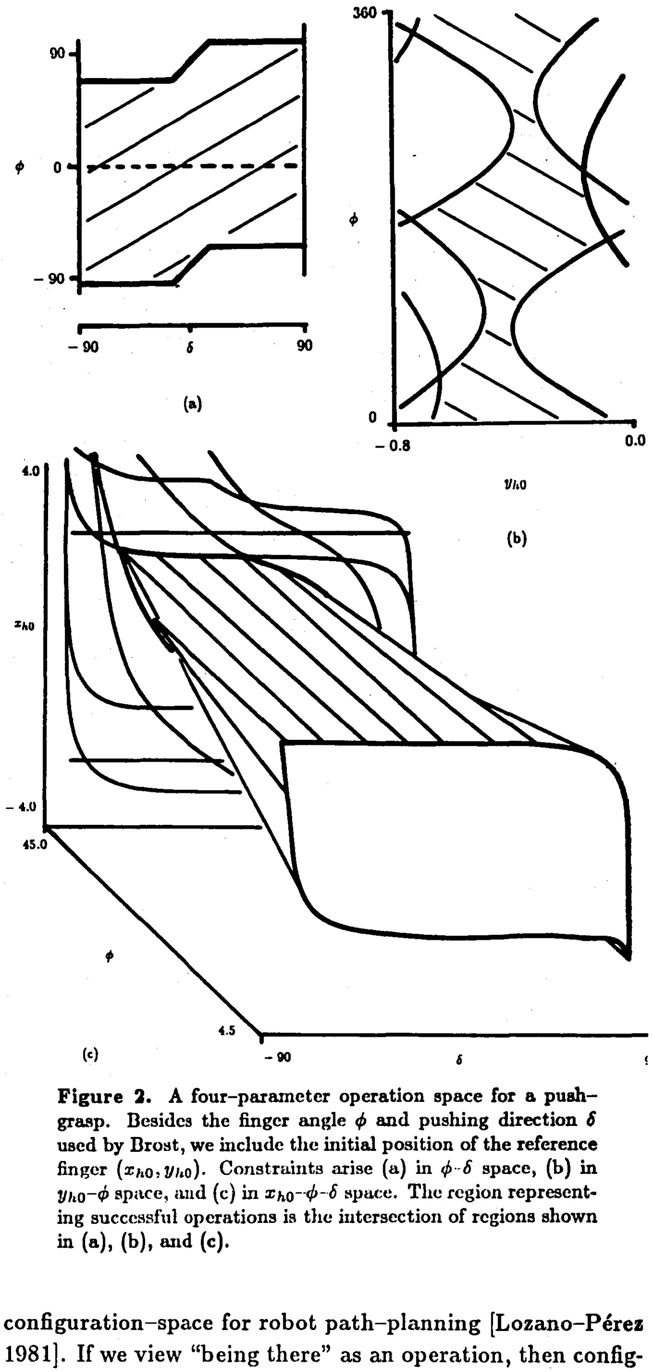


Figure 2. A four-parameter operation space for a push-grasp. Besides the finger angle ϕ and pushing direction δ used by Brost, we include the initial position of the reference finger (x_{h0}, y_{h0}). Constraints arise (a) in ϕ-δ space, (b) in y_{h0}-ϕ space, and (c) in x_{h0}-ϕ-δ space. The region representing successful operations is the intersection of regions shown in (a), (b), and (c).

I.A. Previous work

This paper draws primarily on research in the mechanics of pushing, and planning of pushing and squeezing motions. Mason [1982, 1986] explored the mechanics of pushing, and demonstrated automatic planning of push-grasp operations. Fearing [1984] incorporates the mechanics of pushing, and explores acquisition and stability of grasping with point fingers. Mani and Wilson [1985] address planning of a sequence of pushes to orient objects, and construct an operation space for pushing. Brost [1985, 1986] constructed operation spaces for three different grasping operations. Peshkin and Sanderson [1986] improved on Mason's analysis of pushing.

A number of other issues arise in the context of grasping, including control of a grasped object, mechanical properties of grasp configurations, and finding feasible grasp configurations. A short survey of this area should include Hanafusa and Asada [1977], Lozano-Pérez [1976, 1981], Salisbury [1982], Cutkosky [1985], Asada and By [1985], Kerr and Roth [1986], Abel, Holzmann, and McCarthy [1985].

There are a number of precedents in the use of operation space. Our operation space is a direct extension of the spaces constructed by Brost [1986]. Mani and Wilson [1985] independently constructed a pushing space. Kerr and Roth [1986] defines a space whose dimensions are the magnitudes of internal forces in a grasp; and the jamming and wedging diagrams of Simunovic [1975] and Whitney [1982] have the same character.

The operation space approach is similar to the use of configuration-space for robot path-planning [Lozano-Pérez 1981]. If we view "being there" as an operation, then configuration space is the corresponding operation space. A more useful operation is a straight-line motion, which is characterized by the initial and final configurations. The operation space would be $C \times C$, where C is the configuration space.

Similarities also hold with recent work in fine-motion planning [Lozano-Pérez, Mason, and Taylor 1984, Mason 1984, Erdmann 1984]. Assuming a simple, fixed, model of compliance, and a known task geometry, the robot path for a single motion command is completely characterized by an initial configuration and a nominal velocity.

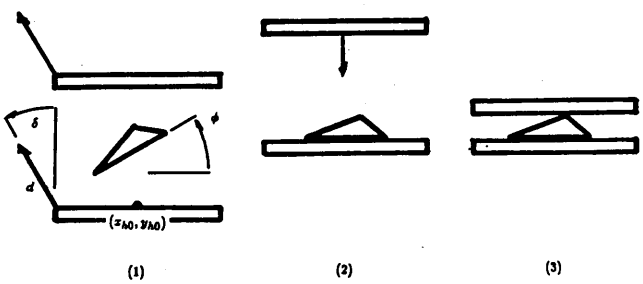


Figure 3. The push–grasp. The first finger pushes a distance d in direction δ, until the object is aligned with the finger. Then the second finger squeezes the object.

II. ANALYSIS OF THE OPERATION

Brost [1986] addresses three operations: push–grasp, squeeze-grasp, and offset–grasp. We will extend his results for the push–grasp, then briefly consider variations required for the other two operations. The push–grasp (see figure 3) pushes the object until it is stably aligned with the finger, then squeezes the object with the second finger. The general form is:

1) Move the hand to an initial position (x_{h0}, y_{h0}) and orientation ϕ, above the object;
2) Move down close to the table;
3) Translate the hand in direction δ for a distance d with velocity v;
4) Close the fingers, and translate the hand to hold the reference finger still.

We will treat the push–grasp as a four–parameter operation $(x_{h0}, y_{h0}, \phi, \delta)$. A lower bound for the pushing distance d arises as a by–product of our analysis. See Mason [1985] for an approach to producing bounds on the pushing velocity v. The initial position and orientation of the hand are expressed in a coordinate system fixed in the object, i.e. they represent *relative* position and orientation. Uncertainty in the object's initial location are equivalent to uncertainty in the parameters of the operation.

We will require the planner to verify that the following necessary conditions are satisfied:

- *geometrical conditions:* fingers clear object at initial position and orientation;
- *convergence and stability:* object converges to desired orientation relative to the fingers, and is stable in the final orientation; and
- *finite motion limits:* object does not roll or slide past the face of a finger.

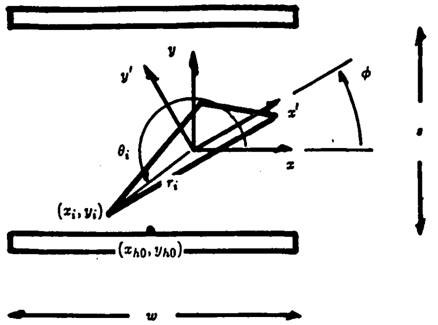


Figure 4. Notation for the push–grasp. Two coordinate frames are located at the object center of gravity. The x'–y' frame is fixed with respect to the object. The x–y frame is aligned with the fingers.

II.A. Geometrical conditions

First we address the problem of preventing undesired collisions, and producing the desired collision, between the fingers and the object. For this problem, the operation–space reduces to configuration–space. Using the notation of figure 4, the constraints are:

$$r_i \sin(\theta_i + \phi) - s < y_{h0} < r_i \sin(\theta_i + \phi)$$

for all i. This means that y_{h0} must fall between two sets of sine waves, as shown in figure 2(b).

II.B. Convergence and stability of orientation

This is precisely the problem addressed by Brost [1986], based on the earlier work of Mason [1982, 1986]. Assuming negligible inertial forces and Coulomb friction, the direction of rotation is determined by the coefficient of friction, the center of gravity, the direction of the finger motion, and the contact geometry. Stability and convergence are obtained for any operation inside the simple constraints shown in figure 2(a).

II.C. Finite fingers and finite pushing distance

The finite translations of the object are important in two respects. First, if our previous analysis is to be valid, we must ensure that each finite–width finger has the effect of an infinite half–plane. This means that the object must never make contact with an edge of the finger–face. Second, we must have an upper bound on the distance traveled during the rotation, so we know how far to push.

The primary complication is indeterminacy. We do not expect to know the distribution of support forces between the object and the table. Although the rotation direction is determined, the rotation rate, and the translational components of the motion, are not determined. Our first require-

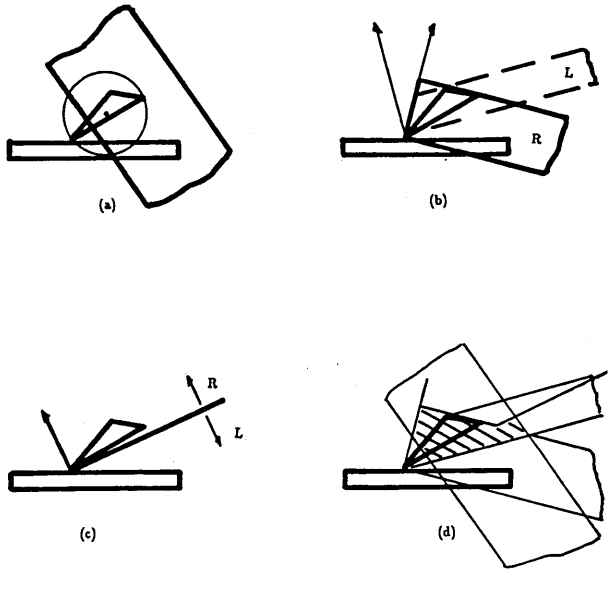


Figure 5. In order to bound the translational motion of the object, we require bounds on the instantaneous rotation center. We will use bounds arising from a variety of considerations. (a) shows the rotation center cannot fall inside the perpendicular bisector of the line connecting the contact point with the center of gravity. Nor can it fall outside a parallel line of distance $\frac{(p+r)^2}{p}$ from the contact, where r is the radius of a circle circumscribed about the support, and p is the distance from the contact to the center of gravity. (Peshkin and Sanderson [1986] have improved this bound close to $p + \frac{r^2}{p}$.) For a fixed ϕ, this defines a band—for ϕ varying over some interval, this defines a "bow-tie". Constraint (b) arises from considering the angle of force at the contact. Assume that the force is in the direction γ_R, a rotation center outside band R would give rise to net forces in a direction orthogonal to the hypothesized direction of force. If the contact force can actually lie anywhere inside the friction cone, we sweep the band to obtain another bow-tie. (b) also includes the result from [Mason 1982] that the rotation center cannot fall inside the friction cone. For constraint (c), we construct a line perpendicular to the finger velocity. Rotation centers above that line must be in region R of (b), and rotation centers below the line must be in region L of (b). (d) shows an example combining the constraints for a fixed δ.

ment, then, is to obtain at least some bounds on the possible motions. Figure 5 shows the bounds that we will use.

Given some bound on the possible rotation centers, we can proceed as follows. Circumscribe a rectangle, aligned with the finger face, about the set of possible rotation centers. Let the coordinates of the rectangle edges be x_r^+, x_r^-, y_r^+, y_r^-. It is easily shown that

$$y_r^- \le \frac{dx}{d\phi} \le y_r^+,$$
$$-x_r^+ \le \frac{dy}{d\phi} \le -x_r^-$$

where (x, y) describes the trajectory of the object vertex. If we compute these bounds for some fixed δ, letting ϕ vary over some interval, we can bound the variations in x and y:

$$y_r^+ \,\Delta\phi \le \Delta x \le y_r^- \,\Delta\phi,$$
$$-x_r^- \,\Delta\phi \le \Delta y \le -x_r^+ \,\Delta\phi.$$

for $\Delta\phi < 0$, as in the example. Let the origin be placed at the initial location of the contact vertex, then the final location is in the bounding rectangle $[x^-, x^+] \times [y^-, y^+]$, where

$$x^- = y_r^+ \,\Delta\phi,$$
$$x^+ = y_r^- \,\Delta\phi,$$
$$y^- = -x_r^- \,\Delta\phi,$$
$$y^+ = -x_r^+ \,\Delta\phi.$$

for $\Delta\phi < 0$. Although the details are not presented here, the bounds on the rotation center, y_r^- etc., are simple trigonometric functions of δ. Hence the bounds on the contact vertex, x^- etc., are trigonometric functions of δ, and linear in ϕ.

We can be confident of $\Delta\phi$ rotation by pushing at least as far as y^+ in the y-direction. However, it is almost certain that the rotation will be completed *before* the motion is complete, so we must also consider the pure translation of the object after rotation ceases. It is easily shown that if $|\delta| \le \alpha$, where $\alpha = \tan^{-1}\mu$, the object will translate in direction δ, otherwise it will translate in direction α.

These observations are summarized in figure 6. The path of the contact vertex, and the contact edge after rotation is complete, is bounded by a polygonal region of the plane. Letting (x_i, y_i) be a vertex of the bounding polygon, we can write

$$x_i - w < x_{h0} + y_i \tan\delta < x_i + w.$$

The x_i are 0, x^-, or $x^+ + l$, where l is the length of the edge, and the y_i are 0, y^-, or y^+. These are all simple trigonometric functions of δ, and linear in ϕ. In general, we obtain constraints of the form,

$$f_i(\delta)\,\Delta\phi + c_i - w < x_{h0} < f_i(\delta)\,\Delta\phi + c_i + w$$

where the $f_i(\delta)$ are trigonometric functions, and the c_i are constants. Figure 2(c) shows an example of these constraints plotted in the operation-space.

This analysis suffices for simple push grasps involving rotation about a single contact vertex. A more general anal-

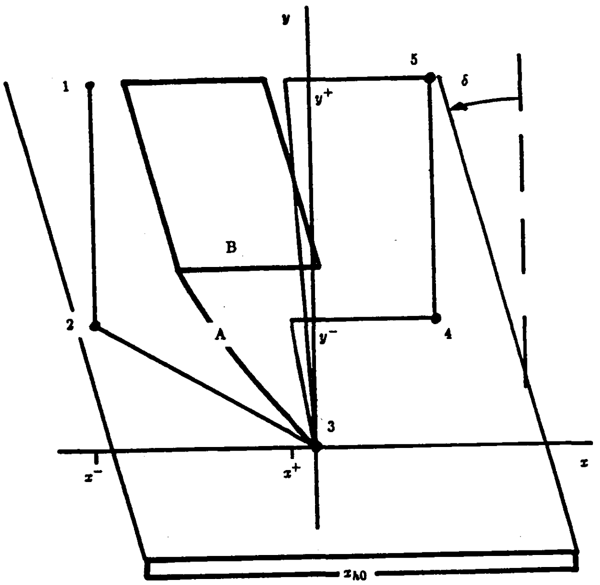


Figure 6. The region 12345 bounds the possible locations of contact of the object with the finger. A sufficient condition is that the area swept by the finger include this region. Also shown is the contact region for a typical motion. *A* shows the locus of the contact vertex as the object rotates, and *B* shows the region of contact after the rotation is complete.

ysis is required for complex grasps where the object may roll from one vertex to the next. The basic idea would be to repeat the above analysis for each vertex, and splice the envelopes together.

II.D. Related operations: squeeze–grasp and offset–grasp

The analysis of the push–grasp almost encompasses two similar operations, the squeeze–grasp and the offset–grasp. Both motions dispense with the preliminary pushing motion, and rely on pure squeezing for alignment and centering of the object. The offset–grasp requires that the reference finger makes first contact with the object, while the squeeze–grasp is indifferent to which finger hits first. The main difference in constructing the operation spaces is that a different set of rotation center bounds are required. Rotation center bounds for an example problem proved to be somewhat simpler than the bounds described in figure 5.

REFERENCES

Abel, J. M., Holzmann, W., and McCarthy, J. M. 1985. "On Grasping Planar Objects with Two Articulated Fingers," *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, St Louis.

Asada, H., and By, A. B. 1985. "Kinematics of Workpart Fixturing," *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, St Louis.

Brost, R. C. 1985. "Planning Robot Grasping Motions in the Presence of Uncertainty," CMU-RI-TR-85-12, Robotics Institute, Carnegie-Mellon University.

Brost, R. C. 1986. "Automatic Grasp Planning in the Presence of Uncertainty," *Proceedings of the 1986 IEEE International Conference on Robotics and Automation*, San Francisco.

Cutkosky, M. R. 1985. "Grasping and Fine Manipulation for Automated Manufacturing," PhD thesis, Carnegie-Mellon University.

Erdmann, M. A. 1984. "On Motion Planning with Uncertainty," TR-810, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Fearing, R. S. 1984. "Simplified Grasping and Manipulation with Dextrous Robot Hands," American Control Conference, San Diego.

Hanafusa, H., and Asada, H. 1977. "A Robotic Hand with Elastic Fingers and its Application to Assembly Process," *IFAC Symposium on Information and Control Problems in Manufacturing Technology.*

Kerr, J., and Roth, B. 1986. "Analysis of Multifingered Hands," *International Journal of Robotics Research*, 4(4).

Lozano-Pérez, T. 1976. "The Design of a Mechanical Assembly System," TR-397, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Lozano-Pérez, T. 1981. "Automatic Planning of Manipulator Transfer Movements," *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(10).

Lozano-Pérez, T., Mason, M. T., and Taylor, R. H. 1984. "Automatic Synthesis of Fine-Motion Strategies for Robots," *International Journal of Robotics Research* 3(1).

Mani, M., and Wilson, W. R. D. 1985. "A Programmable Orienting System for Flat Parts," *Proceedings, NAMRI XIII*, Berkeley.

Mason, M. T. 1982. "Manipulator Grasping and Pushing Operations," AI-TR-690, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Mason, M. T. 1984. "Automatic Planning of Fine Motions: Correctness and Completeness," *Proceedings, IEEE International Conference Robotics*, Atlanta.

Mason, M. T. 1985. "On the Scope of Quasi-Static Pushing," *3rd International Symposium on Robotics Research*, Gouvieux.

Mason, M. T. 1986. "Mechanics and Planning of Manipulator Pushing Operations," *International Journal of Robotics Research*, 5(1).

Peshkin, M., and Sanderson, A. 1986. "Manipulation of a Sliding Object," *proceedings, 1986 IEEE International Conference on Robotics and Automation*, San Francisco.

Salisbury, J. K. 1982. "Kinematic and Force Analysis of Articulated Hands," PhD thesis, Department of Mechanical Engineering, Stanford University.

Simunovic, S. N. 1975. "Force Information in Assembly Processes," *Proceedings, 5th International Symposium on Industrial Robots.*

Whitney, D. E. 1982. "Quasi-Static Assembly of Compliantly Supported Rigid Parts," *Journal of Dynamic Systems, Measurement, and Control* 104:65.

Constructing Stable Force-Closure Grasps

Van-Duc Nguyen [1]

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper addresses the problem of synthesizing planar grasps that are force-closure and stable, with point contacts with friction. The synthesis of force-closure grasps constructs independent regions of contact for the fingertips, such that the motion of the grasped object is totally constrained. The synthesis of stable grasps constructs virtual springs at the contacts, such that the grasped object is stable, and has a desired stiffness matrix about its stable equilibrium.

A force-closure grasp implies equilibrium grasps exist. In the reverse direction, we prove that non-marginal equilibrium grasps with at least two point contacts with friction are also force-closure grasps. Next, we prove that all force-closure grasps can be made stable, by using either active or passive springs at the contacts.

The paper presents fast and simple algorithms for directly constructing stable force-closure grasps based on the shape of the grasped object. It develops a simple geometric relation between the stiffness of the grasp and the spatial configuration of the virtual springs at the contacts. The stiffness of the grasp depends also on whether the points of contact stick, or slide without friction on the edges of the object.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's Artificial Intelligence research is provided in part by the System Development Foundation, and in part by the Advanced Research Projects Agency of the Department of Defense, under Office of Naval Researchcontacts N00014-85-K-0124.

1 Introduction

Robot end effectors have evolved from simple parallel grippers to multi-finger hands to provide greater flexibility and dexterity in manipulation and assembly operations. Robot hands come in many shapes, but they all have in common the ability of being programmed and servoed from a computer. To take full advantage of the dexterity offered by multi-purpose hands, we need to be able to not only *analyze* a grasp but *synthesize* it. In other words, we would like to *construct* grasps that have such features as feasibility, reachability, force-closure, equilibrium, stability, compliance, etc...

This paper addresses the problem of synthesizing grasps that are force-closure, equilibrium, stable, and compliant. Synthesizing a force-closure grasp is equivalent to finding places to put the contacts, such that these contacts totally constrain the motion of the grasped object. The contacts between the fingertips and the object are modeled as point contacts with friction. Constructing an equilibrium grasp is synthesizing the forces and moments at the contacts, such that the object is in equilibrium. Constructing a stable grasp is finding the independent virtual springs at the contacts, such that the grasp has a positive definite stiffness matrix. Constructing a compliant grasp is maping the desired stiffness matrix at the grasped object into the stiffness matrices at the fingertips and at the finger joints. The compliance model used is a generalized spring. We also explore how slip affects the force-closure and stability properties of the grasp.

1.1 A Grasp Planner

A Grasp Planner can construct a stable grasp G on a set of n edges or faces as follows:

- Synthesize a set of grasp points $\{P_1, \ldots, P_n\}$ for which the grasp G at these grasp points is force-closure. Better yet, we find the optimal set of independent regions of contact $\{s_1, \ldots, s_n\}$ for the given n grasping edges or faces. The regions are independent from each other, in the sense that as long as we pick grasp point P_i in region s_i the resulting grasp $G = \{P_1, \ldots, P_n\}$ is always force-closure. The set is optimal in the sense that the set of independent regions has the largest miminum radius for the given set of grasping edges or faces. We then pick the mid point of the region s_i as the optimal grasp point P_i.
- Synthesize a corresponding set of virtual springs, such that grasp G'is stable. Each virtual spring has stiffness k_i and compression σ_{io}. We prove that the stiffnesses k_i's and compressions σ_{io}'s of the virtual springs can always be chosen so as to make the grasp stable. We can also construct the set of virtual springs such that the grasp has some desired compliance center and stiffness matrix.

[1] The author is currently with General Electric Corporate Research and Development Center, in Schenectady, New York.

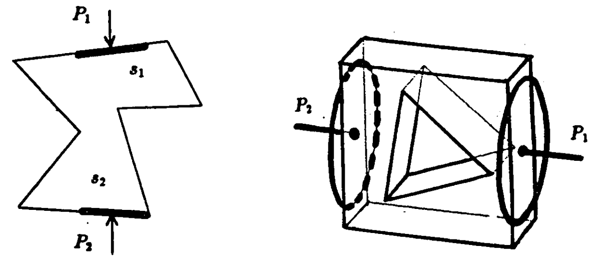


Figure 1: Examples of force-closure grasps found by the synthesis.

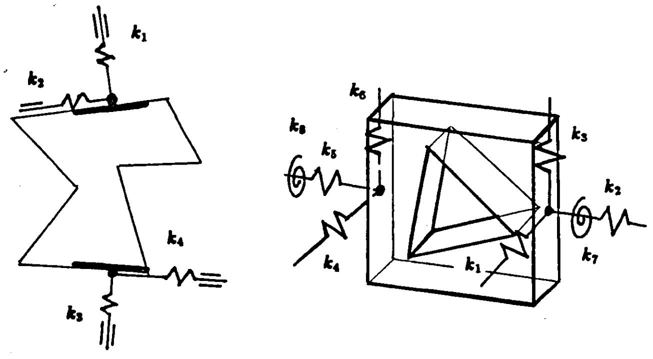


Figure 2: Examples of stable grasps found by the synthesis.

Figure 1 shows examples of force-closure grasps found by the algorithms. The independent regions of contact are highlighted with bold segments and circles. The grasp is force-closure no matter where the finger tips are placed in these regions. This flexibility is of great importance in manipulation since we always have positioning errors and many other uncertainties.

Figure 2 shows examples of stable grasps constructed by the algorithms. The stiffness matrix at the grasped object is mapped into the virtual springs at the contacts. These virtual springs are responsible for generating restoring forces and moments whenever the grasped object is displaced from its stable equilibrium.

1.2 Other Related Works

Related works can be grouped as follows:

- Feasible and reachable grasps. — Feasibility and reachability problems can be solved using the Configuration Space approach (Lozano-Pérez 1983), which grows the grasped object into a configuration obstacle in higher dimensional space, and inversely shrinks the fingers into a configuration point. The two problems become a search respectively for a feasible configuration point, and a path to that feasible point, such that the path does not intersect the configuration obstacle (Lozano-Pérez 1976, 1983).
- Force-closure grasps — Force-closure and total freedom capture the main constraint between the fingers and the grasped object. Ohwovoriole analyzed the geometry of the different repelling screw systems, and use the results to analyze systems of contacting bodies such as an object grasped by a set of fingers, or a pin being inserted into a hole (Ohwovoriole 1980, 1984). Related to force-closure is the solution of systems of linear inequalities (Kuhn and Tucker 1956).
- Form-closure grasps — A grasp is form-closure if the grasped object is totally constrained by the set of contacts, irrespective of the magnitude of the contact forces. Reuleaux (1875) proved that a 2D grasp needs at least four point contacts for form-closure. Lakshminarayana (1978) showed that a 3D grasp needs at least seven point contacts. Form-closure can be viewed as force-closure with frictionless contacts only.
- Equilibrium grasps — There are many works on analyzing the equilibrium of forces in a grasp with different types of contact (Salisbury 1982), with flexible contacts (Cutkosky 1984), or with friction (Abel, Holzmann and McCarthy 1985). Finding a good grasp is often formalized as a search of the space of all grasps with some goal function, such as optimum for internal forces (Kerr 1984), or security of grasp (Jameson 1985).
- Stable grasps — A stable prehension of a planar hand on a polygon can be found by centering the hand on the center of mass, and check for grasps that are stable with respect to rotation, then stable with respect to translation (Hanafusa and Asada 1977), (Asada 1979). Baker, Fortune and Grosse (1985) proved that stable grasps on a convex polygon exist, and presented efficient algorithms that require no incremental search.
- Compliant grasps — We can have active stiffness control of the fingers and the grasped object by using the Grip and Jacobian matrices as in (Salisbury and Craig 1981), (Salisbury 1984, 1982), or build in some proximity damping as in (Jacobsen, Wood, Knutti and Biggers 1985). Grasps can be achieved easily with active compliance and bounded slip at the fingers as in (Fearing 1984), or by exploiting the friction and passive compliance of the object with the fingers and the environment as in (Mason 1982, Brost 1986). Grasping a peg and inserting it into a hole is currently done best with a passive compliance wrist known as the Remote Center of Compliance (Whitney 1982).

2 Constructing Force-Closure Grasps

2.1 Representing Contacts and Grasps

Figure 3 depicts the different types of planar contacts with friction. The first column describes the physical contact with the finger on top and the grasped object below it. The second and third columns describe respectively the wrench convex, representing the range of forces that can be applied to the object, and the twist convex, representing the total freedom of the object. Each convex is represented by a minimal set of generating vectors. The twist convex is computed by taking the dual of the wrench convex.

a. *Point contact with friction* — The friction cone at the point of contact shows the range of pure force that can be applied through the point contact. The wrench convex has two wrenches which are along the two extreme rays of the friction cone. Any force pointing inside this friction cone can be written uniquely as a positive combination of the these two wrenches. The twist convex has two unisense translations, each reciprocal to one wrench and repelling to the other. It also has a free rotation about the point of contact as above.

b. *Edge contact with friction* — It is well known that, for rigid objects, any force distribution along the segment of contact is equivalent to a unique force at some point inside the segment. This unique force is mathematically the positive combination of two ranges of force at the two ends of the segment of contact. Specifically, the wrench convex of an edge contact with friction is equivalent to the convex sum of two wrench convexes, each represents a point contact with friction at one end of the edge of contact.

c. *Soft finger contact* — From a force-closure point of view, a soft finger contacting an edge is the same as an edge contact with friction. The pressure distribution is irrelevant to our current focus, which is concerned with whether the object can be constrained

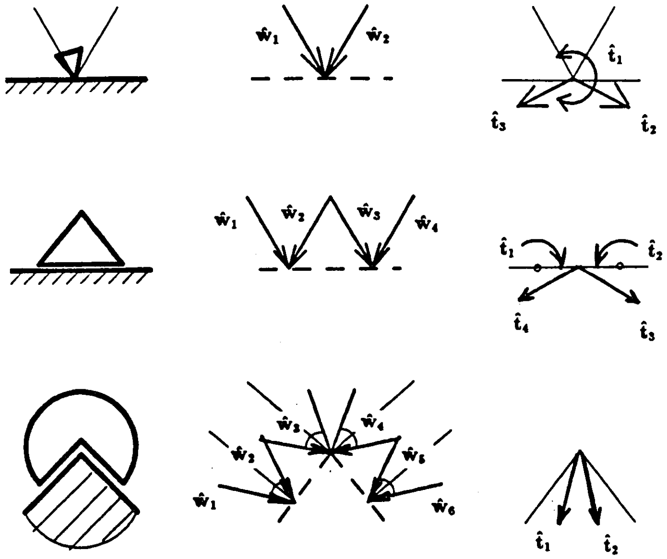


Figure 3: Planar contacts with friction and their twist and wrench convexes.

with the given contacts, rather than how much force should the fingers apply to the object.

A soft finger becomes more useful when it contacts on the inside or outside of a corner. Figure 3.c shows a soft finger contacting on the outside of a corner. The wrench convex is the convex addition of two convexes, each describes the edge contact with friction on one side of the corner. This wrench convex can be approximated by a much larger friction cone.

Gravity is not a contact, but it does play a role in constraining the total freedom of the object. For example, a box is immobile on a table because the force of gravity is holding it down to the table. We can view the box as being grasped, or more exactly constrained, by two contacts: a plane contact between the bottom of the box and the table, and an imaginary point contact at the center of gravity of the box.

Twist and wrench convexes are two dual representations for contacts. We can add wrench convexes from all the contacts or intersect the corresponding twist convexes to find the resulting wrench or twist convex of the grasp. We have two dual view points:

- A constraint view point. — Wrench convex describes the set of forces and moments which constrain the object. A total wrench convex means we can arbitrarily apply any force and moment on the object, and so we can grasp it, instantaneously rotate or translate it in any way we want.
- A freedom view point. — Twist convex describes the total freedom of the object. A total twist convex means the object can freely move relative to the fingers. A null twist convex means the object cannot break contact without external work against the contact forces exerted by the fingers.

For planning grasps, wrench convexes are definitely more efficient since generating wrenches can be deduced readily from the type of contact, and we can just take the union of all the generating wrenches to describe the grasp. The twist convex representation is more efficient for describing the total freedom at the end effectors of linked manipulators. The infinitesimal motions and the velocities of the end effector due to each joint are 'added'. The end effector can have arbitrary motion if the twist convexes of all the joints 'add up' to a total convex.

2.2 Resisting Translation and Rotation

The force closure problem can be formulated as solving a system of linear inequalities:

$$W^T \hat{t}^S \geq 0 \qquad (1)$$

where the columns of W are generating wrenches collected from all the contacts of the grasp. We can design a generate-and-test algorithm which enumerates all the possible grasps, and test each grasp by solving the above system of linear inequalities. We get a force-closure grasp if and only if there is no non-zero solution to the above system, i.e. zero total freedom. There are two main objections to this scheme: first, the set of possible grasps is infinite; second, the grasp synthesis uses an analytical formulation which does not exploit the geometry of the domain.

Polygonal objects have straight edges; contacts on straight edges have wrench convexes whose representations can be split into a convex set for the point contact, and a convex cone for the range of force directions. This property applies only to polygonal and polyhedral objects. A force (resp. infinitesimal rotation) is a line vector while a torque (resp. infinitesimal translation) is a free vector which does not depend on the point of contact. Our key contribution is to make the force-closure constraint explicit for polygonal objects.

Theorem 1 *A set of wrenches W can generate force in any direction if and only if there exists a three-tuple of wrenches $(\hat{w}_1, \hat{w}_2, \hat{w}_3)$ whose respective force directions f_1, f_2, f_3 satisfy:*

- *Two of the three directions f_1, f_2, f_3 are independent.*
- *A strictly positive combination of the three directions is zero.*

$$\alpha f_1 + \beta f_2 + \gamma f_3 = 0$$

The first (resp. second) condition corresponds to no homogeneous (resp. particular) solutions to the system $W^T \hat{t}^S \geq 0$, where twist $\hat{t} = (0, d_x, d_y)^T$ is an infinitesimal translation of the object. For detailed proofs the reader is referred to (Nguyen 1985a, 1986). Theorem 1 can be captured in a more suggestive and compact way, Figure 4, as follows:

Corollary 1 *A set of wrenches W can generate forces in any arbitrary direction if and only if there exists a three-tuple of force-direction vectors (f_1, f_2, f_3) whose end points draw a nonzero triangle that includes their common starting point.*

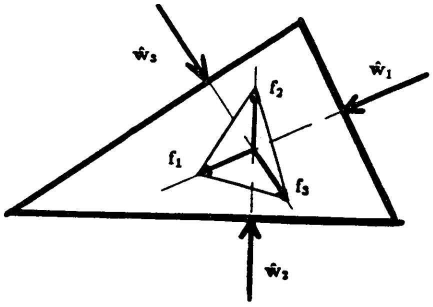


Figure 4: A geometrical view of force-direction closure.

Torque closure can be achieved easily by creating enough friction on some axis of rotation of the object. The friction between the rotating object and its supporting axis will create a torque which resists any clockwise or counter-clockwise rotation of the object. Unfortunately, in most grasp configurations, we have only point contacts, and through a point contact, a finger can exert only a pure force on the object and not torque. The more interesting problem is to achieve torque closure with only pure forces.

Theorem 2 *A set of planar forces W can generate clockwise and counter-clockwise torques if and only if there exists a four-tuple of forces $(\hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2, \hat{\mathbf{w}}_3, \hat{\mathbf{w}}_4)$ such that:*

- *Three of the four forces have lines of action that do not intersect at a common point or at infinity.*
- *Let $\mathbf{f}_1, \ldots, \mathbf{f}_4$ be the force directions of $\hat{\mathbf{w}}_1, \ldots, \hat{\mathbf{w}}_4$. Let $\mathbf{p}_{12}$ (resp. $\mathbf{p}_{34}$) be the point where the lines of action of $\hat{\mathbf{w}}_1$ and $\hat{\mathbf{w}}_2$ (resp. $\hat{\mathbf{w}}_3$, and $\hat{\mathbf{w}}_4$) intersect. There exist $\alpha, \beta, \gamma, \delta$ all greater than zero, such that:*

$$\begin{aligned} \mathbf{p}_{34} - \mathbf{p}_{12} &= \pm(\alpha \mathbf{f}_1 + \beta \mathbf{f}_2) \\ &= \mp(\gamma \mathbf{f}_3 + \delta \mathbf{f}_4) \end{aligned}$$

The first (resp. second) condition corresponds to no homogenous (resp. particular) solutions to system $W^T \hat{\mathbf{t}}^S \geq \mathbf{0}$, where twist $\hat{\mathbf{t}} = (\delta_z, d_x, d_y)^T$ is an infinitesimal rotation of the object. For detailed proofs the reader is referred to (Nguyen 1986). Theorem 2 can be formulated in more geometrical terms, Figure 5, as follows:

Corollary 2 *A set of planar forces W can generate clockwise and counter-clockwise torques if and only if there exists a four-tuple of forces $(\hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2, \hat{\mathbf{w}}_3, \hat{\mathbf{w}}_4)$ such that the segment $P_{12}P_{34}$, or $P_{34}P_{12}$, points out of and into the 2 cones $C_{12}^{\lessgtr}$, $C_{34}^{\lessgtr}$, formed by the two pairs $(\hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2)$, and $(\hat{\mathbf{w}}_3, \hat{\mathbf{w}}_4)$.*

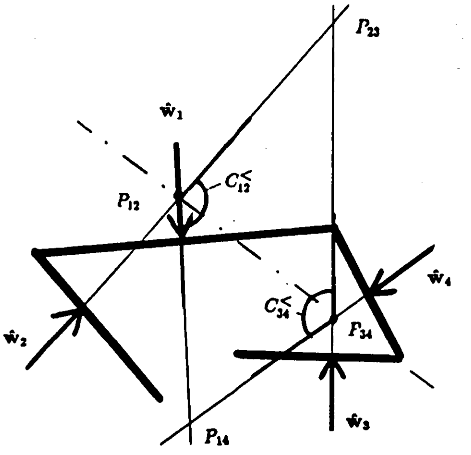


Figure 5: A geometrical view of torque closure.

2.3 Finding All Force Closure Grasps

The convex addition of the convex of all force directions $\mathrm{co}[f_x, f_y]$ and the convex of all torques $\mathrm{co}[m_z]$ is the convex of all planar forces $\mathrm{co}[f_x, f_y, m_z]$. So, the necessary and sufficient condition for force closure is contained in both Theorems 1 and 2. If we assume that through any contact we can only exert force and not torque, then Theorem 2 subsumes Theorem 1. A translation can be viewed as a rotation with point of rotation at the infinity. So, if there is no free rotation for the grasped object constrained by a set of contact forces, then there exists also no free translation. Thus Corollary 2 describes the geometrical necessary and sufficient condition for force closure with planar forces only.

Corollary 3 *Two point contacts with friction at P and Q form a force closure grasp if and only if the segment PQ, or QP, points strictly into and out of the two friction cones respectively at P and Q.*

Proof: This is a well known fact of planar mechanics. Let's however prove the above corollary using a reduction to Corollary 2. A friction cone at P (resp. Q) is equivalent to two forces $\hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2$ (resp. $\hat{\mathbf{w}}_3, \hat{\mathbf{w}}_4$) along the edge of friction cone and going through P (resp. Q), Figure 6. We recognize that point P (resp. Q) is nothing more than the point P_{12} (resp. P_{34}). ∎

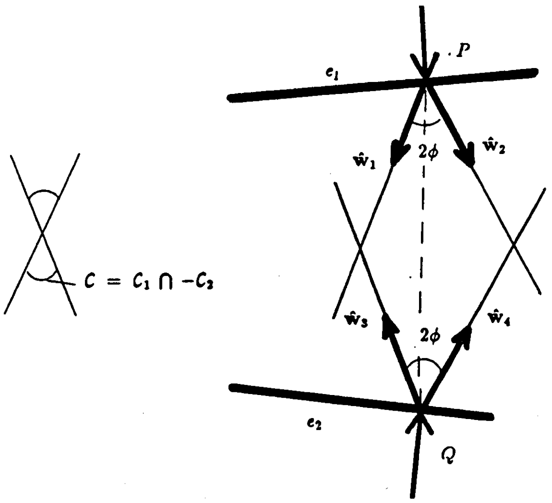


Figure 6: Finding grasps with friction on two edges.

Lemma 1 *The set of all possible grasps with friction on two edges e_1, e_2, denoted $\mathcal{G}(e_1, e_2)$, is completely described by the two edges e_1, e_2, and the counter-overlapping sector $C = C_1 \cap -C_2$ of the two friction cones resp. from edge e_1 and e_2.*

We have seen that a soft finger contacting at a vertex can be approximated as a point contact with a much wider friction cone. From Corollary 3, the larger the friction cones at the points of contacts, the greater is the likelihood that they counteroverlap, or that the grasp is force closure. So a soft finger gives us more flexibility than a point contact with friction. This partially explains why people grasp objects at edges and corners, and also why the contacting surface of human fingers had better be soft than hard like the finger nails.

By definition, a force-closure grasp is a set of contacts which allows us to exert arbitrary force and moment on the object by pushing at the contacts. So, we can exert zero force and moment, i.e. have an equilibrium grasp. In the other direction, it turns out that most equilibrium grasps with point contacts with friction are also force-closure grasps:

Corollary 4 *Let G be a grasp with at least two distinct point contacts with friction. G is a force closure grasp if it is an equilibrium grasp, and has contact forces pointing strictly within their friction cones.*

Proof: The two friction cones gives three lines of force which are not all parallel because the friction cones are not null. These three lines of forces do not all intersect at the same point because the two points of contact are distinct. So, we have three planar wrenches with independent spatial vectors. The set of contact wrenches is also force-closure, or vector-closure, if there exists a strictly positive combination of four contact wrenches that results in the zero wrench, or equilibrium. (The theorem for vector-closure has the same form as Theorem 1, except that we need at least $n+1$ vectors for an n-dimensional space.) The coefficients of the contact wrenches must be strictly positive, i.e. the contact forces must point strictly inside their respective friction cones. ∎

2.4 Finding Independent Regions of Contact

In task planning, we are interested in finding grasps that require as little accuracy as possible. One aspect of that goal is to have grasps such that the fingers can be positioned independently from each other, not at discrete points, but within large regions of the edges. Corollary 3 allows us to cast the problem of finding the independent regions of contact on two edges into a problem of fitting a two-sided cone cutting these two edges into two segments of largest minimum length, Figure 7.

Algorithm 1 *The independent regions of contact on two edges e_1 and e_2 can be constructed as follows:*

1. Find the two-sided cone $C^{\times}$ (I_1, $\pm C$) that cuts all of edge e_1 and very little or none of edge e_2. We get a triangle Δ_1 formed by edge e_1 and vertex I_1. This triangle represents the set of vertices I, where the two-sided cone $C^{\times}$ (I, $\pm C$) monotonically cuts larger segment e'_1 and smaller segment e'_2 as we move from edge e_1 to e_2. Similarly, we find the two-sided cone $C^{\times}$ (I_2, $\pm C$) such that this later cuts exactly the edge e_2 and very little or none of edge e_1. We get a triangle Δ_2 formed by edge e_2 and vertex I_2.

2. Find the trade-off region for vertex I by intersecting the triangle Δ_1 with Δ_2. The trade-off region describes the locus of vertex I, for which the two-sided cone $C^{\times}$ (I, $\pm C$) cuts both edges e_1 and e_2 into segments e'_1 and e'_2. The length of the independent segments e'_1 and e'_2 is proportional to the distance from vertex I of the two-sided cone to the respective edges.

3. We cut the trade-off region with the bisector of the two edges e_1, and e_2. The intersection is the locus of vertex I for which the two segments e'_1 and e'_2 have the same length. The optimal vertex I^* is at one of the two endpoints of the intersecting segment, or anywhere on this segment, depending on the angle between the two edges. If no intersecting segment exists, then the optimal vertex is the point of the trade-off region which is nearest to the bisector.

4. From the optimal vertex I^*, the independent regions of contact s_1 and s_2 are found by cutting the two-sided cone $C^{\times}$ (I^*, $\pm C$) with the grasping edges e_1 and e_2.

The computation of the optimum independent regions of contact for two point contacts with friction on two edges takes about 0.05 seconds. The code is written in Zeta Lisp, compiled and run on a Symbolics machine.

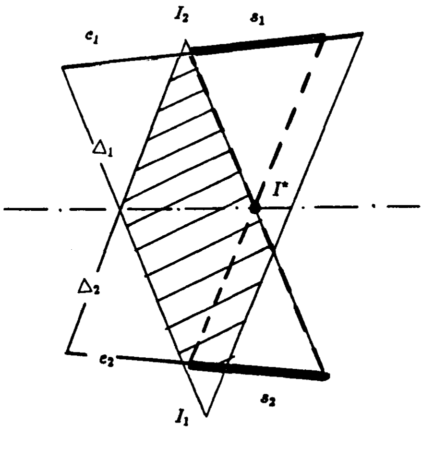


Figure 7: Finding the independent regions of contact on two edges.

3 Constructing Stable Grasps

3.1 Potential Function of the Grasp

Figure 8 shows a finger F_i contacting with friction at point P_i. The softness of the fingertip, the stiffness of the tendons attached to the finger, and/or the active control of the finger joints are all modeled by two virtual springs with linear stiffness k_i, k_j. The linear spring k_i has fixed line of action, with direction $\mathbf{k}_i = (\cos\alpha_i, \sin\alpha_i)^T$, and moment $\mu_i = \mathbf{p}_i \times \mathbf{k}_i$, about the origin O. (The cross product of two 2-dimensional vector is a scalar, which is the product of the two vector magnitudes and the sin of the angle between the two vectors.)

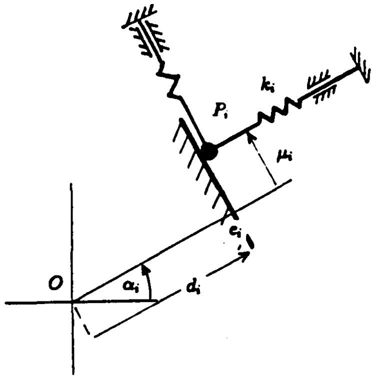


Figure 8: A fingertip which sticks is modeled by two linear springs.

We assume that there is no dissipation of potential energy in grasp, so the fingertips do not slide on the object. As the object is displaced by (x, y, θ), the point P_i is mapped into its new position P'_i given by:

$$\mathbf{p}'_i = \begin{bmatrix} C\theta & -S\theta \\ S\theta & C\theta \end{bmatrix} \mathbf{p}_i + \begin{bmatrix} x \\ y \end{bmatrix} \tag{2}$$

The linear spring k_i is compressed by an amount equal to the projection of the displacement $P_iP'_i$ onto the line of action of spring k_i:

$$\begin{aligned} \sigma_i(x, y, \theta) &= \sigma_{io} + (\mathbf{p}'_i - \mathbf{p}_i) \cdot \mathbf{k}_i \\ &= \sigma_{io} + \big((C\theta - 1)p_{ix} - S\theta p_{iy} + x\big)\ C_i \\ &\quad + \big(S\theta p_{ix} - (C\theta - 1)p_{iy} + y\big)\ S_i \end{aligned} \tag{3}$$

The potential function of grasp G is equal to the sum of the potentials of all its springs (Hanafusa and Asada 1977):

$$U(x, y, \theta) = \sum_{i=1}^{n} \frac{1}{2} k_i \sigma_i^2(x, y, \theta) \tag{4}$$

3.2 Grasp Equilibrium

Theorem 3 *A grasp G composed of n virtual springs is in equilibrium if and only if:*

$$\left\{ \begin{array}{lcll} \left.\frac{\partial U}{\partial x}\right|_{(0,0,0)} & = & \sum_{i=1}^{n} k_i\sigma_{io}\cos\alpha_i & = 0 \\ \left.\frac{\partial U}{\partial y}\right|_{(0,0,0)} & = & \sum_{i=1}^{n} k_i\sigma_{io}\sin\alpha_i & = 0 \\ \left.\frac{\partial U}{\partial \theta}\right|_{(0,0,0)} & = & \sum_{i=1}^{n} k_i\sigma_{io}\,\mu_i & = 0 \end{array} \right. \tag{5}$$

In the above columns, we recognize the spatial vectors $\hat{\mathbf{k}}_i = (\cos\alpha_i, \sin\alpha_i, \mu_i)^T$, describing the lines of action of the linear springs k_i, or as the unit wrenches describing the frictionless point contacts P_i. The above system of equations can be rewriten in a force-closure form:

$$\sum_{i=1}^{n} f_{io}\,\hat{\mathbf{k}}_i \;=\; \hat{\mathbf{0}}, \qquad f_{io} \geq 0 \tag{6}$$

where $f_{io} = k_i\sigma_{io}$. The force-closure condition is sufficient but not necessary for the existence of force equilibrium:

Corollary 5 *If grasp G is force-closure, then we can always find a set of positive contact forces at the points of contact, such that G is in equilibrium.*

3.2.1 Grasp Stability

The grasp G is stable if and only if the potential function $U(x,y,\theta)$ of G reaches a local minimum. The Taylor expansion of the potential function $U(x,y,\theta)$ about the equilibrium as follows:

$$U\,(x,y,\theta) \;=\; \sum_{i=1}^{n} \frac{1}{2}k_i\sigma_{io}^2 + \mathbf{x}^T\nabla\, U|_{(0,0,0)} + \frac{1}{2}\mathbf{x}^T\, H|_{(0,0,0)}\,\mathbf{x} + \ldots \tag{7}$$

where $\mathbf{x} = (x,y,\theta)^T$, and $H|_{(0,0,0)}$ is the Hessian matrix of the potential function at the equilibrium grasp configuration. A multivariable function reaches a local minimum if 1) the first partial derivatives are all zero, and 2) the Hessian matrix of the second partial derivatives is positive definite.

Theorem 4 *A grasp G composed of n virtual springs is in stable equilibrium if both of the following hold:*

- *The gradient $\nabla\, U|_{(0,0,0)}$ is zero.*
- *The Hessian matrix $H|_{(0,0,0)}$ of the potential function $U(x,y,\theta)$ is positive definite.*

$$H_O \;=\; \begin{pmatrix} \frac{\partial^2 U}{\partial x^2} & \frac{\partial^2 U}{\partial x\partial y} & \frac{\partial^2 U}{\partial x\partial\theta} \\ \frac{\partial^2 U}{\partial y\partial x} & \frac{\partial^2 U}{\partial y^2} & \frac{\partial^2 U}{\partial y\partial\theta} \\ \frac{\partial^2 U}{\partial\theta\partial x} & \frac{\partial^2 U}{\partial\theta\partial y} & \frac{\partial^2 U}{\partial\theta^2} \end{pmatrix} \qquad at\ (x,y,\theta) \;=\; (0,0,0)$$

$$= \begin{pmatrix} \sum k_i\cos^2\alpha_i & \sum k_i\sin\alpha_i\cos\alpha_i & \sum k_i\mu_i\cos\alpha_i \\ \sum k_i\sin\alpha_i\cos\alpha_i & \sum k_i\sin^2\alpha_i & \sum k_i\mu_i\sin\alpha_i \\ \sum k_i\mu_i\cos\alpha_i & \sum k_i\mu_i\sin\alpha_i & \sum k_i\mu_i^2 \;-\; \sum f_{io}d_i \end{pmatrix} \tag{8}$$

3.3 Stiffness Matrix of the Grasp

The stiffness matrix K of the grasp is equal to the Hessian matrix H_O about the stable equilibrium of the grasped object. The stiffness matrix K can be written as a sum of two matrices:

$$\begin{aligned} K \;&=\; K_S \;+\; K_P \\ K_S \;&=\; S\;K\;S^T \\ &=\; \begin{pmatrix} C_1 & \cdots & C_n \\ S_1 & \cdots & S_n \\ \mu_1 & \cdots & \mu_n \end{pmatrix} \begin{pmatrix} k_1 & & \\ & \ddots & \\ & & k_n \end{pmatrix} \begin{pmatrix} C_1 & S_1 & \mu_1 \\ \vdots & \vdots & \vdots \\ C_n & S_n & \mu_n \end{pmatrix} \\ K_P \;&=\; -\textstyle\sum_{i=1}^{n} k_i\sigma_{io}d_i \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{9}$$

The first matrix K_S is a product of three matrices SKS^T. S is an $3 \times n$ rectangular matrix, whose columns are the spatial vectors of the linear springs. The matrix S is called the *spatial configuration matrix* of the linear springs. K is an $n \times n$ diagonal matrix with positive stiffnesses of the springs on its diagonal. The product SKS^T is nothing more than the sum of the stiffness matrices of the individual springs expressed in the global frame of the hand.

The second matrix K_P comes from the second order variation of the compressions as the object rotates, $\partial^2\sigma_i/\partial\theta^2$. A more general expression of the position-dependent stiffness matrix K_P is:

$$K_P \;=\; \pm\sum_{i=1}^{n} f_{io}\,(\mathbf{p}_i\cdot\mathbf{k}_i)\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{10}$$

The sign is + (resp. −) if the fingers slide (resp. stick) on the grasping edges. The reader is referred to (Nguyen 1985b, 1986) for proofs. This sign makes outside-in grasps ~~more~~ less stable than inside-out grasps when there is friction and the fingers stick. The reverse holds for frictionless point contacts.

A force closure grasp implies that the set of contact wrenches spans the whole wrench space. If each contact wrench is generated by a linear spring, then the set of linear springs has spatial vectors that span the whole vector space. The spatial configuration matrix of the linear springs S has full row, and $K_S = SKS^T$ is positive definite. The compressions σ_{io} can be chosen small compared to the size of the grasped object, and $\sum k_i\sigma_{io}d_i$ is small compared to $\sum k_i\mu_i^2$. The stiffness matrix K of the grasp is approximatively equal to K_S, and is therefore positive definite. In other words, a force-closure grasp implies a stable grasp exists.

Corollary 6 *If grasp G is force closure, then we can always synthesize a set of linear springs at the points of contact, such that G is in stable equibrium.*

3.4 Compliance Center of the Grasp

The center of compliance of a planar grasp is the point about which the stiffness matrix K is diagonalized into two blocks. The grasped object behaves as though there are two linear and one angular springs attached to it. Figure 9.

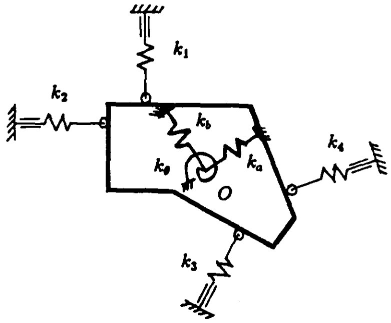


Figure 9: Compliance of the grasped object about its stable equilibrium.

The stiffness matrix K is diagonalizable with independent linear and angular springs if and only if:

$$\sum_{i=1}^{n} \mu_i k_i\,\mathbf{k}_i \;=\; \sum_{i=1}^{n} |\mu_i k_i|\;(sign(\mu_i)\,\mathbf{k}_i) \;=\; \mathbf{0} \tag{11}$$

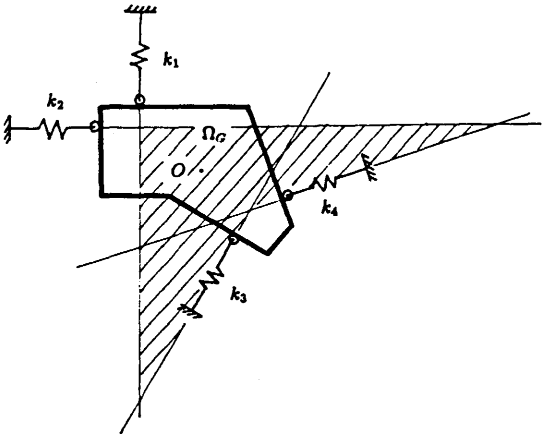


Figure 10: Compliance polygon of a grasp.

This condition is equivalent to the compliance center being inside a polygon delimited by the lines of action of the linear springs. Figure 10 shows the *compliance polygon* Ω_G within which the compliance center of grasp G must be.

We prove that if the grasp is force-closure then the compliance polygon always exists, and so equation (11) can be satisfied. Note that if grasp G is force-closure then the two cones generated by $(-\mathbf{k}_1, -\mathbf{k}_2)$ and $(-\mathbf{k}_3, -\mathbf{k}_4)$ counter-overlap in a non-zero convex polygon C_G, Figure 10. If we pick the compliance center O inside this convex polygon, then the springs k_1 and k_3, resp. k_2 and k_4, have negative, resp. positive, moments about O. One can check that there exists a positive linear combination of $-\mathbf{k}_1, \mathbf{k}_2, -\mathbf{k}_3, \mathbf{k}_4$ such that one walks counter-clock-wise along the boundary of the convex polygon bounded by the lines of action of the springs.

Corollary 7 *If grasp G is force-closure then the compliance polygon of grasp G, denoted Ω_G, is non empty. The compliance polygon Ω_G has boundary supports the lines of action of the linear springs.*

3.5 Finding Virtual Springs at the Fingertips

Figure 11 shows an interesting comparison between compliant fingertips that have passive and active stiffness. Examples of passive stiffness are real physical springs, like fingertips covered with rubber, or the Remote Center Compliance. With two fingers covered with rubber, the compliance center is not only fixed, but can only be inside a compliance rectangle with two diagonal corners at the two points of contact. The rectangle comes from the normal and tangential springs which model the rubber at the points of contact. The Remote Center of Compliance is a wrist built with fixed passive springs. The springs are designed such that the center of compliance is at the tip of the pin.

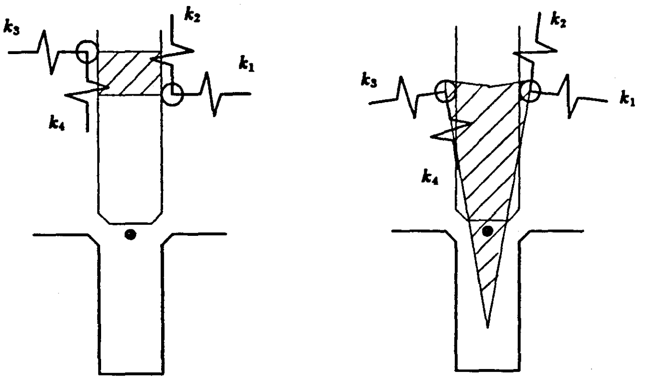


Figure 11: Pin and hole insertion with passive and active stiffness.

Active stiffness comes from stiffness control at the fingertips or at the joints. If the fingers have active compliance, then we can place and orient the virtual springs at the fingertips such that the compliance polygon overlaps the desired compliance center. So, we can not only make a force closure grasp stable, but also synthesize a compliance center for the grasp. The four virtual springs of a 2D grasp are typically computed in about 0.2 seconds on a Symbolics machine.

Algorithm 2 *Let G be a force-closure grasp with a desired compliance center O inside the compliance polygon Ω_G, defined by the lines of action of the n virtual springs. The n virtual springs at the points of contact can be synthesized so that G is stable as follows:*

1. Find a set of contact forces $(f_{1o}, \ldots, f_{no})$ such that force equilibrium is achieved. This is equivalent to solving a system of six equations with n unknowns.

2. From the desired compliance center O, find a set of positive spring constants $(k_1, \ldots, k_n)$ such that:

$$\sum_{i=1}^{n} \mu_i k_i \mathbf{k}_i \;=\; 0$$

where $\mathbf{k}_i$ and μ_i are respectively the direction and moment of the virtual spring k_i about the compliance center O. This is solving a system of two equations in n unknowns.

3. Check that the angular stiffness k_θ of grasp G is strictly positive:

$$k_\theta \;=\; \sum_{i=1}^{n} k_i \left(\mu_i^2 - \sigma_{io} d_i\right)$$

If not scale up the set of spring constants $(k_1, \ldots, k_n)$ and reduce the set of compressions $(\sigma_{1o}, \ldots, \sigma_{no})$, keeping the set of contact forces $(f_{1o}, \ldots, f_{no})$ unchanged, until k_θ is greater than zero.

4. Find the virtual compressions at equilibrium:

$$\sigma_{io} \;=\; \frac{1}{k_i} f_{io}$$

5. Output the set of spring constants $(k_i, \ldots, k_n)$, and the respective set of compressions $(\sigma_{1o}, \ldots, \sigma_{no})$ which model the set of virtual springs at the fingertips.

3.6 Controlling a Compliant Grasp

Figure 12 shows the relationships between force and instantaneous displacement at three different levels:

1. At the grasped object, we want to choose a compliance center and a stiffness matrix for grasp G such that the grasped object is stable and have restoring wrenches as follows:

$$\mathcal{F} \;=\; K_G \, d\chi$$

2. From the desired compliance at the grasped object, we would like to deduce the corresponding set of spring constants and compressions at the fingertips:

$$F \;=\; K_F \, dx$$

3. From the virtual springs at the fingertips, we then would like to derive the stiffness at all of its joints:

$$T \;=\; K_J \, d\theta$$

We can go further and derive the gains in the control loop of each joint, such that the above joint compliance is enforced. Or we can assume that each joint has a stiffness control loop with programmable stiffness.

From the kinematics of the grasp, the external and internal forces applied at the grasped object relate with the fingertip forces by the grasp matrix G^{-T} (Salisbury 1980, Salisbury and Craig 1981, Salisbury 1982). Similarly, from the kinematics of the linked fin-

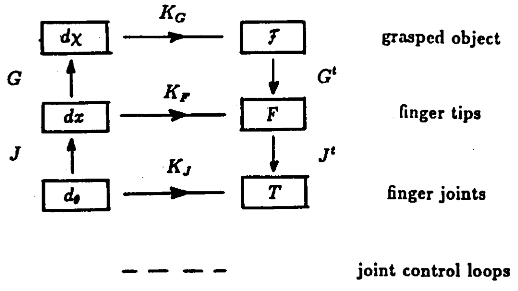


Figure 12: Linked chains and their loop equations.

gers, the force and velocity at each fingertip relate with its corresponding joint torques and velocities by the Jacobian matrices J^{-T} and J. We get loops from which we can derive easily the stiffness matrix of one level in terms of the stiffness matrix of another level. For example, given the desired compliance K_G at the grasped object, we deduce:

$$\begin{aligned} K_F &= G^T K_G G \\ K_J &= J^T G^T K_G G J \end{aligned} \qquad (12)$$

The grip matrix G^{-T} is an $n \times n$ matrix which relates the n external and internal forces applied at the grasped object to the n fingertip forces. K_G is an $n \times n$ generalized stiffness matrix at the grasped object which has the 6 linear and angular stiffnesses plus $n - 6$ internal stiffnesses. K_F is an $n \times n$ matrix which describes the springs at the fingertips, expressed in frames local to the points of contact. From the conservation of equivalent work at the object and at the fingertips, Salisbury and Craig deduced:

$$K_G = G^{-T} K_F G^{-1} \qquad (13)$$

The grip matrix G^{-T} has a $6 \times n$ block which is nothing more than the configuration matrix S. Note that S relates only the 6 external forces and moments at the grasped object to the n fingertip forces at the n springs. G^{-T} and S both capture the spatial configuration of the finger tip springs. The 6×6 stiffness matrix at the grasped object K, is a block of the generalized stiffnesss matrix K_G. The stiffness matrix K describes only the 6 linear and angular stiffnesses of the grasped object. We have shown that the stiffness matrix K of the grasped object is the sum of two matrices K_S and K_P:

$$\begin{aligned} K &= K_S + K_P \\ K_S &= S\, K\, S^T \end{aligned} \qquad (14)$$

$K = K_F$ if the local frames at the points of contact are oriented with the linear springs at the fingertips. So the grip matrix G^{-T} and the configuration matrix S both are only a first order approximation of the linkages between the grasped object and the fingertips. S is first order because its columns are spatial vectors $\hat{\mathbf{k}}_i$ which come from $\nabla \sigma_i$.

The second order approximation of the linkages between the grasped object and fingertips shows up as a position-dependent stiffness matrix K_P, which comes from the terms $\partial^2 \sigma_i / \partial \theta^2$. Our derivation of the stiffness matrix, Equations (9) and (10) is more accurate. The stiffnesses at the grasped object and at the fingertips are related not only by the conservation of total energy in the system, but also by the geometry about the grasp points. The geometry here is the point contacts which either stick or slide without friction on straight edges.

4 Grasping with Slip

4.1 Where Will The Fingers Slip To?

We need to find the direction towards which the fingers will slip, and calculate bounds on the places where the fingers will stick at, after slipping. A fingertip can be seen as being pulled by a virtual spring with the following stiffness behavior:

$$\begin{aligned} \mathbf{f} &= K(\mathbf{p}_o - \mathbf{p}) \\ \begin{bmatrix} f_n \\ f_t \end{bmatrix} &= \begin{pmatrix} k_n & 0 \\ 0 & k_t \end{pmatrix} \begin{bmatrix} \sigma_n \\ \sigma_t \end{bmatrix} \end{aligned} \qquad (15)$$

Figure 13 shows the regions where the fingertip will slip, stick, or loose contact. The stick region is a cone with angle $2\phi\frac{k_n}{k_t}$. It is defined as the region of the fingertip P where the contact force $\mathbf{f}$, generated by the virtual spring, points into the friction cone at the point of contact. The finger will slip towards the stick cone, and is pulled towards its bias position $\mathbf{p}_o$, which is fixed not relative to the grasped object, but relative to the base of the hand. When the object is displaced in the hand, the finger will stick and move with the point of contact, until it reaches one of the edges of the stick cone, then it slips.

The finger is guaranteed to stick inside the edge of contact, if the stick cone cuts in the interior of the edge of contact. This condition can be satisfied by a proper positioning of the bias position $\mathbf{p}_o$, and a proper ratio between the normal and tangential stiffnesses. The finger will not loose contact with the object, if the normal compression σ_{no} is always greater than the translational displacement of the object along this normal direction.

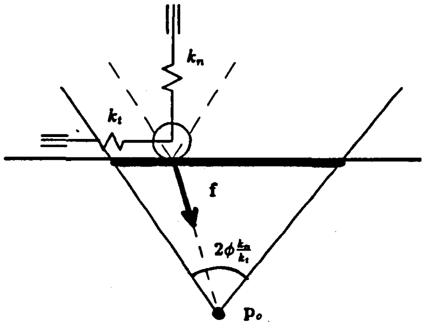


Figure 13: Stick and slip regions for the fingertip.

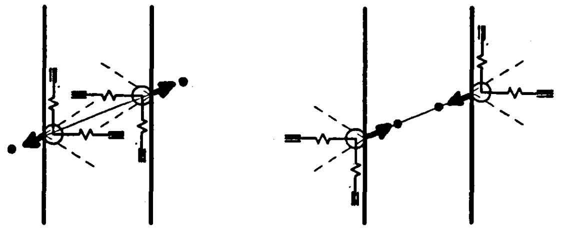


Figure 14: Places where the fingers will slip to.

Figure 14 shows two point contacts on a pair of parallel edges. Each point contact is pulled by a virtual spring towards its own bias position. As long as the stick cones intersect in the interior of the edges of contact, the fingers are guaranteed to stick on the contact edges. Better yet, the virtual springs can be synthesized such that the fingertips will stick inside the independent regions of contact. The bias positions of the virtual springs are constrained by the the equilibrium grasp condition, and so the compressions can only be scaled.

4.2 Effect of Slip on Force-Closure and Stability

We've seen that we can synthesize the virtual springs at the point contacts with friction, such that the fingers, if they slip, will always stick within the edges of contact. These fingers are pulled, independently from each other, towards their own bias positions. As the fingertip slips, the point of contact changes, and so either or both the spatial configurations of the normal and tangential springs change. We have shown that an equilibrium grasp with at least two point contacts with friction is also a force-closure grasp, if the contact forces point strictly inside the friction cones. In other words, if the fingers stick and the equilibrium is not marginal, then the grasp with friction is a force-closure grasp.

With stiffness at the contacts, a force-closure grasp implies a stable grasp, Corollary 6. So, after the fingers slip, the grasp at the points where the fingers stick is both force-closure and stable. To guarantee stability for rotations, the compressions at the linear springs must be small compared to the size of the object.

As the object slips, friction dissipates potential energy in the form of heat, and the total potential energy of the object is less. The potential energy of the object is the sum of the potential energy of the grasp, which is stored in the springs, and the gravity potential from the height of the object in the gravity field. In Figure 14.a, suppose the weight is perpendicular to the contact plane, and the fingers slips on the object, due to some impulse force with negligible work. There is no external work added into the system, and friction dissipates potential energy, so the two fingers will slide towards each other, and the grasp has a lower potential. So, as long as the contact edges are long enough, slip and disturbances make marginally stable grasp more stable.

5 Conclusion

We have presented a formal framework for analyzing and synthesizing grasps in 2D. Research is done in three areas: force-closure grasps, stable grasps, and grasps with possible slip at the contacts. The analysis and synthesis has been extended to grasps in 3D, and to curved objects. The reader is referred to (Nguyen 1986) for more details.

Acknowledgements

I would like to thank Tomás Lozano-Peréz, Kenneth Salisbury for many helpful discussions and for their constant encouragement. Part of the research on stable grasps has been done at the General Motors Research Laboratories during the summer of 1985.

Bibliography

Abel, J.M., Holzmann, W., McCarthy, J.M. "On Grasping Planar Objects With Two Articulated Fingers" Proc. IEEE Intl. Conference on Robotics and Automation, St. Louis, March 1985.

Baker, B.S., Fortune, S.J., Grosse, E.H. "Stable Prehension with a Multi-Fingered Hand: Extended Abstract" Proc. IEEE Intl. Conference on Robotics and Automation, St. Louis, March 1985.

Brady, M., Hollerbach, J.M, Johnson, T.L., Lozano-Péres, T., Mason, M.T. (editors) "Robot Motion: Planning and Control" MIT Press, Cambridge, 1982.

Brady, M., Paul, R. (editors) "Robotics Research" Vol. 1, MIT Press, Cambridge, 1984.

Brost, R.C. "Automatic Grasp Planning in the Presence of Uncertainty" Proc. IEEE Intl. Conference on Robotics and Automation, San Francisco, April 1986.

Cutkosky, M.R. "Mechanical Properties for the Grasp of a Robotic Hand" CMU-RI-TR-84-24, Carnegie Mellon Robotics Institute, 1984.

Erdmann, M.A. "On Motion Planning with Uncertainty" S.M. Thesis, Dept. of Elec. Eng. and Comp. Science, Massachusetts Institute of Technology, August 1984. Also as AI TR 810, MIT Artificial Intelligence Lab, 1984.

Fearing, R.S. "Simplified Grasping and Manipulation with Dextrous Robot hands" AI Memo 809, MIT Artificial Intelligence Lab, Nov. 1984.

Featherstone, R. "Spatial Notation: A Tool For Robot Dynamics" D.A.I Research Paper 213, University of Edinburgh, 1984.

Goldman, A.J., Tucker, A.W. "Polyhedral Convex Cones" in "Linear Inequalities and Related Systems", Kuhn, H.W. and Tucker, A.W. editors, Princeton Univ. Press, Princeton, 1956.

Hanafusa, H., Asada, H. "Stable Prehension By a Robot Hand With Elastic Fingers" Proc. of 7th Intern. Symp. on Industrial Robots, Tokyo, Oct. 1977. Reprinted in "Robot Motion: Planning and Control", Brady et al. editors, 1982.

Hanafusa, H., Inoue, H. (editors) "Robotics Research" Vol. 2, MIT Press, Cambridge, 1985.

Jacobsen, S.C., Wood, J.E., Knuttl, D.F., Biggers, K.B., Iversen, E.K. "The Version I Utah/MIT Dextrous Hand" in "Robotics Research" Hanafusa, H., Inoue, H. eds., 1985, pp. 301-308.

Jameson, J.W. "Analytic Techniques for Automated Grasps" Ph.D. Thesis, Dept. of Mechanical Engineering, Stanford University, June 1985.

Kerr, J.R. "An Analysis of Multi-Fingered Hands" Ph.D. Thesis, Dept. of Mechanical Engineering, Stanford University, Jan. 1985.

Lakshminarayana, K. "Mechanics of Form Closure" ASME Paper 78-DET-32, 1978.

Lozano-Péres, T. "Spatial Planning: A Configuration Space Approach" IEEE Transactions Computers, C-32, Feb. 1983.

Lozano-Pérez, T., Mason, M.T., Taylor, R.H. "Automatic Synthesis of Fine-Motion Strategies for Robots" Proc. Intl. Symposium of Robotics Research, Bretton Woods, NH. Aug. 1983. Also published in "Robotics Research", Vol. 1, Brady, M., Paul, R. editors, 1984.

Mason, M.T. "Compliance and Force Control for Computer Controlled Manipulators" S.M. Thesis, Dept. of Elec. Eng. and Comp. Science, Massachusetts Institute of Technology, May 1979. Also published as AI TR 515, MIT Artificial Intelligence Laboratory, April 1979. Reprinted in "Robot Motion: Planning and Control" Brady, M. et al. editors, 1982.

Mason, M.T. "Manipulator Grasping and Pushing Operations" Ph. D. Thesis, Dept. of Elec. Eng. and Comp. Science, Massachusetts Institute of Technology, May 1982. Also published as AI TR 690, MIT Artificial Intelligence Laboratory, June 1982.

Nguyen, V. "The Synthesis of Force-Closure Grasps in the Plane" AI Memo 861, MIT Artificial Intelligence Lab, Sept 1985. A condensed version is published in Proc. IEEE Intl. Conference on Robotics and Automation, San Francisco, April 1986.

Nguyen, V. "The Synthesis of Stable Grasps in the Plane" AI Memo 862, MIT Artificial Intelligence Lab, Nov 1985. A condensed version is published in Proc. IEEE Intl. Conference on Robotics and Automation, San Francisco, April 1986.

Nguyen, V. "The Synthesis of Stable Force-Closure Grasps" S.M. Thesis, Dept. of Elec. Eng. and Comp. Science, Massachusetts Institute of Technology, May 1986. Also published as AI TR 905, MIT Artificial Intelligence Laboratory, July 1986.

Ohwovoriole, M.S. "An Extension of Screw Theory and Its Application to the Automation of Industrial Assemblies" Ph.D. Thesis, Dept. of Mechanical Engineering, Stanford University, April 1980. Also published as AI Memo 338, Stanford Artificial Intelligence Lab, 1980.

Ohwovoriole, E.N. "On The Total Freedom of Planar Bodies With Direct Contact" ASME Transactions, 1984.

Reuleaux, F. "Kinematics of Machinery" Dover Press, New York, 1875.

Salisbury, J.K. "Active stiffness control of a manipulator in Cartesian Coordinates" Proc. IEEE Conference on Decision and Control, Albuquerque, Dec. 1980.

Salisbury, J.K. "Kinematic and Force Analysis of Articulated Hands" Ph.D. Thesis, Dept. of Mechanical Engineering, Stanford University, May 1982.

Whitney, D.E. "Quasi-Static Assembly of Compliantly Supported Rigid Parts" Journal of Dynamics Systems, Measurement, and Control, March 1982. Reprinted in "Robot Motion: Planning and Control", Brady, M. et al. editors, 1984.

OFF-LINE PLANNING FOR ON-LINE OBJECT LOCALIZATION

Tomás Lozano-Pérez
W. Eric L. Grimson

MIT Artificial Intelligence Laboratory
Cambridge, MA 02139

Abstract. Many robot applications require using sensors to locate objects whose initial pose is constrained but not exactly known. Most techniques for object localization assume that the object's pose is completely unknown. This paper describes a simple method for localizing known objects in a scene. We describe how an off-line computation that exploits constraints on the object's expected pose can be used to reduce the expected time for the on-line computation to localize the object. The objects treated here are modeled as polyhedra that, in principle, can have up to six degrees of positional freedom relative to the sensors.

0. Introduction

The problems of object recognition and localization have received a great deal of attention (see [Jain 86, Grimson and Lozano-Pérez 84,85] for reviews of the literature). Most approaches to recognition assume that the object's pose is entirely unconstrained. In most practical robotics applications, however, the uncertainty in part location is bounded to relatively small ranges. These constraints may come from knowledge of the feeding mechanisms or the physics of part stability. In most recognition systems, it is difficult to incorporate these type of constraints on the initial object pose. There have been a few systems where such information is readily incorporated, but the methods themselves have tended to be fairly complex [Bolles 76, Brooks 81, Goad 83, Baird 85, Faugeras and Hebert 83]. Nevertheless, the approach described here was significantly influenced by these previous methods, especially Goad's excellent paper. The localization algorithm described here is quite simple as is the mechanism for incorporating any available constraints on object pose. In the absence of any global constraints, the algorithm will still work, albeit more slowly.

The specific problem considered in this paper is how to locate a known object in a cluttered scene using sensors that provide dense position information. We assume that worst-case bounds on the pose of the object are available, as well as bounds on sensor measurement error. Our goal is to exploit the known bounds on object pose so as to reduce the amount of on-line computation required to localize the object. An important subgoal is that both the on-line and off-line methods should be simple enough to be easily implemented.

The method described here can be applied to both two-dimensional and three-dimensional sensing situations. In the two-dimensional case, objects have only three degrees of positional freedom relative to the sensor (two translational and one rotational). In this case, the sensors (and their pre-processors) are assumed to compute edges, that is, line segments in the scene. In the three-dimensional case, objects have up to three translational and three rotational degrees of freedom. In this case, the sensors (and their pre-processors) are assumed to compute planar patches in the scene. We do *not* deal with the general case in which only two-dimensional data is available but the object has more than three degrees of freedom. For the sake of brevity, we limit our discussion to the three-dimensional case; the specialization to two-dimensions is straightforward.

We assume that the objects of interest can be modeled as sets of planar faces. Only the individual plane equations and dimensions of the model faces are needed. No face, edge, or vertex connectivity information is required; the model faces do not even have to be connected. Because of this, the method can be applied to curved objects that are readily approximated by planar patches. Of course, such planar approximations are not adequate for all cases, for example, objects of high curvature or multiply curved surfaces.

We assume the availability of a sensor and pre-processor that can compute the planar patches present within some given rectangular sub-window of the scene. A great deal of work in computer vision has been dedicated to solving this problem of obtaining depth from two-dimensional visual data (see [Horn 86] for a representative sample). Other less computationally-intensive methods exist for obtaining the required patches, notably range sensing (see [Jarvis 83] for a review). We will not address this problem further.

In section 1, we present the basic on-line localization method. In section 2, we describe the off-line computations required for the on-line method. In section 3, we discuss the method and point out areas for further work.

1. A simple on-line localization algorithm

The process of localization is carried out in three steps:

- The first step is to identify possible assignments of sensed data to model faces consistent with a set of measurements derived from the model. This is the crucial step.
- The second step is to identify the pose of the object from each of these assignments.
- The third step is to pick the solution that best matches all the available data.

At this level of description, the method is similar to the *interpretation tree* method described in [Grimson and Lozano-Pérez 84, 85] and draws results from that earlier method. The method described in this paper differs in the first of these steps, while the earlier method does not do any hypothesis verification, the method described here goes very early into a hypothesize/verify cycle.

The current method is geared to situations where the set of possible matches of data patches to model faces can be constrained a priori. The goal is to reduce the combinatorics of the matching process in the earlier methods by exploiting the global position and orientation constraints.

1.1 Sensor Input

The on-line algorithm uses as input a list of the planar patches present in each of a set of windows specified by the off-line planner (see section 1.2). The patch must be completely within the window to be elegible. Each patch obtained by the sensor is characterized by a list of points and a plane equation in the form $\mathbf{n} \cdot \mathbf{x} = c$, where $\mathbf{n}$ is the unit normal to the plane. All the points are required to be inside the same face of the object. We represent patches by a set of points, instead of polygons, so as to simplify the processing and to accommodate a wide variety of sensors, including sparse sensors such as tactile sensors.

It is possible to use information about any patch feature, such as, size, color, reflectivity, and texture, to limit the possible model faces that a patch could match. The availability of these measurements can significantly reduce the combinatorics of the matching process. This is straightforward extension and we do not discuss it further.

1.2 Constraints from the off-line planner

The information from the off-line planner is used to reduce the combinatorics of the matching process. There are three sorts of constraints that are useful for this purpose: (1) restrictions on the data patches that can be involved in match a given model face, (2) restrictions on the model faces that can be involved in a solution, and (3) a ranked list of legal initial hypotheses. In particular, the off-line planner provides the following information for the on-line matching algorithm:

- A list of windows — each data patch is assigned to one or more rectangular windows in the scene.
- A list of matching constraints for each model face — each face requires that the patch that is matched to it be drawn from a specific window, furthermore there are constraints on the normal of the matching data patch.
- Ranges of possible measurements between pairs of model faces — these are used to check the consistency of assignments of patches to model faces.
- A list of visible face lists — these lists are indexed by the face pairs in the initial hypotheses and describe the faces visible if a particular hypothesis is correct.
- An ordered list of face pairs — these pairs are used to form initial hypotheses as to the object pose; they are ordered by size of the faces.

The windows could, in principle, be used to restrict the application of the sensor pre-processing to subsets of the scene. The windows, however, may overlap arbitrarily. In the general case, it is preferable to apply the sensor pre-processing to the whole scene and then assign the data patches to the corresponding windows. That is the strategy we have used. In simple situations where only a few windows are needed, it makes sense to limit the pre-processing to these windows. In any case, the main purpose of the windows is not to reduce the sensory pre-processing but to reduce the combinatorics of the matching process.

The key information computed by the off-line planner are the constraints on face assignments. Each face is restricted to matching a patch from a specific window. In fact, the windows are actually computed as the loci of particular model faces. The windows enforce the position constraints on faces derived from the global pose constraints. Associated with each face are constraints on the data patch normal that can match that face. This constraint is expressed as a list of cones, that is, center vectors and a minimum value for the cosine of the angle between the patch vector and the specified vector (see section 2). Also associated with each face is a window where the matching data patch must appear.

The off-line planner identifies those pairs of model faces that can appear in some view of the object consistent with the pose constraints. Of these pairs, the ones with significantly different orientation can be used to obtain an initial solution for the object's orientation. It is these pairs that drive the initial phase of the matching algorithm described in section 1.3.

The off-line planner also computes all the sets of model faces that can be simultaneously visible given the known pose constraints. These are used during the verification phase of the matching algorithm.

The object model provided by the user is described by a set of planar faces. Each face is specified by a polygon, which may be non-convex, and a plane equation. This form of the model is not particularly useful to the matcher. The off-line planner computes from this model three tables described below. Let $\mathbf{p}_i$, $\mathbf{p}_j$ and $\mathbf{n}_i$, $\mathbf{n}_j$ be respectively points on and normals to faces i and j.

- Distance — For each pair of model faces, i and j, the upper and lower bounds on distances between all possible pairs of points $\mathbf{p}_i$ and $\mathbf{p}_j$.
- Angle — For each pair of model surfaces, the angle between the face normals $\mathbf{n}_i$ and $\mathbf{n}_j$.
- Distance vector — For each pair of model surfaces, the upper and lower bound on the values $(\mathbf{p}_j - \mathbf{p}_i) \cdot \mathbf{n}_i$ and $(\mathbf{p}_j - \mathbf{p}_i) \cdot \mathbf{n}_j$, that is, the component along the face normals of vectors connecting the faces.

In each case, the values in these tables take into account the error bounds on measuring both the patch normals the positions of points. Algorithms for computing these tables are found in [Grimson and Lozano-Pérez 84]. The combination of these three types of measurements has been shown to be quite powerful in discarding invalid matches even in the presence of significant measurement error [Grimson and Lozano-Pérez 84, 85].

1.3 The matching algorithm

The matching algorithm is very simple and is based on the assumption that the set of data patches that can match a given model face is relatively small (this is up to the off-line planner to guarantee). The method works by considering each of the possible model face pairs determined by the off-line planner as suitable for constructing an initial hypothesis. The first step is to find all the pairs of data patches that can be matched to the face pair under consideration. The data patch pair must satisfy the following conditions:

- Each data patch must come from the window specified by the off-line planner for the corresponding face.
- The measured normals for the data patches must be within one of the cones associated with the corresponding face.
- The three sets of measurements (distance, angle, and distance vector) for the data patch pair must be consistent with those of the model face pair.

We could find such data pairs by looking at all combinations of patches, but that would be time consuming. It is straightforward to improve the expected performance by using hashing. Simply pre-process all pairwise combinations of data patches and place them into buckets based on the angle between their normals. Then, given a candidate pair of model faces, one can in constant time limit the data patch combinations to those whose angle is within the measurement error of the angle between the model faces. Only these data pairs need to be subjected to further testing.

Having obtained the feasible matches to a model face pair, the next stage is to use them to obtain supporting evidence for particular combinations of visible faces. Note that the match of two data patch normals $(\mathbf{n}_i, \mathbf{n}_j)$ to two independent model face normals $(\mathbf{m}_i, \mathbf{m}_j)$ determines uniquely (up to a sign) the rotation matrix R, where $R\mathbf{x} + \mathbf{p}$ is the transformation that maps points, $\mathbf{x}$, in the model coordinate system to vectors in the sensor coordinate system.

The rotation matrix R can be easily computed in the form $Rot(\mathbf{r}, \theta)$, where $\mathbf{r}$ is the axis of rotation and θ is the rotation angle. The axis of rotation is the unit vector in the direction

$$(\mathbf{m}_i - \mathbf{n}_i) \times (\mathbf{m}_j - \mathbf{n}_j)$$

and the angle of rotation can be obtained from these two relationships

$$\cos\theta = 1 - \frac{1 - (\mathbf{n}_i \cdot \mathbf{m}_i)}{1 - (\mathbf{r} \cdot \mathbf{n}_i)(\mathbf{r} \cdot \mathbf{m}_i)}$$

$$\sin\theta = \frac{(\mathbf{r} \times \mathbf{n}_i) \cdot \mathbf{m}_i}{1 - (\mathbf{r} \cdot \mathbf{n}_i)(\mathbf{r} \cdot \mathbf{m}_i)}$$

For a detailed derivation see [Grimson and Lozano-Pérez 84].

Using R we can compute for each potentially visible face on the object, the nominal data patch normal that can match that face. We must take into account measurement errors when matching the predicted normal to actual measured normals.

The match also constrains the translation vector, $\mathbf{p}$, to be on a line determined by the planes of the two measured patches. The range of values of $\mathbf{p}$ can be determined as follows: Let the equation of the patch planes be of the form $\mathbf{n}_i \cdot \mathbf{x} = c_i$ and the equation of the model planes be of the form $\mathbf{m}_i \cdot \mathbf{x} = d_i$. Then we can write $\mathbf{p}$ as a one parameter family of vectors:

$$\mathbf{p} = \alpha\mathbf{n}_i + \beta\mathbf{n}_j + \gamma(\mathbf{n}_i \times \mathbf{n}_j)$$

where γ is the free parameter and

$$\alpha = \frac{(c_i - d_i) - (\mathbf{n}_i \cdot \mathbf{n}_j)(c_j - d_j)}{1 - (\mathbf{n}_i \cdot \mathbf{n}_j)^2}$$

$$\beta = \frac{(c_j - d_j) - (\mathbf{n}_i \cdot \mathbf{n}_j)(c_i - d_i)}{1 - (\mathbf{n}_i \cdot \mathbf{n}_j)^2}$$

The parameter γ can be constrained by requiring that all the points in the data patch be mapped by the resulting transformation to be *inside* the model face. In our implementation, we compute bounds on γ by considering the set of translations that map the center of area of the patch onto each of the vertices of the face.

Let each potentially visible face have plane equation $\mathbf{m}_k \cdot \mathbf{x} = d_k$. Given the range of $\mathbf{p}$ we can compute not only the predicted normal, $R\mathbf{m}_k$, for a matching data patch but also the range of values of the offset c in the plane equation of the data patch: $c = d_k + (R\mathbf{m}_k) \cdot \mathbf{p}$, where $\mathbf{p}$ is parameterized by γ.

We also know, from the off-line computation, the window in which the data patch must appear. Thus, the initial verification process consists of checking the windows for patches satisfying the position and orientation constraints. Of course, once a third independent patch is found, the position and orientation can be determined uniquely up to the sensing error.

Once a feasible pairing of model faces and data patches, together with the corresponding object pose(s), is found there still remains a detailed verification process. This proceeds in two steps: (1) check that the patch points are all mapped to the inside of the face polygon by the computed transformation and (2) check that the patches measured in each of the windows are consistent with the computed model pose, that is, that no data patch has been measured to be *below* where the model predicts a surface to be. Note that if a data patch is above a predicted surface then this is neutral evidence, since this situation could arise from occlusion. Nevertheless, one wants to ensure that a hypothesis accounts for a sufficiently large percentage of the object's measured surface.

The matching process described above should be carried out in "depth-first" fashion, verifying each hypothesis as it is generated, rather than in "breadth-first" fashion, finding all the hypotheses and then verifying each one. Doing the matching depth-first allows us to terminate the matching once an adequate match is found. Observe that if the object were completely visible, each and every initial hypothesis should lead to a complete and correct interpretation. This points out the redundancy of examining all the initial hypotheses. In practice, face occlusion requires that we be ready to examine all the possible pairings even though we seldom will. Note that in the depth-first mode the ordering of the hypotheses by likelihood of locating the faces can significantly reduce the expected time to verify the hypothesis.

Let us quickly recap the flow of control in the on-line matcher. First, the sensing is done within a window guaranteed to contain all the faces. All the data patches are found and allocated to any windows that completely contain the patch. The matcher then proceeds through its list of legal face pairs attempting to form initial hypotheses and then verify them. Once an acceptable hypothesis is found, the matching stops. For each face pair, the matcher looks for a patch pair that has the appropriate angle between the normals. Each patch must be in the appropriate window for the matching face and satisfy the orientation constraint. Also, the assignment of the two faces to the two patches must pass all the pairwise constraints derived from the model. Only after all these test are satisfied does the matcher have a valid initial hypothesis. The next step is to verify the hypothesis. First, a potentially visible face is chosen, its orientation and range of displacements predicted and then its presence verified. If a matching patch is located, the pose of the object can be computed and all the other face predictions checked. A completeness score (based on correctly predicted area) is computed for the hypothesis.

1.4 Complexity

The matching method described above can be used with a single window and no off-line computation. In that case it has a worst-case complexity $O(n^3m^3)$ where n is the number of faces in the model and m is the number of data patches in the scene. This naive bound is simple to derive: The outer loop considers all n^2 permutations (with repetition) of the n model faces, for each such pair it tests for consistency all $m(m-1)/2$ pairwise patch combinations (without repetition). In the worst case, each pair of these $O(n^2m^2)$ combinations needs to be examined further. The further computation requires checking for each of the n faces of the model, each of the m data patches.

This bound for matcher performance is for the very worst case. The angle bucketing described above should improve the expected performance. In this version of the algorithm, one still has the $O(n^2)$ outer loop, but one expects significantly fewer than $m(m-1)/2$ data patches will have to be considered. Having found a pair of model faces and data patches, they can be tested for consistency with the distance, angle, and distance vector constraints. This will further reduce the combinatorics of the method. Of course, the actual performance will depend on the object model and the measured data. The worst performance will be for a very symmetric object such as a cube.

Thus far, we have not considered the effect of having a priori bounds on the object pose. In the absence of such bounds we are limited to using coordinate-frame independent constraints, such as angles between pairs of faces. Once we know bounds on the pose of the object, we can constrain individual matches. For example, suppose we knew that the object was in some particular stable pose on a horizontal table, then the z component of each face normal is known within the measurement error. Given a candidate model face, only data patches whose normals have z components in the appropriate range need be considered as matches. This reduces the effective number of data patches to be the expected number of data patches with the same z component.

The matching algorithm described in section 1.3 is aimed at exploiting this type of constraint. The windows enforce global position constraints and the angle cones associated with each model face enforce global orientation constraints. The purpose of these constraints is to minimize the number of data patches that can match a model face. In the ideal case, only a single patch can match each face and the whole algorithm boils down to finding three visible data patches.

These constraints, therefore, affect only the inner loop of the matcher; there still remains a potentially $O(n^2)$ outer loop. The algorithm attempts to reduce the expected number of initial hypotheses that need to be verified in two ways. The pre-processing of the pairs of model faces serves to capture other global constraints such as the fact that parallel faces cannot be used for the initial hypothesis and that not all pairs of faces can be visible simultaneously. These constraint tend to reduce the number of initial pairs well below the n^2 value. The constraints derived from the model (distance, angle, etc.) are used to prune out those initial hypotheses that remain before any detailed prediction and verification is done.

1.5 Example

Here, we consider a simple example of the matching algorithm; figure 1 outlines the stages of processing. Figure 1A shows three-dimensional depth data of a very cluttered scene obtained with a structured light range sensor, figure 1B shows the model of the target object given by the user, figure 1C shows the result of pre-processing the depth values to obtain planar patches, figure 1D schematically suggests the matching process of patches to faces, and figure 1E shows the resulting localized object superimposed on the original scene.

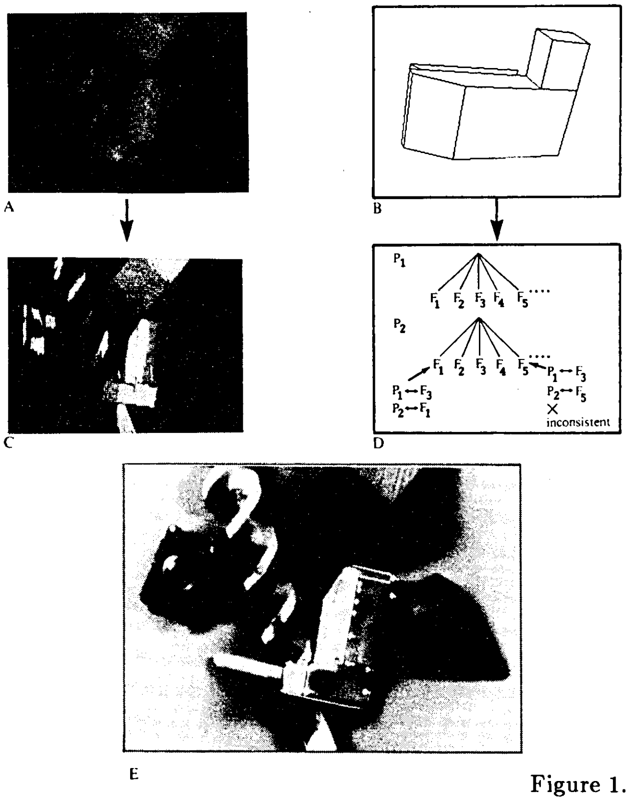


Figure 1.

Note that the matching algorithm can operate with no bounds on the pose of the input object. In that case, there is a single window within which all data patches can be found, also, there are no global constraints on the orientation of matching patches. The constraints that remain in effect are the pairwise matching constraints derived from the model, the pairwise visibility constraints, and the constraint that face pairs for the initial hypothesis have independent normals. Under those circumstances, the algorithm finds 865 legal initial hypotheses in the scene shown in figure 1. Recall that an initial hypothesis is an assignment of a pair of model faces to data patches that satisfy all the constraints. Note that for this example $n = 12$ and $m = 30$, so the worst case is 62,640 potential hypotheses. This illustrates that even in the absence of global pose constraints, the coordinate-frame independent constraints are quite powerful.

Continuing the example, assume that the target object's pose is constrained so that all the patches are within a smaller window (say with $m = 15$). This constraint reduces the number of potential initial hypotheses by roughly a factor of four (15,120). Furthermore, assume two specific faces can be constrained to smaller sub-windows (see figure 2), each with at most five patches in it ($m = 5$). Then, the upper bound on the legal number of initial hypotheses, even without considering the other geometric constraints is cut by approximately a third (10,540). We can also exploit the fact that the two front faces and two back faces can never be simultaneously visible, similarly the two top faces and the bottom face. This eliminates an additional

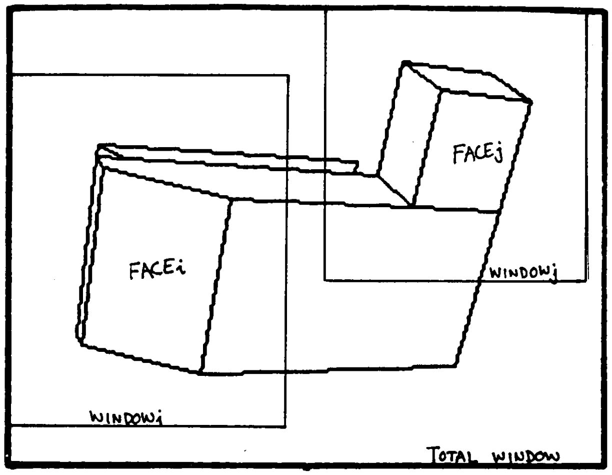


Figure 2.

1260 possible hypotheses. All of these numbers address the worst case bound; as shown above the actual number of hypotheses is typically substantially less. In the example shown above, of the 865 legal pairwise interpretations, 642 satisfy the window constraints described above.

In addition to the window constraints, there are global face orientation constraints that reduce the expected number of patches that can be assigned to the faces. These constraints will effectively reduce the value of m within the windows. As we saw above, reducing m produces large reductions in the possible number of hypotheses. For example, if the expected value of m can be uniformly reduced to 5, then the upper bound on the hypotheses (without the visibility constraint) is only 1040. The visibility constraint prunes an additional 120 potential hypotheses. The actual implementation actually had to consider 266 initial hypotheses.

2. The off-line planner

The efficiency of the off-line planner is not as crucial as that of the matching algorithm. Accuracy is not essential either; the only requirement is that the bounds be conservative. Therefore, we have adopted essentially brute-force sampling techniques for all the computations of the off-line planner.

2.1 Windows for the faces

Windows can be computed as rectangular bounds on the loci of the face's vertices over the range of legal poses. That is, the legal orientations of the object are sampled and the position of each face's vertices are computed. An enclosing rectangle is computed for the vertices of each face and updated so as to include all the point positions. This rectangle is then swept over the range of legal x, y translations of the object (also a rectangular range).

This computation generates a rectangular window for each face. Windows that overlap by some large fraction of their area are merged to form a single window. This is done to reduce the amount of computation that the sensory pre-processor needs to do when it assigns patches to windows. A window that contains all the other windows is also computed; this window bounds the area where sensor processing needs to be done.

2.2 Face orientation constraints

The global orientation constraints on the faces are represented as a set of cones for ease of testing. This can also be computed while sampling the range of object orientations. We proceed by tesselating the Gaussian sphere into the faces of an icosahedron. This gives us twenty uniformly distributed orientation buckets, represented as cones whose center vectors are the normals of the iscosahedron's faces. As we sample the orientations of the object, we keep track of which of these buckets the normal of each face falls in. This list of buckets is a conservative representation of the constraint on the orientation of the data patch that can match a face.

2.3 Visible faces

Potential visibility is determined by examining the sign of the component of the face normal along the sensing direction. This is what is known in graphics as "back face" elimination, rather than a full visible surface computation. The combinations of visible faces are also computed as the orientations of the object are sampled.

2.4 Face pairs

The planner constructs the list of all pairs of simultaneously visible faces whose normals make an angle greater than a predefined threshold ($\pi/6$ in our case). The list is ordered so that pairs involving large faces are at the front of the list. These are the most likely faces to be visible in spite of occlusion.

2.5 Pairwise geometric constraints

See [Grimson and Lozano-Pérez 84] for a description of how these constraints can be computed for polyhedral models.

3. Discussion

The algorithm presented in this paper is primarily based on a prediction/verification style: predicting the orientation and positions of faces in the model and verifying the presence of consistent data patches. The hypotheses are driven off of pairwise matches of faces to patches and it is the number of possible matches at this level that determines the performance of the method. In the unconstrained case, the algorithm needs to examine a large number of initial hypotheses. By considering some simple constraints arising from global constraints on the object pose, the number of legal initial hypothesis is significantly reduced to a manageable number. Recall that the actual prediction and verification process is already reasonably efficient so that hundreds of hypotheses can be evaluated in a matter of seconds.

Initial indications are that the method performs extremely well when the pose of the object is tightly constrained. As the range of possible poses grows, the performance reaches a plateau dictated by the performance of the algorithm in the absence of constraints. Of course, this limiting performance is a non-linear function of the number of faces and number of data patches. Future work will attempt to derive better expected bounds on the performance of the algorithm.

The main advantage of the method described here is its simplicity. The major limitation of the algorithm is its reliance on planar face approximations; extending the approach to curved objects would not be straightforward. Another disadvantage is the relatively weak coupling between the sensor processing stage and the matching. In principle, a tighter coupling could be implemented so that the amount of sensor processing could be reduced. On the other hand, the simplicity of the algorithm is due largely to the fact that the pre-processing is simple and uniform.

The on-line matching algorithm described in section 1 has been implemented on a Symbolics Lisp Machine; the off-line planner is currently being implemented. The testing of the on-line method has been done with simple window and orientation constraints specified by the user.

Acknowledgments

We thank Philippe Brou for kindly providing the laser ranging system with which we obtained the data reported in figure 1. This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's Artificial Intelligence research is provided in part by a grant from the System Development Foundation, and in part by the Advanced Research Projects Agency under Office of Naval Research contracts N00014-80-C-0505 and N00014-82-K-0334.

Bibliography

Baird, H. 1986. *Model-based recognition.* MIT Press, Cambridge, Ma.

Bolles, R. C. 1976. Verification vision within a programmable assembly system. Stanford Artificial Intelligence Laboratory Memo 295.

Brooks, R. A. 1981. Symbolic reasoning among 3d models and 2d images. *Artificial Intelligence.* 17(1-3):285-348, August.

Faugeras, O. D. and Hebert, M. 1983. A 3D recognition and positioning algorithm using geometrical matching between between primitive surfaces. *Proc. Eigth Int. Joint Conf. on Artificial Intelligence*, Karlsruhe, W. Germany, 996-1002, August.

Goad, C. 1983. Special purpose automatic programming for 3d model-based vision. in *Proceedings of DARPA Image Understanding Workshop.*

Grimson, W. E. L., and Lozano-Pérez, T. 1984. Model-based recognition and localization from sparse range or tactile data. *Int. J. Robotics Res.* 3(3):3-35.

Grimson, W. E. L., and Lozano-Pérez, T. 1985. Recognition and localization of overlapping parts from sparse data in two and three dimensions. *Proc. IEEE Conf. on Robotics and Automation*, St. Louis, Mo., 61-66, March.

Horn, B. K. P. 1986. *Machine Vision.* MIT Press, Cambridge, Ma.

Jarvis, R. A. 1983. A perspective on range finding techniques for computer vision. *IEEE Trans. on Pattern Analysis and Machine Intelligence.* 5(2):122-139, March.

Jain, R. 1986, Three-dimensional object recognition. *Computing Surveys.*

AML/X: A Programming Language for Design and Manufacturing

Lee R. Nackman
Mark A. Lavin
Russell H. Taylor
Walter C. Dietrich, Jr.
David D. Grossman

Manufacturing Research Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York, 10598

Abstract

AML/X is a modern general purpose high level programming language, aimed at applications in manufacturing and computer aided design. It includes features for both conventional and object-oriented programming. The AML/X interpreter is implemented in C and has been ported to IBM 370, Motorola 68000, and IBM PC hardware, running under CMS, UNIX, XENIX, and DOS. This paper describes the rationale for AML/X and gives an overview of the language itself.

1.0 Introduction

Architects of industrial automation systems make a trade-off between two goals: ease of use and flexibility. A decade ago, when computers were far more expensive, these goals were seen as competing alternatives.

Historically, many industrial automation systems limited their flexibility to the minimum requirements of some (hopefully) large class of users, but no more. Those users then specified applications with a minimal language, which was simple and easy to learn. An example of this approach in robot programming was teaching by showing, in which the user manually moves a robot through a sequence of motions, recording them for later replay[1]. The disadvantage is the lack of a growth path: once the user's needs exceed the flexibility provided, the system becomes ineffective.

Clearly, there is significant benefit in providing systems with much greater flexibility. Generally, this flexibility is achieved by deferring certain choices until as late as possible. For example, a robot achieves more flexibility than fixed automation by deferring the choice of motions from the time the equipment is designed until it is actually used.

Flexibility gained by deferring choices implies the need for a much richer *language* in which to specify the choices to be made. The challenge for the system architect is to layer this language so that ease of use is not sacrificed. As computers decline in cost, this approach of layering is becoming an increasingly practical means of combining flexibility with ease of use.

One begins at the bottom layer with as much flexibility as one can afford (more on this later) and a language for specifying the remaining choices. The language must be expressive enough to allow the most sophisticated users to take advantage of the flexibility included. Perhaps less obvious, but equally important, the language must provide mechanisms for composing higher layers, each essentially a new language, with less flexibility and fewer remaining choices than the preceding layer. The highest layers, those with the least flexibility, provide ease of use for unsophisticated users along with potential growth to lower, more flexible layers.

This is the context in which we have designed AML/X, a general purpose programming language tailored for use in manufacturing and computer aided design. Its roots are in AML[2], a programming language originally developed for use in a research robot system[3] and subsequently made available as the programming language of the IBM 7565 Manufacturing System. AML also saw use as the base language for AML/V, an industrial machine vision programming system built as an extension to the research robot system[4]. AML/X is the result of a major redesign of AML and is part of the programming environment for our research activities in robotics[5], machine vision, and computer aided design. As such, it is likely to continue to evolve. It is also the basis for AML/2, the programming language for IBM's new 7575 and 7576 Manufacturing Systems.

This paper describes the rationale for the design and implementation of AML/X, provides a brief overview of the language, and illustrates its use. For brevity and clarity this paper emphasizes the use of AML/X in robotics; however, AML/X is actively used in our research in machine vision, workcell layout, and computer aided design. The next section outlines our objectives and their influence on the design of AML/X. This is followed by an overview of the basic structure and facilities of AML/X, just enough to provide the flavor of the language and to be able to follow subsequent examples. (A detailed description of AML/X is available in [6].)

2.0 Design Rationale

2.1 Guiding Principles

Designing a programming language requires balancing the many conflicting requirements of the anticipated user community. To help achieve this balance we have tried to follow a few broad, general principles, against which specific design decisions could be evaluated.

First among these is Hoare's dictum[7] that the language designer's task is one of "consolidation, not innovation." In keeping with that principle, we have not introduced any radically new constructs in AML/X. Instead, we have chosen from

CH2345-7/86/0000/0145$01.00 © 1986 IEEE

among constructs and ideas that have been tested in other languages and have tried to integrate them into a consistent, coherent whole. LISP, APL, and SMALLTALK have had an especially strong influence.

The second guiding principle is to prefer general purpose constructs to those meeting specific, limited, application needs. A corollary is that it must then be easy to provide layers that adapt general purpose features to meet the specific needs. For example, AML/X's interactive debugging tools are a small extension of a very general exception handling mechanism. These principles keep the language a coherent whole, rather than an accumulation of features. In practice, this has been very difficult. When an important user requests a specific new feature, it is hard to say no.

The third principle is orthogonality, which demands that separate features be separate, that is, that the legality and meaning of the use of a construct should be independent of its use in combination with some other construct. Orthogonality aids in achieving generality, but also makes it easier to write obscure programs. It can also be difficult to achieve in a practical implementation. For these reasons, we have occasionally violated orthogonality, but not without careful thought.

These three principles are helpful, but they do not substitute for an understanding of the anticipated applications of the language.

2.2 Language Design Criteria

We have designed AML/X keeping in mind three major areas of use: robot programming, machine vision programming, and computer-aided design (primarily of mechanical objects and assemblies, see e.g. [8]). An analysis of the requirements of robot programming and a survey of existing robot programming systems appears in [9]. Requirements for machine vision programming and an abstract language for vision programming (which could be embodied in many ways) are described in [10]. Some issues in programming languages for CAD systems and a particular language design are discussed in [11].

In all three areas, flexibility has historically been sacrificed, primarily to achieve ease of use. As a consequence, existing systems rarely exploit the intrinsic similarities between these domains. For example, all three domains deal with mechanical objects, assemblies, geometric algorithms, and transformations in 2 and 3 dimensions.

In the future, applications will expand in all three domains, and there will be increased need in industry to integrate them. It is therefore desirable to have a single language that can effectively deal with robotics, vision, and CAD.

In the following subsections we discuss the implications of the intended uses of AML/X.

Classes of Users: Roughly speaking, we anticipate three classes of users. *End users*, such as manufacturing engineers or mechanical designers, are the people who specify the operations to be carried out by the automation or CAD system. They typically have little programming skill, but need to be able to "chain together" sequences of pre-existing, relatively high-level commands, possibly including some simple control flow. *Application developers* write application packages, programs that provide facilities for use by a small class of end users. In a sense, an application package decreases generality while increasing ease of use for a particular class of users. Application developers typically have some programming skill and a deep knowledge of the application area. Often they produce new application packages by building a layer on top of an existing, more general program. *Application development environment developers* write the (typically) large systems application developers use. Robot programming and CAD systems are examples. They need both very high programming skill and a reasonably deep knowledge of the application area. This mix of users and the way they work has had an important influence on the language design.

Most significantly, the language must be a modern, general purpose language well-suited to developing large, layered systems. It must support the definition and use of abstract data types and have facilities for manipulating strings, complex data structures, and symbolic information. A general exception handling mechanism is essential so that an application can "catch" exceptions from lower layers, thus avoiding the incomprehensible error messages to end users that would otherwise result.

At the same time, the language must be simple, or at least have simple subsets, so that it is accessible to end users. However, as Hoare has observed[7], subsetting is simpler said than done, because an errant program can invoke some language feature outside of the subset. It is therefore important to be able to selectively disable or hide portions of the language. Moreover, the use of subsets increases the need for consistency and lack of special cases to provide an easy growth path for users who become more sophisticated over time.

Robot Programming: Our experience with several generations of robot system[3, 12, 13] has confirmed that specification of manipulator motion represents only a small, though very important, proportion of the total code required for a working robot application. Other components include I/O for auxiliary devices and communications, operator interfaces, calibration and setup routines, bookkeeping, access to manufacturing data bases, etc. The failure of special-purpose manipulation languages such as AL [14] to provide adequate support for these other components has led to renewed interest [e.g., 15] in the use of standard general-purpose languages for robot programming. The approach in AML, carried over to AML/X, was to design a new general-purpose language whose design trade-offs make it convenient for automation programming.

Robotic applications *do* have many special characteristics. Manipulator motion specifications and calibration packages rely heavily on geometric calculations, and means must be provided for expressing these conveniently. Most applications require some level of concurrency. Application programs are usually debugged (and often written) "on-line" and are hard to restart, thus making support for interactive programming especially important.

The structure of programs is often rather different from that found in other, more "algorithmic" domains. Generally, the main line of an application program is quite straightforward, and consists of little more than a sequence of commands, with most of the useful work being done as a "side effect". Unfortunately, the vagaries of the physical world [16] cannot be ignored. Our experience has been that robust programs can easily have three to five times as much error testing and recovery as main line code. Often, the error recovery actions are both context dependent and safety related. For example, it may be appropriate to freeze manipulator motion if a gripper feedback sensor fails *unless* the hand happens to be in a furnace at the time. These considerations have led us to place special emphasis on powerful exception handling mechanisms.

Machine Vision: Many of the considerations for robot programming also apply to machine vision [4, 10]. Indeed, one of our objectives was to promote better integration of robot and vision programs. The large amount of data that must be handled in many vision programs means that it is especially important to provide data representations and execution primitives with efficient low-level implementations while still providing great expressive power at higher levels. Beyond this, the language should permit suitable interfaces to special-purpose computational hardware.

Computer-aided Design: The requirements for computer-aided design overlap those for robotics and vision. We require good support for layering, support for interaction at high levels, efficient low-level code execution, and extensibility. Requirements that are especially acute in CAD applications include support for self-describing objects, efficient and accurate numerical computation, and the ability to build up extremely complex data structures.

Other criteria: Since we were generally pleased with our experience with AML, we decided early in our design effort to maintain its general flavor, but not to require strict upward compatibility. The result is that most nontrivial AML programs will not run unaltered on AML/X. One reason for this decision was to enable us to make several changes for future compilabilty.

Although we wanted to take advantage of object-oriented language constructs to provide language extensibility, "layering", and support for modular programming, we also wanted to preserve the procedural style that many of our users were accustomed to. We also felt that it was essential to provide good interfaces to other languages to provide "maturity" (e.g., mathematical subroutine packages), to allow for efficient low-level execution (our first implementation is an interpreter), and to allow us to integrate large and diverse subsystems.

3.0 Language Description

This section describes the basic entities and operations available to the AML/X programmer. We begin with some definitions:

objects These are the entities that can be directly manipulated by AML/X.

values The value of an object is the interpretation of its contents; it is meaningful to distinguish between an object and its value, since the value of an object can be changed.

types Each AML/X object has a type, which defines the set of values that that object can have. AML/X includes several numeric types (INT, REAL, etc.), types for character- and bit- strings, and more complicated types.

variables These are symbolic names (e.g., Foo, VAR_NAME, x) by which a programmer can refer to objects in AML/X; the object to which a variable refers is called its *binding*. We also speak of the "type" and "value" of a variable, by which we mean the type and value of its binding.

As in most other programming languages, variables are a key feature of AML/X. They can be manipulated in several ways:

evaluation Retrieving the binding of a variable, that is, the object to which it refers symbolically.

assignment Changing the value of the object bound to a variable. Since the type of an object is fixed, assignment cannot change the type of a variable.

binding Changing the binding of a variable, that is, causing it to be bound to a new object. Unlike assignment, there is no restriction about the type of the new object to which the variable is bound.

Expressions are combinations of variables, constants, and operators that are *evaluated* to produce new objects and, in the case of assignment, to change the values of existing objects. AML/X is a so-called *expression-oriented* language, in that all program execution can be described as expression evaluation.

3.1 Data Objects and Operators

AML/X has the usual complement of basic data objects and operators necessary for "language-hood". These, and a few other objects and operators are described cursorily in this section. All AML/X code is written in THIS TYPE FONT so that it stands out from the rest of the text without resorting to the use of various awkward quote marks. In code examples, we have used upper case for usage that is required (e.g., reserved words) and lower case elsewhere (e.g., identifier names).

Numeric Objects and Operators: AML/X has four kinds of numeric objects: INT and LONG_INT, which correspond to 16- and 32-bit integers, and REAL and LONG_REAL, which correspond to single- and double-precision floating point numbers respectively. The usual binary arithmetic operations addition (+), subtraction (-), multiplication (*), division (DIV), and exponentiation (**) are provided.

String Objects and Operators: AML/X provides both BIT and CHAR strings, consisting of zero or more bits or bytes, which can be manipulated collectively or individually. The usual operations for manipulating both CHAR and BIT strings, including current and maximum length, lexicographic comparison, concatenation, and selection, are provided. The standard bitwise logical operations AND, OR, XOR, logical and arithmetic shifts, rotate, and (unary) NOT are provided for BIT strings.

Boolean Objects and Operators: BOOLEAN objects can have two values, which are denoted by the reserved words TRUE and FALSE. The standard Boolean operations AND, OR, XOR, and (unary) NOT are available in AML/X. There are also two "short-circuit" Boolean operators, CAND and COR, which do not evaluate their right operands unless it is necessary. Thus, if x is

#	Concatenate two aggregates to form a new aggregate
OF	Replicate an aggregate a specified number of times
IOTA	Make an aggregate containing a sequence of integers
ISAGG	Is an object an aggregate?
AGGSIZE	Number of elements in an aggregate?
MAP	Distribute an operator or subroutine over arguments
REDUCE	"Place" an operator between successive elements of an aggregate and evaluate
SCAN	Like REDUCE but return an aggregate of partial results
ANY	Are any elements of an aggregate TRUE?
ALL	Are all elements of an aggregate TRUE?
EQUAL	Are the arguments equal in both structure and value?
COMPRESS	Select elements of an aggregate using an aggregate of BOOLEANs as a mask
AGGLOC	Locate a target aggregate within another aggregate

Figure 1. Aggregate built-in operators and subroutines.

zero, `x NE 0 CAND y / x GT 10` evaluates to FALSE and does not divide by zero.

Symbol Objects: SYMBOL objects are variables that are not evaluated (like quoted atoms in LISP) and can therefore be used as names. The notation $name defines a SYMBOL object.

Reference Objects: An object of type REF "points at" an AML/X object, which is called the *referand* of the REF. REF objects can be created by either using the & operator or by calling REF. The referand of a REF object is obtained by using the dereferencing operator (!).

DEFAULT: The AML/X object DEFAULT is used in situations where an object is needed but no particular one is required. For example, the return value of a function called only for its side effects is DEFAULT.

Type Objects and Operators: All AML/X objects have a type, which can be determined using the typeof operator (?). For example, the result of the expression ?3 is the type INT.

3.2 Aggregate Data Objects

Aggregates are one of the most important features of AML/X since they are its most basic data grouping mechanism. An *aggregate* is exactly what its name implies: a collection of objects that can be treated as one. An AML/X object that is not an aggregate, such as an INT or CHAR, is called a *scalar* object.

Creating Aggregates: Aggregates are created using the *aggregation operator* pair, < >, to form a single AML/X object from an explicit list of other AML/X objects. The general form

```
< e1, ..., ek, ..., en >
```

creates an n-element aggregate containing the elements ek. The important points to note are that: (1) any AML/X object can be an aggregate element; (2) elements of an aggregate need not all be the same type; and (3) each element is the object that is the result of evaluating an (arbitrary) expression that specifies the element. Once an aggregate is created, its size cannot be changed. Figure 1 lists some of the operators and built-in subroutines for working with aggregates.

Aggregate Subscripts: Individual aggregate elements can be referred to by numeric indices or *subscripts*. If a is an aggregate, the expression a(s) is a *subscripted aggregate reference*; we also say that a is being *applied* to s. Elements of multi-dimensional aggregates (where some elements are themselves aggregates) can be referenced by multiple subscripts. For example, a(3)(2) refers to the second element of the third element of a. Since this syntax is awkward, multiple subscripts can be elided, so that a(3)(2) can be written as a(3, 2).

The most general case of subscripting occurs with multi-dimensional aggregates subscripted by arbitrary combinations of scalar and aggregate subscripts. The rules are simple. Suppose

a	is an arbitrary aggregate
i	is a positive integer
s	is an aggregate of positive integers or aggregates
..rest..	denotes the "rest" of a list of subscripts (possibly empty).

Then, the meaning of any subscript can be determined by applying the following rules recursively until all subscripts are reduced to scalar values:

Expression	*Equivalent*
a(i,..rest..)	(a(i)) (..rest..)
a(s,..rest..)	<...,a(s(k),..rest..),...>
a(DEFAULT,..rest..)	<...,a(k,..rest..),...>

A few examples will help to give an intuitive understanding of what these rules mean. We'll start with the 3×4 matrix m defined by

```
m: NEW < < 1,  2,  3,  4 >,
         < 5,  6,  7,  8 >,
         < 9, 10, 11, 12 > >;
```

The NEW keyword indicates that this is a variable declaration.

It is simple to use aggregate subscripts to refer to rows and columns of m. If i is a scalar, then m(i) refers to the i-th row of m and m(DEFAULT, i) refers to the i-th column of m. The latter can be written more succinctly as m(, i). In general, any "missing" subscript is treated as if DEFAULT had been specified. The "missing" subscript notation is convenient for denoting entire rows or columns. Selected rows or columns can be referred to by using aggregate subscripts, as illustrated below:

```
m(<1, 3>)          ## 1st and third rows
m(, IOTA(2,4))     ## 2nd through 4th cols
m(, <4, 1>)        ## 4th and 1st columns
m(<1,2>, <1,2>)    ## Upper left 2x2
```

- IF cond THEN e1 [ELSE e2]
  evaluates e1 if cond is TRUE and to e2 otherwise; if e2 is ommitted, DEFAULT is used
- WHILE cond DO expr
  continues to evaluate expr while cond is TRUE; the result is the result of the last evaluation of expr, or DEFAULT if none
- WHILE cond DO COLLECT expr
  continues to evaluate expr while cond is TRUE; the result is an aggregate of the successive results of evaluating expr, or the empty aggregate if none
- REPEAT expr UNTIL cont
  continues to evaluate expr until cond becomes TRUE; the result is the result of the last evaluation of expr, or DEFAULT if none
- REPEAT COLLECT expr UNTIL cont
  continues to evaluate expr until cond becomes TRUE; the result is an aggregate of the successive results of evaluating expr, or the empty aggregate if none
- BEGIN e1; ... ek; END
  the expressions ei are successively evaluated; the result is the result of evaluating ek.
- SELECT expr CASE c1 THEN e1 ... CASE ck THEN ek [OTHERWISE oexpr] END
  The result of evaluating expr is compared against successive ei; if there is a match, the result is ci; if no match the result is oexpr if present, and DEFAULT otherwise

Figure 2. Control Operators

Operator Mapping: Operators extend to aggregate operands through a set of *mapping rules* (adopted with slight alteration from AML[2]) which are an abstract form of the distributive law of arithmetic. Suppose

s	is any scalar object
<u1,...,un>	is an n-element aggregate
<v1,...,vn>	is an n-element aggregate
op	is an AML/X operator

Then, the operator mapping rules are:

Expression	*Equivalent*
op <u1,...,un>	<op u1,...,op un>
s op <v1,...,vn>	<s op v1,...,s op vn>
<u1,...,un> op s	<u1 op s,...,un op s>
<u1,...,un> op <v1,...,vn>	<u1 op v1,...,un op vn>

These mapping rules apply recursively to multi-dimensional aggregates.

A key point about the mapping rules is that they work in a uniform way for most AML/X operators. In particular, "parallel assignment" can be written in the form <v1,...,vk> = expression.

3.3 Control Operators

AML/X supports control flow constructs typical of a structured programming language. These are summarized in Figure 2. Since AML/X is an expression-oriented language, all control flow constructs are operators that yield a result like any other operator, as illustrated by the expression

```
move(WHILE ask('More?') DO COLLECT teach())
```

which creates an aggregate of (presumably) points returned by successive calls to teach and passes it to the subroutine move.

3.4 Expression Evaluation

AML/X is an *expression-oriented* language, which means that *every* AML/X program construct is an expression. Thus, expression evaluation is the fundamental computational process in AML/X.

Before an expression is evaluated, it is parsed into a tree of subexpressions based on the usual sort of precedence rules found in most languages. Then it is evaluated by applying operators to objects according to two evaluation rules, one for ordinary operators and one for *special form* operators. To evaluate an expression,

1. If the operator is not a special form, first evaluate the operator's subexpressions (left-to-right), then apply the operator to the resulting objects;
2. If the operator is a special form, the operator is applied without prior evaluation of its subexpressions; the operator may cause any or all of its subexpressions to be evaluated.

The rules are applied recursively and must be augmented by two additional rules which are the basis of the recursion:

1. A constant (e.g., 1, 2.3D5, 'FOO') evaluates to itself.
2. A simple variable (e.g., a, arm, current_speed) evaluates to its binding (not a copy).

The details of the evaluation rule are only relevant when evaluation of a subexpression causes a *side effect*, i.e., when the value of some variable is changed during the course of evaluating the subexpression. We will consider several examples.

Example: <a, b> = <1, 2>
This expression assigns 1 to a and 2 to b. The left-hand-side evaluates to an aggregate of the bindings of a and b (not copies of the bindings); the right-hand-side evaluates to the aggregate <1,2>. The usual mapping rules then cause the assignment operator to be "distributed" over the corresponding aggregate elements.

There are some circumstances where such behavior is not desired. For those cases, AML/X has a *copy operator*, denoted by %. An expression of the form %expr evaluates to a *copy* of

```
diagonal:  SUBR ( m );
   ## Returns the diagonal elements of the n x n matrix m
   n:  NEW AGGSIZE ( m );   ## Number of rows (cols) in m
   i:  NEW 0;               ## Index over rows/cols
   RETURN ( WHILE ++i LE n DO COLLECT m(i, i) );
END;

id: NEW 3 OF 3 OF 0.;    ## Make a 3 by 3 matrix of zeros
diagonal(id) = 1.;       ## Set diagonal elements to 1
```

Figure 3. The subroutine diagonal returns the diagonal elements of a square matrix represented as a nested aggregate. Its use is illustrated by constructing the 3 by 3 identity matrix id.

the result of evaluating expr. Its use is illustrated in the following example.

Example: <a, b> = <b, a>
This expression might be written (incorrectly) to swap two variable values. Each side of the assignment evaluates to an aggregate of the bindings of a and b, but in different order. Applying the mapping rule, the expression is equivalent to <a = b, b = a>. Since these are evaluated left-to-right, the value of a will be "lost". The copy operator, used in the expression <a, b> = %<b, a>, causes a copy of the original values to be made before any of the assignments are done, thus achieving the desired effect.

Example: diagonal(m) = 1
The subroutine shown in Figure 3 uses WHILE..DO..COLLECT to construct an aggregate of the bindings of the diagonal elements of the matrix m. Since diagonal(a) evaluates to an aggregate of the bindings of the diagonal elements, diagonal(a) = 1 has the effect of setting all diagonal elements of a to 1. Again, the key point is that variables in AML/X evaluate to their bindings, *not* copies of those bindings, and the WHILE..DO..COLLECT operator aggregates but does not copy the successive values of the loop expression.

3.5 Subroutines

This section describes AML/X subroutines and variable declarations. A subroutine is defined by a statement of the form

```
subrname: SUBR(...formal_arguments...)
   declarations
   ...
   statements
END;
```

This statement really consists of two parts, the subroutine expression itself (SUBR...END) and the variable name subrname. The statement defines a subroutine and makes it the binding of the variable subrname. There may be any number of formal arguments, including zero. The *body* of the subroutine consists of any number (including zero) of *local variable declarations* followed by any number (including zero) AML/X statements.

A subroutine is called when a subroutine object is *applied* to an argument list, as follows:

```
subrexpr(actual_arg_1,...,actual_arg_n)
```

The subroutine to which the expression subrexpr evaluates is called with the specified actual arguments.

Local Variables and Declarations: Local variables are variables whose names are known only within the extent of a particular subroutine. A local variable is defined by being *declared* at the beginning of a subroutine. Variable declarations (except labels and internal subroutines) are of the form

```
varname: declarator init_expr;
```

where varname is the name of the variable being declared, declarator specifies various properties of the local variable, and init_expr is an arbitrary AML/X expression that defines the variable's type and initial value. The effect of a variable declaration is to associate a variable name with the storage that holds its binding, as defined by four attributes:

> **Memory space:** Determines which memory space the binding is stored in. If the binding is in the stack, it will be discarded when the subroutine terminates.
> **Constant:** Determines whether or not the value of the binding can be altered once it is initialized.
> **Copy:** Determines whether the result or a copy of the result of evaluating the initialization expression becomes the binding.
> **Persistent:** The binding is created when the subroutine is loaded and is reestablished each time the subroutine is invoked.

Possible declarators and their attributes are shown in F';ure 4.

A subroutine definition contained in another subroutine is an *internal subroutine* and can only be called from within the containing subroutine. Free variables in internal subroutines are bound lexically. Labels are declared by prefixing any statement by a name, as in

```
labname: stmt;
```

Internal subroutines and labels are two exceptions of the rule that declarations appear at the beginning of a subroutine. In this case, ease-of-use seemed to outweigh consistency.

Arguments: A simple formal argument is a variable name, implicitly declared as a BIND declaration, and bound to the corresponding actual argument when the subroutine is called. In effect, arguments are passed by reference. The caller can specify that an argument be passed by value by preceding the actual argument with the copy operator (%). The formal argument can also be preceded by the copy operator, in which case the argument is passed by value regardless of how the actual argument is passed. This is illustrated in the following example:

Declarator	Memory Space	Constant	Copy	Persistent
NEW	Stack	No	Yes	No
NEW CONSTANT	Stack	Yes	Yes	No
CONSTANT	Stack	Yes	Yes	No
STATIC	Heap	No	Yes	Yes
STATIC CONSTANT	Heap	Yes	Yes	Yes
BIND	—	—	No	No
STATIC BIND	—	—	No	Yes

Figure 4. Variable declarators

```
fact: SUBR(%i)
  ## Returns i factorial (i GE 1)
  f: NEW i;
  WHILE --i GT 1 DO f *= i;
  RETURN(f);
END;
```

AML/X provides a way to specify values for missing arguments. If the formal argument has the form

```
formal_arg DEFAULT expr
```

and the corresponding actual argument is supplied, then formal_arg is processed as described above. However, if the corresponding actual argument is missing, or if its value is DEFAULT, then expr is evaluated and its result becomes the binding of formal_arg. Thus in the code

```
s: SUBR(p, tol DEFAULT 1.0e-6)
...
END;
s(3);
s(3, 1.0e-8);
```

tol is bound to 1.0e-6 the first time s is called and to 1.0e-8 the second time.

Subroutines can also access excess actual arguments using the predefined variable ACTUAL_ARGS, which is bound to an aggregate of the actual arguments.

It is easy to write *generic subroutines* in AML/X because it is not necessary to declare the type of formal arguments. This is often convenient, especially for small programs, but can lead to programming errors and make it very difficult to compile efficient code for the subroutine. In keeping with our philosophy of letting the user make the trade-off between flexibility and efficiency, AML/X allows optional type declarations for formal arguments. A formal argument (possibly including a DEFAULT clause) can be followed by a type specification of the form

```
MUSTBE type_spec
```

where type_spec is an aggregate of types. If the corresponding actual argument is not one of the specified types, an exception is raised.

Exiting from Subroutines: Any AML/X object can be returned as the result of a subroutine call by passing the object to the RETURN built-in subroutine. The object returned is not copied unless it would be destroyed by termination of the subroutine (e.g., a NEW variable). Therefore a variable binding can be returned and a subroutine call can be used on the left-hand side of an assignment, as illustrated in Figure 3.

The built-in subroutine CLEANUP can be called while executing a subroutine to request an action when the subroutine terminates. For example, suppose a subroutine which opens a file should always close it, even if the subroutine terminates because of some error condition. This can be done by the code shown in Figure 5.

3.6 Exception Handling

When an error is detected during program execution an *exception* is *raised.* The action taken by a program when an exception is raised is determined by the *exception handler* defined for that particular exception. The design of AML/X's exception handling is described in [18].

Each possible exception has a name, which is represented by a SYMBOL. When an exception is raised, either by the system or by the user through the RAISE_EXCEPTION built-in subroutine, AML/X finds the most recent activation of a block containing a variable of that name which was declared as a HANDLER. If none is found and a variable of the appropriate name exists at top-level, it is used. The binding of that variable is the exception handler used.

The type of the binding determines what kind of action is taken, as follows:

EXPR: The EXPR is evaluated and the result becomes the result of the exception handler.

SUBROUTINE: The SUBROUTINE is called. The arguments passed provide the SUBROUTINE with detailed information about the exception that occurred. The result of the subroutine call becomes the result of the exception handler.

LABEL: The LABEL is branched to. The exception handler has no result.

BOOLEAN: The BOOLEAN object is set to TRUE and the result of the exception handler is TRUE.

SYMBOL: The value of the SYMBOL is used as the name of another exception to raise. This allows exceptions to be grouped hierarchically into exception groups, each consisting of several exceptions all handled by the same exception handler.

```
file_update: SUBR()
   c: NEW OPEN('file.name','w'); ## Open file
   CLEANUP( $(CLOSE(c)) );       ## Request cleanup action
   ...                           ## Processing code
END;
```

Figure 5. An example of a subroutine cleanup action: The expression $(CLOSE(c)) passed to CLEANUP is an (unevaluated) expression object which will be evaluated when file_update terminates.

A program can use the EXCP_BINDING built-in subroutine to determine the current exception handler for a specified exception. In this way, a subroutine can decide to let an exception be handled by its caller if its caller has an appropriate handler, or can handle the exception itself if the caller doesn't provide a handler. For example, in the code in Figure 6, the subroutine default_handle_set is used in foo to bind EXCP_ZERODIV to the caller's handler if it exists, or to a boolean flag if it doesn't exist.

Most system-defined exceptions are *continuable*, meaning that execution continues from where the exception was raised and the result of the exception handler becomes the result of the operator that caused the exception. The operation is not "retried", although the exception handler is free to retry the operation or provide a reasonable result, as in

```
EXCP_ZERODIV: HANDLER SUBR()
   ## Return largest possible number
   RETURN(MAXVAL(LONG_REAL));
END;
```

User exceptions can be either continuable or non-continuable.

3.7 Object-oriented Programming

The use of abstraction is a very powerful tool for building large programs. Powerful or complicated abstractions can be implemented by using simpler ones so that each implementation is small and (presumably) easy to understand. AML/X supports abstraction by providing *classes*, a mechanism for defining new objects and operations on them. A class defines a new type in the language; a *class instance* is a particular object derived from a class. This is analogous to built-in types and instances of the built-in types: for example the number 2 is an instance of the type INT. Thus, if one writes a class definition for complex numbers, each instance of that class would correspond to a particular complex number. The data for each instance is held in its *instance variables*, which are accessible only from within the class unless access elsewhere is granted explicitly.

Class Definitions: A class definition is defined by a statement of the form

```
classname: CLASS(...formal_arguments...)
   IVARS
      instance variable declarations
   END;
   declarations
   initialization statements
   methods
END;
```

This statement defines a TYPE and makes it the binding of classname. Each declaration in the IVARS section defines an instance variable. All of the non-STATIC declarators shown in Figure 4 can be used. Figure 7 shows part of a simple class definition for vectors.

Class Instantiation: A class instance is created (the "class is instantiated") by calling the class definition as one would a subroutine, the only difference being that a class returns an instance containing the current bindings of the class' instance variables. Thus, vector(1,2,3) would return an instance of vector with instance variables 1E0, 2E0, and 3E0.

Methods and Operator Overloading: Classes are only useful if there is a way to do something to class instances. A *method* is a special kind of internal subroutine that (1) is contained in a class definition but can be called from outside of the class definition, and (2) has access to the instance variables of an instance of the class. A method is invoked by executing an expression of the form

```
obj_exp.method_name(...formal_argumens...)
```

where obj_exp is an expression that evaluates to a class instance and method_name is the name of a method. The method executes exactly like an ordinary subroutine except that the instance variables are bound to the values of the instance variables contained in the instance instead of to the result of evaluating their initialization expressions. Also, the predefined variable SELF is bound to the class instance itself.

AML/X operators can be extended to class instances, or *overloaded*, on a class-by-class basis by associating a method with the operator. This is done simply by having in the class a method whose "name" is a literal form of the operator to be overloaded. If the left operand of an operator is a class instance, the corresponding operator method in the appropriate class definition is invoked with the right operand as actual argument; if the left operand is an instance of a built-in type but the right operand is a class instance, the corresponding *modifier method* is invoked passing the left operand as actual argument. This allows non-commutative operators to be overloaded.

Exposed Instance Variables: Ordinarily, instance variables are not accessible except within the class definition or through method calls. Direct access to instance variables can be explicitly granted by declaring them to be EXPOSED as in

```
IVARS
   x: EXPOSED NEW REAL();
END;
```

An exposed instance variable is referenced by an expression of the form:

```
instance.inst_var_name
```

Note that a class definition containing only exposed instance variables is equivalent to C's structures and PASCAL's records.

Exposed instance variables were added to the language in response to user's complaints that they often had to write a method just to access a single instance variable. However, because they expose the data representation used by a class, they

```
default_handle_set: SUBR(ex_name, new_handler)
  ## If the caller of the caller of this routine does not
  ## have an exception handler for the exception named
  ## ex_name, return the specified new handler; otherwise,
  ## return the existing handler.

  existing_handler: BIND EXCP_BINDING(ex_name, CALLER(CALLER()));

  RETURN( IF existing_handler NE UNBOUND THEN existing_handler
                                         ELSE new_handler      );
END;

foo: SUBR()
    ## Set up handler for EXCP_ZERODIV
    EXCP_ZERODIV: HANDLER BIND
      default_handle_set($EXCP_ZERODIV, FALSE);
    ...
END;
```

Figure 6. Providing a default exception handler: The subroutine foo binds the result of calling default_handle_set to EXCP_ZERODIV, thus making it the exception handler for dividing by zero. The subroutine default_handle_set first determines the exception handler binding in its caller's caller (i.e., foo's caller). If there is one, it is returned for use; otherwise the new_handler is returned for use as the exception handler.

violate the abstraction that classes were intended to provide. PRIVATE EXPOSED instance variables, which only allow direct access to instance variables from within the class definition, were introduced so that data representation could be exposed inside the class definition but remain hidden from outside view.

4.0 Examples

4.1 Cartesian Data Types

Data types for vectors, rotations, and coordinate transformations are often provided in special-purpose languages for robotics and CAD [e.g., 14, 15, 19, 20]. The conciseness and consistency checking provided by such types, compared to the subroutine libraries providing comparable functions for general-purpose languages, significantly improves programmer productivity and program readability. Unfortunately, users of special-purpose languages are often stuck with whatever internal representation and function set the system implementers have chosen to provide.

This section illustrates the use of AML/X classes and operator overloading to implement these data types in a way that provides both expressive power and easy customizing.

Vectors: Vectors are represented by three real numbers, stored in EXPOSED instance variables x, y, z. Methods overloading the normal arithmetic operators can be provided for vector addition, subtraction, and scaling. "Multiplication" of two vectors is used for vector inner product and "exponentiation" is used for cross product. A method for assignment can also be provided. A sketch of such a class definition is given and several standard constant vectors are declared in Figure 7.

Rotations: Rotations are commonly represented as 3x3 orthogonal matrices. This representation is easily understood and is computationally efficient if many vectors are to be rotated. On the other hand, it is wasteful of storage, subject to numerical inconsistencies, and computationally expensive for many operations, including composition and specification from angles.

As an alternative, we have sketched in Figure 8 a class definition for rotations that uses quaternions[21, 22] as the underlying representation. In this case, EXPOSED PRIVATE instance variables are used in order to hide implementation details while permitting efficient execution within methods. Two "class" methods,

```
r = rotation.polar(axis_vector, angle);
r = rotation.euler(abt_z_1,abt_y,abt_z_2);
```

permit rotations to be specified either as a right-handed twist about a specified axis or as a sequence of rotations about cardinal axes. Multiplication is overloaded to provide for composition of two rotations and rotation of a vector and division is overloaded to provide for formation of an inverse rotation and for multiplication by the inverse.

Two methods,

```
<axis_vector,angle> = r.polar_parms();
<abt_z_1,abt_y,abt_z_2> = r.euler_parms();
```

invert the polar and euler methods, respectively. One potential problem with the latter method arises when the second angle, corresponding to rotation about the "y" axis, is zero. In this case, only the sum of the first and third angles is determined. By default, the third value will be set to zero and an exception, EXCP_degen_rot, is raised. However, the exception handler can override the default. For example, the simplified kinematic solution procedure shown in Figure 9 uses an exception handler to divide the angle sum evenly between the first and third wrist joints.

Transformations: Arbitrary coordinate transformations, consisting of rotation followed by translation, are straightforwardly implemented using the class definitions for vectors and rotations. In a typical class definition (not shown), multiplication would be overloaded to provide transformation of vectors and composition, and division would be overloaded to provide inverses and composition with inverses. A class method for co-

```
vector: CLASS(xx DEFAULT 0.0, yy default 0.0, zz DEFAULT 0.0)

  IVARS
    x: EXPOSED NEW REAL(xx);
    y: EXPOSED NEW REAL(yy);
    z: EXPOSED NEW REAL(zz);
  END;

  $*: METHOD(v)      ## Inner product and scaling
    SELECT (?v)
      CASE vector THEN RETURN(x*v.x+y*v.y+z*v.z)
      OTHERWISE RETURN(vector(x*v, y*v, z*v))
    END;
  END;
  $*: MOD_METH(s) RETURN(vector(s*x, s*y, s*z)); END;

  $**: METHOD(v MUSTBE <vector>)  ## Cross product
    RETURN(vector(y*v.z-z*v.y, z*v.x-x*v.z, x*v.y-y*v.x));
  END;

  ...

  $=: METHOD(v)
    <x, y, z> = SELECT (?v)
                  CASE vector THEN <v.x, v.y, v.z>
                  OTHERWISE v
                  END;
    RETURN(SELF);
  END;

  uvect: METHOD() RETURN(SELF/sqrt(self*self)); END;
END;

null_vector: STATIC CONSTANT vector(0, 0, 0);
x_axis:      STATIC CONSTANT vector(1, 0, 0);
y_axis:      STATIC CONSTANT vector(0, 1, 0);
z_axis:      STATIC CONSTANT vector(0, 0, 1);
```

Figure 7. Class definition for vectors: Methods that overload the addition, subtraction, and division operators have been omitted for brevity.

ercing vectors and rotations to transformations would also be useful.

4.2 Coordinate Frames and Affixment

Coordinate transformations arise naturally from part-subpart relationships in both robot and CAD programming. If the location (i.e., position and orientation) of Part A relative to the workstation is given by a transformation, frame_a, and trans_ab gives the location of a Subpart B relative to A, then the location of B relative to the workstation is given by frame_a*trans_ab. Similarly, if the location of C relative to B is given by trans_bc, then the location of C relative to the workstation is given by frame_a*trans_ab*trans_bc. In practice, these expressions become very cumbersome and tend to interfere with the readability of programs. To get around this, AL [14] introduced the concept of *affixment*, in which part-subpart relationships and similar dependencies were declared explicitly, as in

```
AFFIX part_b TO part_a AT trans_ab;
```

Programs then simply referred to part_b to get the current location of Part B. If Part A was moved or if a new value for its location was determined by sensing, then the location value for Part B was updated automatically.

One of the interesting aspects of the AL implementation of affixment [23] was that recomputation of coordinate frame values was deferred until they were needed, but the values were saved to eliminate needless recomputation. This saving can be quite important in robotic applications where parts are being moved about the workstation and where the expressions involved in recomputation may involve a long chain of affixments. An AML/X implementation of much the same idea is shown in Figure 10. Once again, classes and operator overloading are used to provide a new data type, frame, whose value corresponds to a coordinate system.

Figure 11 illustrates the use of this data type in a simple assembly application. Figure 11(a) shows a box and cover plate being delivered to an assembly station on a small tray. The coordinates of the box and cover are initially known relative to the tray. Furthermore, grasping points relative to the box and cover have been defined. The problem is to use vision to locate the tray, then use a vision routine to locate the box and cover more precisely, based on the tray location. Finally, place the cover on the box and pick up the box. A program to accomplish this is sketched in Figure 11(b).

```
rotation: CLASS(ss DEFAULT 0.0, vv DEFAULT null_vector MUSTBE <vector>)

  IVARS
    s: PRIVATE EXPOSED NEW REAL(ss);
    v: PRIVATE EXPOSED NEW vv;
  END;

  $=: METHOD(r) <s,v> = <r.s, r.v>; END;

  polar: CLASS_METH(axis MUSTBE <vector>, angle)
    RETURN(rotation(cos(angle/2), sin(angle/2)*axis.uvect()));
  END;

  euler: CLASS_METH(a, b, c)
    RETURN(rotation.polar(z_axis,a) * rotation.polar(y_axis,b) * rotation.polar(z_axis,c));
  END;

  $*: METHOD(p MUSTBE <rotation,vector>)
    SELECT (?p)
      CASE rotation THEN RETURN(rotation(s*p.s-v*p.v, p.s*v+s*p.v+v**p.v))
      CASE vector THEN RETURN((SELF*rotation(0,p)/SELF).v)
    END;
  END;

  $/: METHOD(p MUSTBE <rotation>)
    RETURN(rotation(s*p.s+v*p.v, p.s*v-s*p.v-v**p.v))
  END;

  $/: MOD_METH(p)
    IF p NE 1 THEN RAISE_EXCP($EXCP_invalid_inv,,<p,SELF>,TRUE);
    RETURN(rotation(s,-v));
  END;

  euler_parms: METHOD()
    ad:   BIND s**2 + v.z**2;
    bc:   BIND v.x**2 + v.y**2;
    beta: BIND acos((ad-bc)/(ad+bc));
    gpa:  BIND atan2(v.z, s);
    gma:  BIND IF beta NE 0 THEN atan2(v.x, v.y)
           ELSE BEGIN rslt: BIND RAISE_EXCP($EXCP_degen_rot,, gpa, TRUE);
                  IF is_3_reals(rslt) THEN RETURN(rslt) ELSE 0.0;
                END;  ## See also Figure 9
    RETURN(<gpa-gma, beta, gpa+gma>);
  END;

  polar_parms: METHOD()
    s_sqd: BIND v**2;
    RETURN(IF s_sqd EQ 0.0 THEN <z_axis, 0.0>
                            ELSE <v/sqrt(s_sqd), acos(s**2-s_sqd)>);
  END;

  is_3_reals: SUBR(val)
    RETURN( ISAGG(val) CAND AGGSIZE(VAL) EQ 3 CAND ALL(?VAL EQ REAL) );
  END;

END;

EXCP_degen_rot: BIND HANDLER FALSE;  ## Default: ignore exception
null_rot: STATIC CONSTANT rotation.polar(z_axis, 0);
```

Figure 8. Class definition for rotations

The declarations create an "affixment tree" of frames. The instance variables associated with each frame specify the parent frame, the offset of the frame relative to its parent, the present value of the frame—i.e., its transformation relative to the workstation—a "mark" counter used to determine whether the value is valid, and a flag specifying whether the affixment is "rigid". The mark counter is incremented every time a new value is saved, and a frame's value is valid if and only if its mark counter is greater than its parent's. "Rigid" affixments are those

```
hand_vector: STATIC CONSTANT vector(0,0,9);

solve_arm: SUBR( f MUSTBE <frame> )
  t: BIND f.xf();  ## Frame transformation
  cj: BIND t.v - t.r * hand_vector;
  RETURN( <cj.x, cj.y, cj.z> # (t.r).euler_parms() );

  EXCP_degen_rot: HANDLER SUBR(a, b, ang_sum);
      RETURN(<ang_sum, 0., ang_sum>/2);
  END;
END;
```

Figure 9. Simplified kinematic solution procedure for a cartesian robot such as the IBM 7535: This subroutine returns an aggregate of six real numbers giving joint values corresponding to specified hand frame. Note the use of an exception HANDLER subroutine to override the default handling of degenerate wrist rotations.

in which updates to the frame value are to cause the parent's value to be updated as well.

The initial assignment statement causes the value of tray to be updated, and its mark to be incremented. The second statement first causes the value of cover to be computed as part of the call to locate_object. Since tray has been updated, its mark counter is higher than that of cover, so the value is obtained by obtaining a valid value for tray and then composing the result with the offset stored for cover. The assignment then updates the value stored in cover a second time. The third statement repeats the process for box.

Subsequent statements call subroutines to pick up the cover, place it on the box, etc. The fragments from grasp_object and move_object illustrate the use of affixment to simplify programming. grasp_object moves the robot to the specified grasping point, closes the gripper, and then affixes the object (by assumption, the most remote rigidly affixed ancestor) to the robot. Subsequent motions of the robot will cause all location attributes of the object to be updated. move_object verifies that motion_frame is affixed to the robot and then moves the robot so that the value of motion_frame is equal to destination.

5.0 Implementation

AML/X is implemented by a portable interpreter written in C. It runs on the IBM 370 family of machines under VM/CMS, on the IBM PC under DOS and XENIX, on the IBM RT/PC under AIX, and on several other machines. Facilities exist on all machines for writing C subroutines callable from AML/X; on the VM/CMS implementation, there are also interfaces to Fortran and PL/1.

6.0 Experience and Conclusions

AML/X has now been in use for about a year in our research in robotics, computer-aided design, and machine vision. Execution speed of the interpreter has proved adequate for robotics applications. As anticipated, however, the interpreter is too slow for production use in the lowest layers of more complex systems, especially in CAD applications. We have begun work on a prototype compiler that should resolve this issue. Meanwhile, several of our researchers find AML/X sufficiently expressive that they prototype low level geometric data structures and algorithms in AML/X and recode in C where necessary for efficiency. Classes, including operator overloading, are often used in this work, and the resulting programs are both readable and modifiable.

AML/X has also been used as a programming "front-end" to a powerful geometric modelling system [17]. In this case, class definitions for geometric objects have been written in AML/X but the actual data representation is created and manipulated by the modelling system. Each class instance essentially contains a "handle" on the data maintained by the modelling system. Methods implement geometric operations by passing these "handles" to the modelling system, which then does the necessary computation. Our initial (limited) experience with this use of AML/X in the higher layers of a system is that it provides a very powerful programming environment with very reasonable performance. The drawing in Figure 11 was produced by an AML/X program running on this system.

The need for concurrency can to some extent be met by simple interfaces to operating system services. However, in response to the requirements of automation programming, we have begun to consider providing concurrency directly in the language.

Providing interfaces to other languages has allowed us quick access to a variety of existing code, ranging from mathematical subroutines, to graphics routines, to a large modelling system. We expect that AML/X will continue to be used at the highest layers of large systems built from existing components and that using it in this way will help us to integrate various automation technologies.

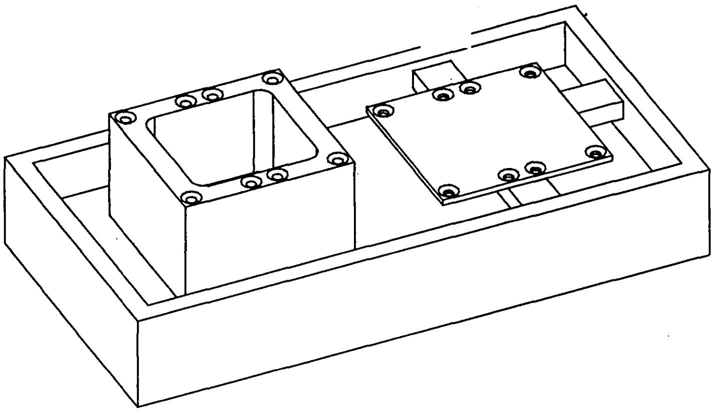
Acknowledgments

AML/X is a descendant of AML (A Manufacturing Language) and owes much to its developers. As early users of AML/X, Georg Maier and Vijay Srinivasan at IBM Yorktown provided especially valuable suggestions on ways to improve the language. Dave Klein of the Manufacturing Systems Program (MSP) at IBM Boca Raton designed and implemented the garbage collector. Ken Morgan, also of MSP, implemented most of the debugging facilities. Many others in the Automation Research group at IBM Yorktown and the Manufacturing Systems Program at IBM Boca Raton have also made useful comments and suggestions.

```
frame: CLASS(afx  DEFAULT NULL MUSTBE <REF,frame>,
             ofst DEFAULT trans(),
             rigidly DEFAULT FALSE MUSTBE <BOOLEAN> KEY)

  frame_counter: STATIC LONG_INT(0);

  IVARS
    parent: PRIVATE EXPOSED NEW IF ?afx EQ REF THEN afx ELSE &afx;
    offset: PRIVATE EXPOSED NEW trans.coerce(ofst);
    value:  PRIVATE EXPOSED NEW IF parent EQ NULL THEN
                                trans() ELSE (!parent).xf()*offset;
    mark:   PRIVATE EXPOSED NEW frame_counter++;
    rigid:  PRIVATE EXPOSED NEW BOOLEAN(rigidly);
  END;

  $=: METHOD(f)
    value = trans.coerce(f);
    IF parent NE NULL THEN
       IF rigid THEN (!parent) = value/offset
       ELSE offset = (1/(!parent).xf())*value;
    mark = frame_counter++;
    RETURN(%value);
  END;

  $*: METHOD(b) RETURN(SELF.xf() * b); END;

  $/: METHOD(b) RETURN(SELF.xf() / b); END;

  xf: METHOD() SELF.validate(); RETURN(%value); END;

  unfix: METHOD() parent = NULL; rigid = FALSE; END;

  affix_to: METHOD(afx MUSTBE <REF,frame>, ofst,
                   rigidly DEFAULT FALSE MUSTBE <BOOLEAN> KEY)
    IF ?ofst EQ DEFAULT THEN SELF.validate();
    SELF.unfix();
    rigid = rigidly;
    parent = IF ?afx EQ frame THEN &afx ELSE afx;
    IF ?ofst EQ DEFAULT THEN
      offset = (1/((!parent).xf()))*value
    ELSE
      BEGIN mark = 0; offset = trans.coerce(ofst); END;
  END;

  validate: PRIVATE METHOD()
    if parent EQ NULL then RETURN();
    (!parent).validate();
    IF mark LE (!parent).mark THEN
       BEGIN value = (!parent).value*offset; mark = frame_counter++; END;
  END;

  rigid_ancestor: METHOD()
    RETURN(IF rigid CAND parent NE NULL THEN (!parent).rigid_ancestor() ELSE SELF);
  END;

  has_ancestor: METHOD(f)
    RETURN(IF SELF EQ f THEN TRUE
           ELSE IF parent EQ NULL THEN FALSE
           ELSE (!parent).has_ancestor(f));
  END;

END;
```

Figure 10. Class definition for Cartesian frames and affixments

(a)



(b)

```
tray:        NEW frame();
cover:       NEW frame(tray,  trans(...));
cov_grasp:   NEW frame(cover, trans(...), rigidly=TRUE);
box:         NEW frame(tray,  trans(...));
box_top:     NEW frame(box,   trans(...));
box_grasp:   NEW frame(box,   trans(...), rigidly=TRUE);

tray =  locate_object(DEFAULT, ...); ## no a priori info
cover = locate_object(cover,   ...); ## locate cover better
box =   locate_object(box,     ...); ## locate box better

grasp_object( cover_grasp, ...);     ## grasp the cover
move_object( cover, box_top, ... );  ## move it
release_object( ... );               ## let go
grasp_object( box_grasp, ... );
move_object( ... );

...

grasp_object: SUBR(grasp_frame, ... );
   ...
   move_robot(grasp_frame, ...);
   close_gripper( ...);
   (grasp_frame.rigid_ancestor()).affix_to(robot);
   ...
   END;

move_object: SUBR(motion_frame, destination, ...);
   ...
   IF NOT motion_frame.has_ancestor(robot) THEN
      RAISE_EXCP( ... );
   move_robot(destination/motion_frame*robot, ... );
   ...
   END;
```

Figure 11. Simple robotic assembly task: (a) Initial situation and (b) Sketch of program. The models in (a) were implemented using an AML/X front-end to the IBM Geometric Design Program [17].

References

[1] D.D. Grossman, "Programming a computer controlled manipulator by guiding through the motions," IBM Research Report RC6393, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1977.

[2] R.H. Taylor, P.D. Summers, and J.M. Meyer, "AML: A Manufacturing Language," *Intl. J. Robotics Research*, vol. 1, no. 3, pp. 19-41, Fall 1982.

[3] R. H. Taylor and D. D. Grossman, "An Integrated Robot System Architecture", *IEEE Proceedings*, vol. 71, pp. 842-855, July 1983.

[4] M.A. Lavin and L.I. Lieberman, "AML/V: An Industrial Machine Vision Programming System," *Intl. J. Robotics Research*, vol. 1, no. 3, pp. 42-56, Fall 1982.

[5] J. Korein, G. Maier, R. Taylor and L. Durfee, "A Configurable System for Automation Programming and Control," *Proc. 1986 IEEE Conf. on Robotics and Automation*, San Francisco, pp. 1871-1877, April 1986.

[6] L.R. Nackman, M.A. Lavin, R.H. Taylor, and W.C. Dietrich, Jr., "AML/X User's Manual," IBM Research Report RA 175, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, (1986).

[7] C.A.R. Hoare, "Hints on Programming Language Design," Keynote address given at the ACM SIGACT/SIGPLAN Conf. on Prinicples of Programming Languages, Boston, (1973) as quoted on pp. 255 and 257 of C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, New York: John Wiley, 1982.

[8] M.A. Wesley, "Construction and Use of Geometric Models," in *Computer Aided Design*, J. Encarnacao, ed., Lecture Notes in Computer Science 89, Springer Verlag, 1980.

[9] T. Lozano-Perez, "Robot Programming," *Proc. of the IEEE*, vol. 71, no. 7, pp. 821-841, July 1983.

[10] M.A. Lavin and L.I. Lieberman, "AVL0 -- A Vision Language," IBM Research Report 8390, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, (1980).

[11] C. Eastman and M. Henrion, "Glide: A Language for Design Information Systems," Proc. ACM SIGGRAPH'77, *Computer Graphics*, vol. 11, no. 2, pp. 24-33, Summer 1977.

[12] P. Will and D. Grossman, "An experimental system for computer controlled mechanical assembly", *IEEE Trans. Comput.*, vol. C-24, p. 879., 1975.

[13] R. Evans, et. al., "Software system for a computer controller manipulator", IBM Res., Yorktown Heights, NY, Rep. RC-6210, 1977.

[14] R. Finkel, R. Taylor, R. Bolles, R. Paul and J. Feldman, "AL, A Programming Language for Automation," Stanford Artificial Intelligence Laboratory Memo AIM-243, Stanford University, 1974.

[15] V. Hayward and R. Paul, "Robot Manipulator Control under UNIX," from TR-EE 84-10, Purdue University School of Electrical Engineering, pp. 22-34, Jan. 1984.

[16] A. Bloch, *Murphy's Law and other reasons why things go gnorw.* Los Angeles: Price/Stern/Sloan, 1977.

[17] M.A. Wesley, T. Lozano-Perez, L.I. Lieberman, M.A. Lavin, and D.D. Grossman, "A Geometric Modeling System for Automated Mechanical Assembly," *IBM J. Res. Dev.*, vol. 24, pp. 64-74, Jan. 1980.

[18] L.R. Nackman and R.H. Taylor, "A Hierarchical Exception Handler Binding Mechanism," *Software--Practice and Experience*, vol. 14, no. 10, pp. 999-1003, Oct. 1984.

[19] B. Shimano, "VAL: An industrial robot programming and control system", *Proc IRIA Sem. of Languages and Methods of Programming*, Rocquencourt, France, pp. 47-59, June 1979.

[20] C.M. Brown, "PADL-2: A Technical Summary," *IEEE Comp. Graphics & Applications*, vol. 2, no. 2, pp. 69-84, Mar. 1982.

[21] W. R. Hamilton, *Elements of Quaternions*, Third Edition, New York: Chelsea Pub. Co., 1969.

[22] R. H. Taylor, "Planning and execution of straight line manipulator trajectories", *IBM J. of R. & D.*,vol. 23, no. 4, pp. 424-436, July 1979.

[23] R. H. Taylor, *A Synthesis of Manipulator Control Programs from Task-Level Specifications.*, PhD Dissertation, Memo AIM-282, Artificial Intelligence Laboratory, Stanford Univ., Stanford, CA, 1976.

Satyr and the Nymph[†]
Software Archetype for Real Time Robotics

J. Bradley Chen, Brian S. R. Armstrong, Ronald S. Fearing and Joel W. Burdick
Stanford Artificial Intelligence Laboratory
Department of Computer Science
Stanford University

Nymph is a multiprocessor system created at Stanford University for the implementation of real time control systems. After a brief description of the nature of Nymph, the design and implementation of the system software is discussed. Software in Nymph emphasizes the benefits of abstraction in the user interface, with an awareness of limitations abstraction creates when used in excess. The distinction between latency and throughput, and its importance in the design of real time multiprocessor software is discussed, motivating use of synchronized parallel processes. Discussions of implementations of the Cosmos force control system and a control system for the Stanford/JPL hand reflect further experiences with Nymph. A general discussion of debugging a tightly coupled multiprocessor follows; what makes it unusually difficult, techniques that have been useful with Nymph, and possible hardware/software strategies for creating debugging environments in future systems.

Introduction

The architecture of multiprocessor systems for high performance real time control applications creates unusual problems for the programmer. The added complexity of parallel software design, and the fact that the programmer is working on an unusual and often unique machine architecture demand creative solutions to old problems such as communication and synchronization. In addition to difficulties inherent in the design of parallel software systems, implementation of this software is made difficult by the lack of development tools, such as debuggers, for a tightly coupled real time system.

The difficulties created by the multiprocessor architecture are enhanced by the demands and requirements of the software to be implemented. The programmers of real time control systems are aware of the performance of which the multiprocessor is capable, and they insist upon having all of the computing power available to them. Because of this, common issues of system design such as security and quality of user interface subordinate to the singular goal of computational efficiency. The challenge for the system software designer is to implement a programming environment conducive to software development without compromising performance.

Software for the Nymph system has been designed to achieve a good user interface within the requirement of minimal system overhead. The general philosophy in design is that people need the abstraction and refined user interface provided by a sophisticated operating system, but robots don't.

Additionally, for optimal utilization of hardware resources in a real time multiprocessor environment, the programmer must have a clear understanding of how parallel processes interact. Conventional knowledge of serial processing is not sufficient. Experience with and awareness of the peculiarities of parallel computing have helped us create software which allows top performance from the Nymph hardware. One significant example in which intuition will misguide the programmer without experience in parallel environments is the choice between synchronous and asynchronous processes. This issue is discussed in the Synchronization section of this paper.

The multiprocessor environment creates new problems in debugging. Beyond the new kinds of errors which occur in parallel software design, the debugging tools and techniques available in a tightly coupled environment are limited. In the Debugging section of this paper debugging problems specific to multiprocessors, and some techniques for coping with these problems are discussed.

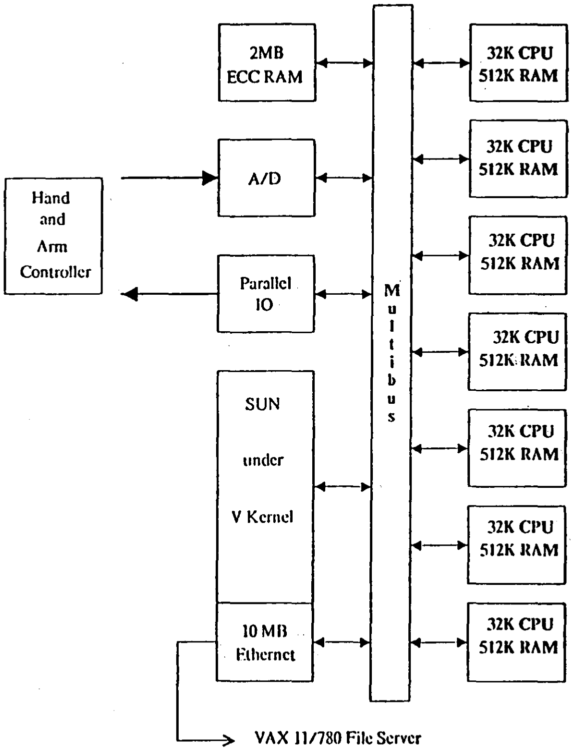


Figure 1. The Nymph System.

[†] Not Your average MultiProcessor Hack

Related Projects

The diminishing size and cost of high speed microprocessors have made multiprocessor systems a popular solution to the problem of computational architectures for real time robotic control. A general overview of the Nymph system can be found in [Chen et. al. 1986].

Researchers at the MIT AI Labs are working with a multiprocessor which uses MC68000 series computers on a common bus, linked to a VAX computer via a DMA link. This system is designed for the implementation of a control system for the Utah/MIT dexterous hand. The system utilizes a mail-box communications model and scheduling of asynchronous processes by means of a servo-loop-scheduler routine. [Siegel et. al. 1986]

Another interesting system is being worked on by the National Bureau of Standards. Theirs is a loosely coupled multiprocessor system, with a fine grained modular structure for a programming model, encouraging generic modules. [Haynes and Wavering 1986]

Members of the Manufacturing Research Department of the IBM T.J. Watson Research Center have developed a multiprocessor control system to support research in automation programming and motion control. Their system consists of a Programming System for program development and user interaction, linked by a shared memory bridge to a Real Time System, which is a tightly coupled multiprocessor. The system also utilizes a Verb structure for command syntax. [Korein et. al. 1986]

Gaglianello and Katseff of AT&T Bell Labs have done work on a distributed system for control applications called Meglos. The system is based on multiprocessor clusters linked by a high speed network, with user interaction and programming environment based on the Unix operating system. [Gaglianello and Katseff 1986]

The Nymph System

In the Nymph system (in its current configuration) eight NSC 32016 single board computers (hereafter denoted as 32k computers) provide the computing power. Each board carries 512k bytes of dual-ported RAM, a parallel port, two serial ports, and a floating point co-processor, which enables a 32k to do a floating point multiply in six microseconds.

In addition to the 32k computers, Nymph includes a MC68010 based Sun Workstation which provides the interface between Nymph and the rest of the world. The Sun communicates with server machines on a 10MB/second Ethernet to provide facilities such as file I/O and downloading of programs. Since the Sun and the 32ks reside on the same Multibus, communication between the machines is very fast. The Sun runs the V-System distributed operating system [Cheriton 1984] with the VGTS windowing system [Lantz 1984]. Using the graphics and I/O routines provided by the VGTS, interaction windows are created for each 32k processor.

Nymph also has 2.5M bytes of EDC RAM, parallel I/O boards to communicate with robot hardware, A-D converters to process force and tactile sensor data, and D-A converters to drive motors. A schematic of the Nymph system appears in Figure 1.

Software Overview

Abstraction can be defined as the removal of hardware specific detail in a communication. Clearly abstraction is useful in communications between computers and people, examples being ascii character encoding and high level languages (versus high and low voltage levels). However, in communications between machines and computers, the abstraction is only useful in as much as it makes it easier for people to control the machines' communication. It does nothing to improve the quality of interaction between machines. Additionally, abstraction tends to produce several determental side affects:

1. Inefficiency. This problem is analogous to adding bureaucracy to government. Abstraction in communications generally results in making the communication path more complicated, and therefore slower.
2. Loss of Robustness. As paths of communications become more complex, the number of possible states in the computation increases, making it less and less likely that the programmer will (or will be able to) test all the possible failure modes of the software.
3. Loss of access to the machine. In removing detail from a communications path, that detail often becomes inaccessible, along with some of the functionality/flexibility it provides.

In such a machine oriented environment as a robotics control system, these problems are precisely those which we seek to avoid. Thus, there are many cases where abstraction would be appropriate in typical computer systems, but becomes an unpleasant burden in a control system. Software for the Nymph system has been created with the problems of abstraction in mind, with a goal being to create a system that uses abstractions when clearly worthwhile, and avoiding abstraction when it seems reasonable to do without it.

In the Nymph system, these concepts about abstraction have been applied to the implementation of the Nymph run-time environment. Nymph was designed with the Sun Workstation dedicated to user interaction, and the 32k processors strictly dedicated to the real time system. Thus, abstraction abounds on the Sun, where it is useful in creating a refined user interface, but is confined to the Sun, and avoided in the 32k. On the 32ks, the real time control system gets 'the whole machine,' simplifying the implementation of time critical control. When a 32k requires some sort of operating system service, such as file I/O, it uses facilities provided by the Sun in a client-server relationship. Thus, the 32k computers can exert an interrupt on the Sun to have there system requests serviced, but the Sun can't interrupt the 32k, keeping its environment clear of unpredictable interrupts and consistently available for real time computations.

There is not a complete "operating system" which runs on the 32k processors of Nymph, but rather a collection of run-time libraries which interface the 32k processors to the Sun and V-system. This creates a convenient interaction environment for the programmer, providing a familiar procedure call interface to system facilities provided by the Sun. Also the runtime library implementation insures that the resources consumed by system overhead is limited to that which the application requires.

Nymph Windows

Due to restrictions in design and a primary concern for efficient use of hardware resources, many real time systems have a very primitive user interface. This is unfortunate, because when hardware systems are inconvenient to use, either the user's time is wasted or the system is not used at all.

The Sun Workstation of the Nymph system was selected because of the excellent user interface and abundance of software which was locally available for it. In letting operating system people develop the operating system, we have been able to concentrate our efforts on the problems relevant to real time systems.

User interaction between the programmer and the 32k processors on the Nymph system occurs on the workstation console. Through the VGTS system, I/O windows are created for each 32k processor. For each window there is an I/O handler process running on the Sun. The 32k computers communicate with the handler processes using the message passing system described in [Chen et. al. 1986]. The interaction windows are initialized by a simple procedure call from a C program, which defines **stdin**, **stdout**, and **stderr** in terms of the interaction window, as well as initializing facilities for file I/O. Figure 2 illustrates some of the mechanisms of the Nymph I/O system. In this case, abstraction has been used to the fullest extent: each 32k processor has its own output device, but the details of the communication between the 32k processor and the interaction windows are hidden to the program. However, the computational burden of the abstraction is confined to the Sun, preserving the real time environment of the 32k processors.

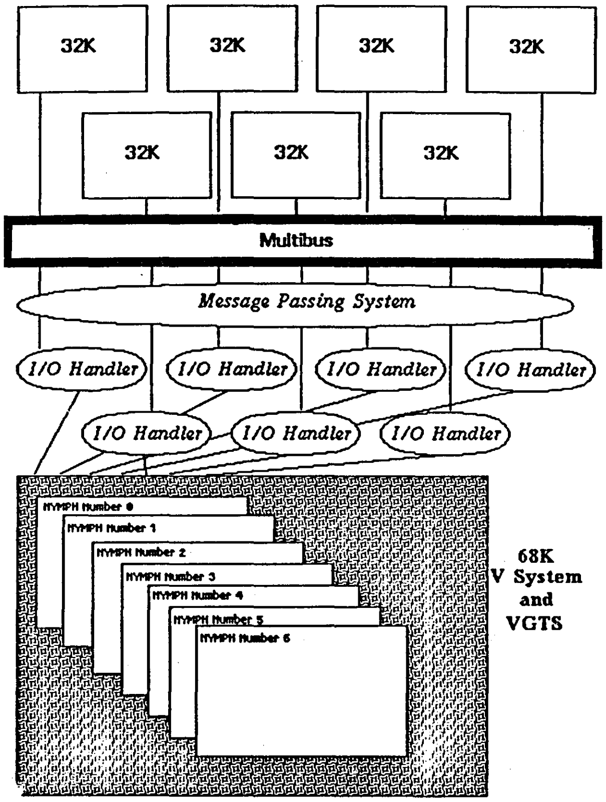
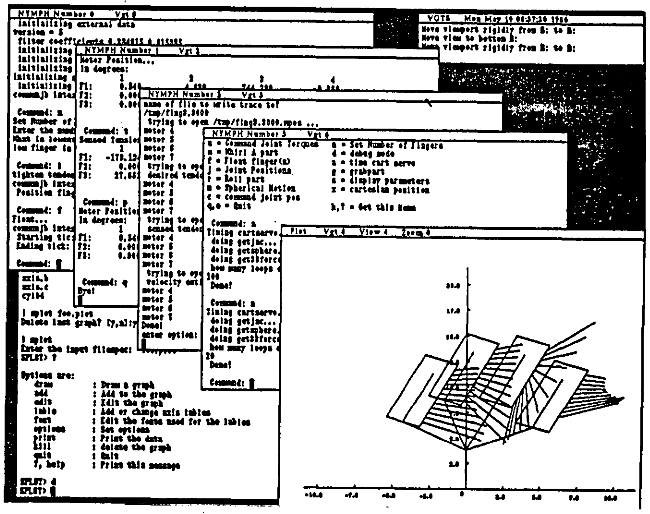


Figure 2. Mechanisms of Nymph I/O.



```
NYMPH Number 0     Vgt 0
 initializing external data
version = 5
 filter coefficients 0.994815 0.015985
 initializing
 initializing
 initializing
initializing
 initializing
commjb inter

 Command: n
Set Number of
Enter the numb
What is lowest
low finger is

 Command: l
tighten tendo
commjb inter
 Position fing

 Command: f
Float...
commjb inter
 Starting tic
 Ending tick:

 Command:
```

```
NYMPH Number 1     Vgt 2
Motor Position...
In degrees:
           1
F1:      0.840
F2:      0.000
F3:      0.000

 Command: t
Sensed Tensio
           1
F1:   -179.12
F2:      0.00
F3:     27.68

 Command: p
Motor Positio
In degrees:
           1
F1:      0.840
F2:      0.000
F3:      0.000

 Command: q
Bye!
```

```
NYMPH Number 2     Vgt 3
name of file to write trace to?
/tmp/fing0.3000
 trying to open /tmp/fing0.3000.mpos ...
motor 4
motor 5
motor 6
motor 7
 trying to ope
 desired tendo
motor 4
motor 5
motor 6
motor 7
 trying to ope
 sensed tendon
motor 4
motor 5
motor 6
motor 7
 trying to ope
 velocity ent
motor 4
motor 5
motor 6
motor 7
Done!
enter option:
```

```
NYMPH Number 3     Vgt 6
a = Command Joint Torques     n = Set Number of Fingers
w = Whirl A part              d = debug mode
f = Float finger(s)           s = time cart serve
j = Joint Positions           g = grabpart
r = Roll part                 s = display parameters
m = Spherical Motion          x = cartesian position
c = command joint pos
q,e = Quit                    h,? = Get this Menu

Command: a
Timing cartserve.
 doing getjac...
 doing getsphere.
 doing get3dforc
 how many loops
100
 Done!

Command: a
Timing cartserve.
 doing getjac...
 doing getsphere.
 doing get3dforc
 how many loops
20
 Done!

Command:
```

```
VGTS     Mon May 19 08:37:30 1986
Move viewport rigidly from B: to B:
Move view to bottom B:
Move viewport rigidly from B: to B:
```

```
 axis.b
 axis.c
 cyl04

! mplot foo.plot
Delete last graph? (y,n):y

! mplot
Enter the input filespec:
SPLOT> ?

Options are:
    draw         : Draw a graph
    add          : Add to the graph
    edit         : Edit the graph
    lable        : Add or change axis lables
    font         : Edit the fonts used for the lables
    options      : Set options
    print        : Print the data
    kill         : delete the graph
    quit         : Quit
    ?, help      : Print this message

SPLOT> d
SPLOT>
```

Figure 3. Nymph Windows.

The windowing capability of the Nymph system has proven to be a great convenience in the use of the system. It allows the user to interact quickly and efficiently with all the processors, with all facilities available from the mouse and keyboard. It also allows (conceptually) simultaneous output from all processors, a feature useful in debugging. Figure 3 shows the user's view of Nymph.

It should be noted that this refined user interface would have been difficult to achieve if we had had to create the Sun operating system and windowing system ourselves. We choose to adopt an existing system, rather than reimplement it ourselves. This greatly simplified the implementation of the user interface.

Synchronization

The Motivation

In studying the performance of real time control systems implemented on multiprocessors, one must draw a distinction between system throughput and computation latency. In a uniprocessor throughput and latency are reciprocals; a calculation that has a latency of 10 milliseconds between input and output will run at 100 samples per second. In a multiprocessor, throughput will in general be greater than the reciprocal of the latency. High throughput is the motivation behind a number of pipeline architectures, and for most computation is a reasonable design goal. But for real time systems latency is critical. Increasing the throughput without decreasing the latency simply accelerates the generation of old data. We will show that synchronization decreases latency, though throughput is also decreased.

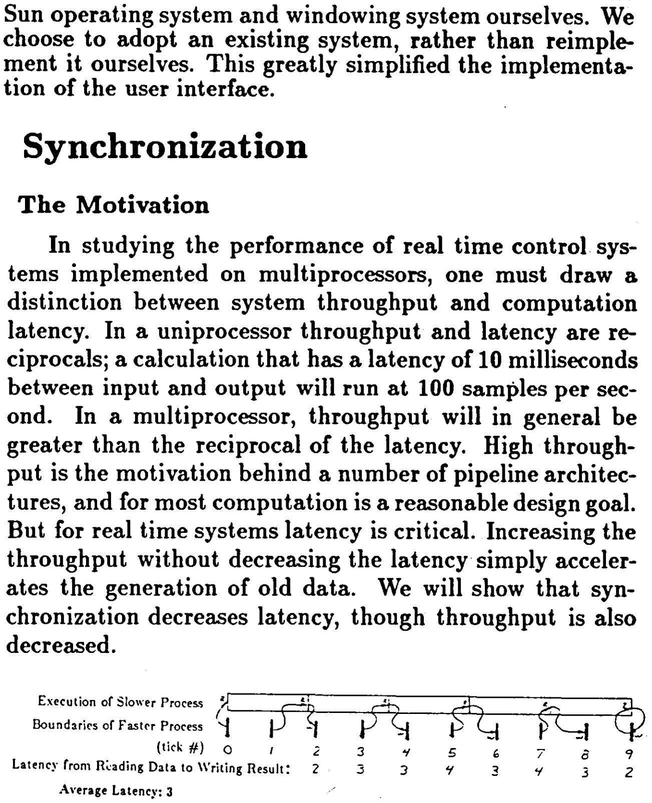

Execution of Slower Process
Boundaries of Faster Process
(tick #) 0 1 2 3 4 5 6 7 8 9
Latency from Reading Data to Writing Result: 2 3 3 4 3 4 3 2
Average Latency: 3

Figure 4. Asynchronous Process Communications and Latency.

In Figure 4 an example of the possible timing between two interacting processes is shown. The faster process is assumed to produce the input and consume the output of the slower process at the beginning of each faster process cycle (tick mark in the figure; we shall refer to the fast process intervals as ticks). This example might arise in a system with a high speed I/O process synchronized with external devices. The top portion of Figure 4 shows the alignment of the slower process with the faster one. We have made the favorable assumption that the process speeds are in an integer ratio, 5:9, and that the slower process lines up exactly with the faster one every nine ticks. Along the bottom of the figure are numbers to indicate how many ticks have elapsed since the reading of the data used by the slower process to compute the result used by the faster process at the tick shown. The mean of the numbers in this row is the mean latency.

Looking at the second execution of the slower process in Figure 4, one sees that it begins at $t = 1.8$, and produces its result at $t = 3.6$. When the process began, the freshest input data came from $t = 1$, and thus was 0.8 ticks old. The result of the second execution of the slower process is applied at $t = 4$ and $t = 5$, the result of the third execution of the slower process then becomes available at $t = 5.4$ and is applied at $t = 6$. At $t = 5$, the system applies a result based on the data measured at $t = 1$. The data and the result are 4 ticks old, even though the slow process requires only 1.8 ticks to execute. The mean latency for the asynchronous case is 3 ticks.

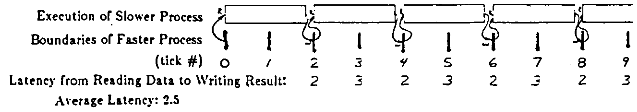

Execution of Slower Process
Boundaries of Faster Process
(tick #) 0 1 2 3 4 5 6 7 8 9
Latency from Reading Data to Writing Result: 2 3 2 3 2 3 2 3
Average Latency: 2.5

Figure 5. Synchronous Process Communications and Latency.

In Figure 5, the same process interactions are shown for the synchronized case. Here the slower process blocks until fresh data is available; it is synchronized with the faster process. Notice that 10% of the computing power of the slower processor is sacrificed while the process blocks for synchronization. As in Figure 4, the latencies are shown. The mean latency for the synchronized case is 2.5 ticks, a 16% reduction from the asynchronous case, even though the process ran five times in the asynchronous case and in the synchronous case only four. The asynchronous case also shows a broader range of latency than does the synchronous case.

A general analysis of asynchronous process latency can be had by assuming the phase relation between the faster and slower processes is a random variable with uniform distribution. The expected value of latency resulting from asynchronous process interactions is given by Equation 1. In the following, phase is measured in units of the ticks and ranges from 0 to 1. Equation 1 is derived from the interaction model of Figures 4 and 5 by summing the latency contribution over all possible event orders. Figure 6 shows the possible event orders for the case where $u = 1.8$ ($(u - \lfloor u \rfloor) > 1/2$). When the phase is between 0 and ϕ_i, the mean latency is $\mathrm{E}\{l_2^3\}$, which is 2.5. When the phase is between ϕ_i and ϕ_j the latency is $\mathrm{E}\{l_3^3\} = 3$. The final possibility is that the phase is greater than ϕ_j, which produces a mean latency given by $\mathrm{E}\{l_3^4\} = 3.5$.

$u = 1.8 \quad i = 2 \quad j = 4 \quad \phi_i = 0.2 \quad \phi_j = 0.4 \quad \lfloor u \rfloor = 1$

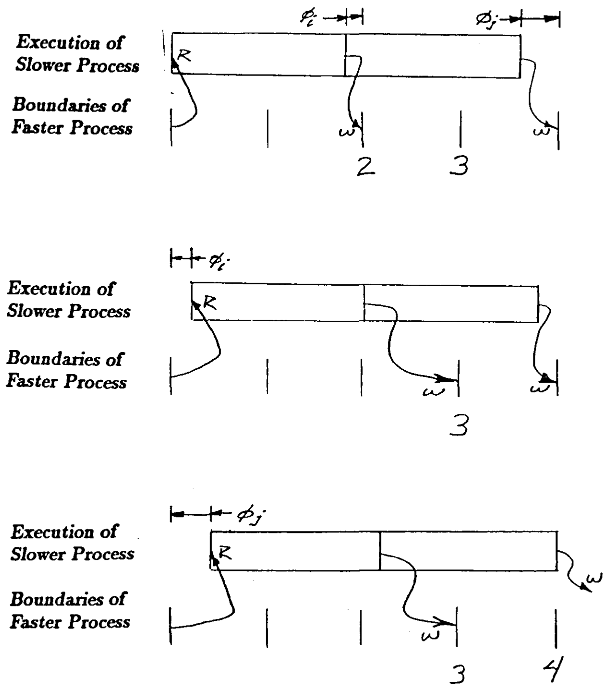

φi
φj
Execution of Slower Process
Boundaries of Faster Process
R
ω
2
3
φi
Execution of Slower Process
Boundaries of Faster Process
R
ω
3
φj
Execution of Slower Process
Boundaries of Faster Process
R
ω
3
4

Figure 6. Possible Permutations of Event Order for two Processes of rate ratio 1.8:1, Asynchronously Interacting.

The three event orders correspond to the three product terms of the expected latency. A separate equation is needed for the case where $(u - \lfloor u \rfloor) < 1/2$ because a new permutation of events is possible and one permutation shown in Figure 6 is not possible. The two equations give identical result when $(u - \lfloor u \rfloor) = 1/2$.

If $(u - \lfloor u \rfloor) \leq 1/2$,

$$\mathrm{E}\{L\} = P_{(\phi_j - 0)} * \mathrm{E}\{l_i^{i+\lfloor u \rfloor - 1}\} + P_{(\phi_i - \phi_j)} * \mathrm{E}\{l_i^{i+\lfloor u \rfloor}\} + P_{(1-\phi_i)} * \mathrm{E}\{l_{i+1}^{i+\lfloor u \rfloor}\}$$

if $(u - \lfloor u \rfloor) > 1/2$, (1)

$$\mathrm{E}\{L\} = P_{(\phi_i - 0)} * \mathrm{E}\{l_i^{i+\lfloor u \rfloor}\} + P_{(\phi_j - \phi_i)} * \mathrm{E}\{l_{i+1}^{i+\lfloor u \rfloor}\} + P_{(1-\phi_j)} * \mathrm{E}\{l_{i+1}^{i+\lfloor u \rfloor + 1}\}$$

where $P_{(\alpha - \beta)}$ is the probability that the phase relation between the slow and fast processes lies in the region between α and β; and $\mathrm{E}\{l_m^n\}$ is the expected latency when a result of the slow process is applied from ticks m to n, where m and n are measured with respect to the tick at which the raw data of the slow calculation are taken.

$$E\{l^n_m\} = \frac{\sum_{k=m}^{n} k}{n-m+1} = \frac{n(n+1)-m(m-1)}{2*(n-m+1)}$$

When the phase relation between the slower and faster processes is uniformly distributed,

$$P_{(\alpha-\beta)} = \alpha - \beta$$

Note that the phase relation ranges from 0 to 1, not from 0 to 2π. The remaining terms in Equation 1 are defined to be:

u	is the rate ratio of the two processes;
i	is the least integer greater than u ;
j	is the least integer greater than $2u$;
ϕ_i	is $(i-u)$;
ϕ_j	is $(j-2u)$;
$\lfloor u \rfloor$	is the greatest integer less than u.

The expected latency for the asynchronous and synchronous calculations, as a function of the process rate ratio, is shown in Figure 7. The stair step shape describes the synchronous case, where latency is fixed for increasing u until a boundary is reached and one more tick is required to complete the slower calculation. Figure 7 shows that for rate ratios above 1.3:1 the synchronous case always gives less expected latency. For ratios just greater than an integer, the reduction in latency by synchronization is a few percent; for ratios just less than an integer, the reduction in latency by synchronization is more than a full tick. The expected latency for a rate ratio of 1.8:1 is 3.2 ticks. In the example of Figure 4 we assume that the asynchronous processes "line up" every nine ticks, with a lower expected latency of three ticks. The integer ratio "lining up" assumption was made to show the asynchronous construction in optimal conditions.

Latency is pure delay: the anathema of feedback control. In this final thrust to convince the unbelieving that you can throw away computer power and still improve control, we will compare the tracking of a linear first order filter by its equivalent discrete filter under four assumptions: (1) uniprocessor, (2) two asynchronous processors, (3) two synchronized processors, (4) a uniprocessor of twice the power. For this numerical exercise we will take our input signal to be 10 hertz, our desired linear filter to have a pole at 10 hertz, our uniprocessor (example 1) to have computation power sufficient to sample at 80 hertz, and our problem to be amenable to parsing in chunks with a ratio of 1.8:1. The results are shown in Table 1. The error measured is the difference between the linear filter and the digital equivalent. The digital filter has been designed with the forward integration approximation. The excess lag (excess over the linear filter lag of 45°) was measured by finding the advance that should be supplied to the digital filter to minimize the squared error. The results show that by going to two processors a substantial improvement is achieved, and that by synchronizing a further improvement is achieved. The fourth configuration, a uniprocessor of twice the power, is included to show that two processors are not twice as powerful as one.

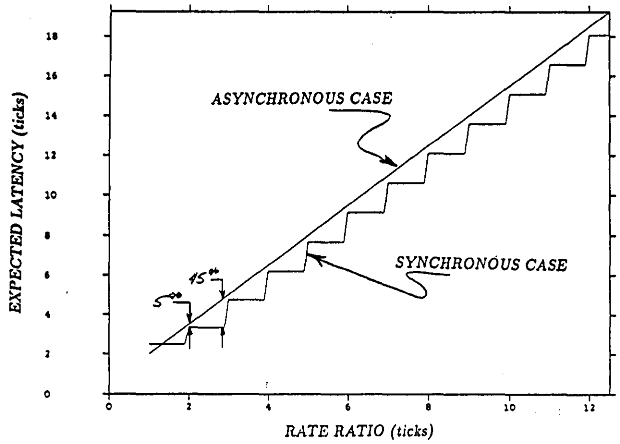


Figure 7. Expected Latency in Synchronous and Asynchronous Process Coordination.

Table 1. A Comparison of Synchronized and Unsynchronized Digital filters. The error measures are relative to a single pole linear filter with an input signal of 10 hertz.

Filter Configuration	Mean Squared Error (unit input)	Excess Delay (milliseconds)
A uniprocessor at 80 hertz	0.558	25.7
Two processors, problem split 1.8:1, fast rate: 224 hertz, slow rate: 124.4 hertz, slow process unsynchronized.	0.280	16.9
Two processors, as above, fast rate: 224, slow rate: 112, slow process synchronized. (see Figures 4 and 5)	0.235	14.3
A uniprocessor at 160 hertz.	0.156	12.6

The Implementation

The basic design of the Nymph synchronization primitives is presented in [Chen et.al 86]. Since the writing of that paper, in implementing several control systems, one lesson on is clear: Asymmetric synchronization primitives are often needed, as opposed to the symmetric synchronization primitives of the original implementation. By symmetric we mean that the behavior of the synchronization primitives, `Synch_Signal()` and `Synch_Wait()`, depends only on the order of occurrence, not on the processor upon which the commands occur. Thus if processors A and B are synchronized, B will block if it arrives at its `Synch_Wait()` first, to be awakened by A; or A will block if it arrives at its `Synch_Wait()` first, to be awakened by B.

There are cases in which one wants B to block for A, but not vice versa, or for B to block only if A is on some critical segment. In both of these scenarios A never blocks for B, and B never does any awakening. The primitives Synch_Block(n, patience) and Synch_Check(n, patience) were written for these cases. As in [Chen 86], n is an index indicating the synchronization event, and patience is a parameter indicating how long the processor is willing to sleep.

Synch_Block() is used when one or more processors must block until one processor reaches some event. It is useful for MASTER-SLAVE interactions and for synchronizing several processors to a single time base.

Synch_Check() is used for mutual exclusion of a critical section. A typical example is in I/O channels that have internal state and thus may not be used by multiple processors concurrently. The priority processor, or the one expected to be first, will execute Synch_Signal() before entering the critical section (possibly long before) and Synch_Wait() upon leaving the section. The next processor will execute Synch_Check() before entering its critical section; if the primary has completed its action, the secondary will continue; if the primary has not completed its action, the secondary processor will block. The advantage of this special construction is that Synch_Check() appears as a single if statement when the program does not block, and runs in 8 microseconds. Thus, Synch_Check() can be used liberally to ensure event order without sacrificing performance

The example shown in Figure 9 illustrates the use of Synch_Block() and Synch_Check(). Awakenings are indicated by arrows. A Synch_Block() or a Synch_Check() can only be awakened by a Synch_Wait().

Nymph for Hand Control

The Stanford/JPL hand has three fingers with three degrees of freedom each, actuated by a coupled pulley system with 4 tendons and motors for each finger, for a total of twelve motors. Each tendon has a tension sensor mounted near the finger to allow control of joint torque, and motor shaft encoders to determine motor position. This hand has some dexterity for fine motion and force control of grasped objects, and for regrasping operations where objects are reoriented within the hand [Fearing, 1986].

The previously proposed hand control structure [Chen et al 1986] has been revised to provide a high bandwidth tendon controller (see Figure 8). The system is currently running with seven processors, with a force and tendon processor for each finger. The low level tendon processor runs at 480 Hz, and the force servo on a separate processor runs at 120 Hz, and the two are tightly coupled by synchronization. The seventh processor coordinates finger actions. Three more processors wil be added for the processing of tactile data.

The three low level processors share a 12 channel motor interface, and synchronization is used to prevent collisions when accessing this device. This also reduces potential delays due to all three processors contending for the bus simultaneously if they were locked in step. The second use of synchronization is the transfer of data between the high and low servo levels. This ensures minimal latency and prevents the use of partially updated information by the two levels.

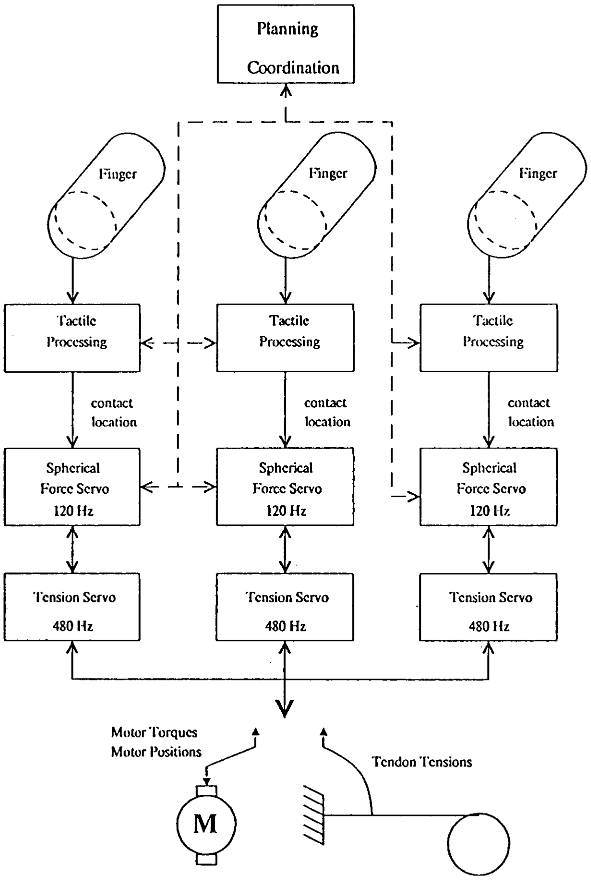


Figure 8. Multiprocessor System for the Stanford/JPL Hand.

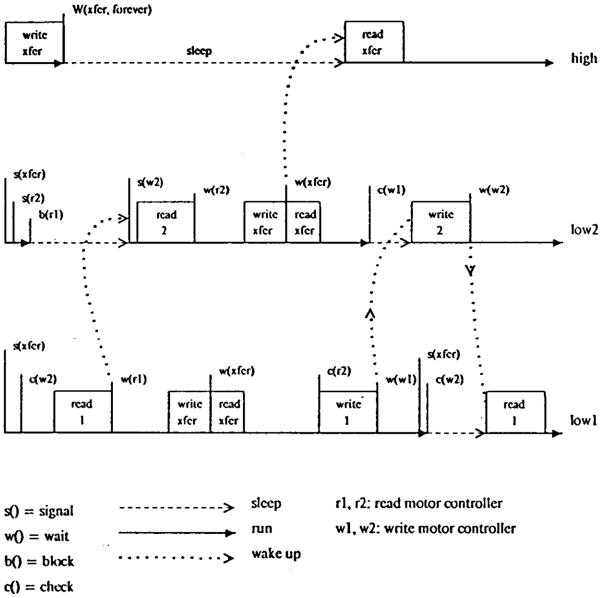


Figure 9. Synchronization of low and high level control.

Figure 9 shows a portion of the synchronization scheme. Processor "low1" is the master processor and sets the timing for the other processors. There are five separate synchronization events: two reads and two writes of the motor controller, and filling the transfer buffer between the high and low levels. The high level writes desired tendon tensions into the transfer buffer and waits `Synch_Wait(transfer, for_ever))` until all low level processors have written measured motor positions into the buffer. The high level can now proceed with correct data, and the proper servo rate.

Signals are used to protect the motor controller from accesses by other processors by forcing the other processors to "`Synch_Check()`" and thus wait until the signal is cleared by `Synch_Wait()` on the signal issuing processor.

Cosmos

COSMOS is an experimental programming system designed to facilitate experiments in manipulator position and force control. The control algorithm used in COSMOS is based on the operational space method, described in [Khatib and Burdick 1986]. Parallelism can be extracted from this algorithm by conceptually dividing the computations into two levels:

1. A "high level" system which computes the configuration dependent dynamic models at a relatively lower rate.
2. A "low level" servo system which computes the forward kinematics and servo equations at a faster rate using sensor data and the dynamic data from the "high level."

The low level system measures the manipulator position and end-effector forces, and then computes joint torques using a series of vector and matrix operations. The vector and matrix elements used in these computations are dependent on the configuration of the manipulator, and are updated by the high level. The high level requires processed sensor data from the low level in order to perform its computations. Thus, time dependent data passes in both directions between the high and low level. A third level, the "programming level" interacts with the manipulator programmer, performing run time program execution, path planning, and error checking. The computations of each of these levels can be further divided among different processors to extract more parallelism, reducing latency in the overall computation.

Because of the decomposition of the algorithm into two major levels, and the decomposition of each of these level into separate computational tasks (which are implemented on different processors) various levels of synchronization and data transfer occur in the COSMOS system. Within each level, very high speed symmetrical synchronization (on the order of a few microseconds) occurs between processors. The high level computations are asymmetrically synchronized with the low level in order to minimize the latency in the low level computations, which is the crucial latency for stability of the control algorithm. In the current COSMOS implementation on the NYMPH system, a low level kinematics and servo rate of 200 Hz and a high level dynamics computation rate of 100 Hz has been achieved.

While COSMOS uses one particular control algorithm, most advanced manipulator feedback control algorithms can be decomposed into fast and slow loops, with the minimization of the latency in the fastest loop being critical to the success of the implementation. Thus, for optimal implementation of a manipulator control algorithm on a multiprocessor, tight coupling of the processors of the fast loop, and asymmetrical coupling of the fast loop with the slower loops is necessary to minimize latency.

Debugging in a multiprocessor environment

Software development can be divided broadly into two activities, design and debugging. These activities are related in that greater complexity in design leads to more difficult debugging problems. The design of multiprocessor software is made more complex by the need to take advantage of parallelism in the computation. Genuine concurrency and inability to guarantee the order of events can lead to errors from programmers who are used to working in a serial environment. Another source of bugs is that processors share memory, and so they are able to corrupt eachothers' memory. This creates situations in which the bug in the system is not necessarily in the software which fails, but rather in another piece of software which is irresponsible in its memory activities. The result is that new sources of program bugs exist in the multiprocessor software, in addition to the errors common in serial software.

Debugging for real time robotic control applications is additionally complicated for two reasons. First, synchronization and timing within the system is critical down to the level of a few microseconds, and so debugging systems that use the single step method are useless, since the critical timing information is lost. Second, since the multiprocessor is connected to robotic hardware, much of the input to the software algorithms is sensor data, which is difficult to predict or model in a software simulator.

In the planning stages of this project it was recognized that there should be a convenient and fast (low overhead) monitoring system for keeping track of many processors simultaneously. At the speeds involved in real time control, monitoring can only be done by hardware. The hardware solution is a meter, an LED, and an input switch permanently connected to each processor for diagnostic purposes.

The meter has three primary and switchable functions: measuring bus utilization, showing the time spent waiting in synchronization, and showing the execution time for various sections of the servo loops. The meters immediately show the case where processors lose synchronization, because there will be significantly different waiting times if one processor misses a synchronization event. The meters are also useful for determining which processor amongst the various processors is responsible for "wedge" conditions. Because the meters indicate how much time each processor is idling, the applications programmer can use the meters in the early stages of software design to rebalance the computational load among processors to minimize latency. An additional meter monitors activity on the global Multibus.

The LED can be turned on and off under software control in critical sections of code, and is useful for determining the exact sequencing of events. The software readable input switch is primarily useful not so much for debugging, but as an input for application programs in which the programmer is interested in switching between different servo algorithms in real time for comparison purposes. Individual non-maskable interrupt switches are also provided for each processor. Figure 10 is a picture of the Nymph front panel.

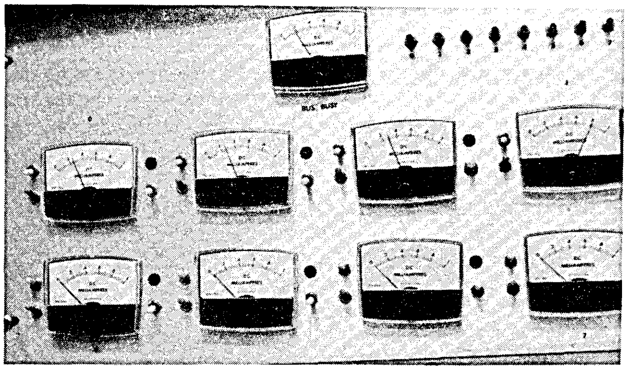


Figure 10. The Nymph Front Panel.

Another important debugging tool for the Nymph system has been the ability to make long traces of control system states without modifying servo code or introducing extra servo delay. This is made possible by having shared variables in globally accessible memory, and having extra processors available that can be dedicated to the trace task. To trace passed variables, for example the data passed from high to low level in the three finger hand control example, all that is needed is to do `Synch_Wait(xfer)` on the trace processor. This ensures that all data is captured and at the same time.

A major source of difficulty in debugging in a real time tightly coupled multiprocessor environment is that there is no central control. There is no intelligent entity on the bus which, in an error situation, is capable of preventing other devices on the bus from corrupting available state information. This makes the development of a runtime system debugger in a multiprocessor extremely difficult.

The central control which does exist is the bus, but this device is generally a slave to the processors in the system which it connects. It is not surprising that many of the more difficult debugging situations are characterized by a bus lock situation, in which some irate processor or processors monopolize the bus, making it impossible to examine state information in the memory of devices on the bus.

One plausible solution to the problem of central control would be to assign a single processor to debugging tasks, give it priority to control the bus, and somehow wake it up whenever an error situation occured. Because of the asymmetry involved, the processor would have to be dedicated to debugging and not a part of the control system, for the debugging system would be useless to debug code running on the debugging processor. The cost of a dedicated processor would be small if it was an effective debugging tool, but its effectiveness would be limited by its ability to capture and analyze the system state at the time the error was detected.

Conclusions

The Nymph system has proven to be a versatile and effective system for the implementation of real time control systems. Much of the success of Nymph can be attributed to the use of the 68k based Sun Workstation with existing software, with which the implementation of the user interface was greatly simplified.

The workstation provides a logical boundary for Nymph software. The 32k processors are dedicated to real time computations, and the Sun is dedicated to system functions. By isolating the system software on the workstation, we have been able to avoid the complications imposed by system software in the real time system, allowing simple and efficient implementations.

In a real time environment, throughput subordinates to latency as the key issue in optimal performance. The use of synchronized processes is a useful technique for minimizing latency in parallel computations.

Debugging in a multiprocessor environment is a difficult problem in which many opportunities for improvement exist. In Nymph we have attempted to provide maximum availability of system resources through user interaction windows, greatly improving the debugging environment. Real time debugging information provided by the front panel and trace processors has also proven to be a useful tool. The main limitation in improving the debugging environment is that real time debugging capabilities require hardware support.

Acknowledgements

The authors wish to gratefully acknowledge Professor Thomas O. Binford for his guidance and support. We would also like to thank Oussama Khatib for his many contributions throughout the project. Special thanks to Lance Berc and Professor Keith Lantz for making available their expertise on the V-System, and all of Stanford's Distributed Systems Group for making the V-System and VGTS available to us. This work was funded by DARPA Contract MDA903-86-K-0002, and by industrial sponsors under the Stanford Institute for Manufacturing and Automation. BSA appears courtesy of the Hewlett Packard Faculty Development Program.

References

J.Bradley Chen, Ronald S. Fearing, Brian S.R. Armstrong, and Joel W. Burdick, "NYMPH: A Multiprocessor for Manipulation Applications," *IEEE Conference on Robotics and Automation*, April 1986, pp. 1731-1736.

David R. Cheriton, "The V Kernel: A software base for distributed systems," *IEEE Software*, April 1984, pp. 19-42.

Ronald S. Fearing "Implementing a Force Strategy for Object Reorientation," *IEEE Conference on Robotics and Automation*, April 1986, pp. 96-102.

Robert D. Gaglianello and Howard P. Katseff, "A Distributed Computing Environment for Robots," *IEEE Conference on Robotics and Automation* , April 1986, pp. 1890-1896.

L.S. Haynes and A.J. Wavering, "Real-Time Control System Software: Some Problems and an Approach," *IEEE Conference on Robotics and Automation*, April 1986, pp. 1705-1712.

Oussama Khatib and Joel W. Burdick, "Motion and Force Control of Robot Manipulators," *IEEE Conference on Robotics and Automation*, April 1986, pp. 1381-1386.

James U. Korein, Georg E. Maier, Russell H. Taylor, and Lawrence F. Durfee, "A Configurable System for Automation Programming and Control," *IEEE Conference on Robotics and Automation*, April 1986, pp. 1871-1877.

Keith A. Lantz and William Nowicki, "Structured Graphics for Distributed Systems," *ACM Transactions of Graphics* , 3(1):23-51, Jan. 1984.

David M. Siegel, Sundar Narasimhan, John M. Hollerbach, David J. Kriegman and George E. Gerpheide, "Computational Architecture for the Utah/MIT Hand," *IEEE Conference on Robotics and Automation* , March 1985, pp. 918-924.

The Meglos User Interface

Robert D. Gaglianello
Howard P. Katseff

AT&T Bell Laboratories
Holmdel, NJ 07733

Abstract: Meglos provides a user-level programming environment for a system of interconnected processors and provides a convenient testbed for developing, testing, and running large-scale robotics applications. Meglos extends the program development and execution environment of the UNIX® Operating System with simple, powerful and efficient communications and synchronization primitives. Each processor has a priority-based preemptive scheduler, simplifying the programming of real-time applications. Meglos provides a convenient environment for writing and debugging applications that require multiple processors for their execution and allows several programmers to simultaneously run and debug their applications.

I. Introduction

As robotics applications exceed the capabilities of computers traditionally used in robotics, it becomes necessary to investigate alternative types of computing systems. An attractive candidate for such a system is one consisting of many interconnected microprocessor-based computers[1]. In addition to providing additional computational power through parallelism, a multiprocessor system exhibits many useful characteristics such as low cost, modular growth, and increased reliability through replication. To achieve these benefits, the cost of communications must be low; small enough so that the communications cost does not exceed the time savings obtained by parallel execution on the different processors. Furthermore, if time-dependent operations are to depend on communications between processors, then there must be a guaranteed maximum latency for interprocessor communications.

This paper describes the programming environment provided by Meglos[2] [3], a multiprocessor system that is well suited to robotics applications. The system consists of up to twelve processors using Motorola 68000 based Multibus computer systems. Each processor is connected to the S/NET[4] [5], a high throughput (80 Mbit/sec), low latency, message-oriented interconnect. The processors communicate by sending messages—no shared memory or other means of communications is provided. A DEC VAX computer running the UNIX operating system serves as a host processor for the system. Programs running on Meglos are controlled from the UNIX System, which provides terminal and file system access for these programs.

The satellite processors are built around the Pacific Microsystems PM-68K single-board computer[6]. This board contains a Motorola 68000 processor clocked at 10 MHz along with 256 kbytes of memory that is accessed with no processor wait-states. Each PM-68K is connected to a separate Multibus backplane. In addition to the single board computer, each backplane contains one or two additional boards with 512 kbytes of memory, a Sky Fast Floating Point Processor board[7], and an interface board that connects the processor to the S/NET interconnect. The Multibus may also contain other boards that interface to devices such as robot arm controllers, sensor interfaces, or frame-grabbers.

The addition of the Sky floating point boards to each processor is a requirement for robotics applications. Without it, all floating point calculations must be done in software. The speed of performing floating point calculations in software is so slow as to be almost useless for time critical computations such as robot arm kinematic solutions. With a Sky board, the floating point performance of a Motorola 68000 is 20 times better than with software floating point and for many applications, is 50% that of a VAX 11/780 with a floating point accelerator[8].

The initial design and implementation of Meglos was influenced by our sister Robotics Research Department. They were interested in several projects that were beyond the capabilities of a single 68000-based computer system and realized that a multiprocessor system based on the S/NET interconnect would be well suited to robotics applications with real-time requirements because the S/NET hardware guarantees that data is always transmitted within a specified time interval.

Because of their interest, many of the features of Meglos are designed to simplify the programming of robotics applications. Its interprocess communications mechanism is well suited for use in real-time applications because it has predictable and small latencies. This mechanism is based on communications paths called *channels* that permit two programs to communicate via message passing. Each channel is independently flow-controlled, so that a program may set up multiple channels to communicate simultaneously with many other programs. Channels may be set up between two programs running on the single-board computers, or between a program on an attached processor and a pro-

CH2345-7/86/0000/0169$01.00 © 1986 IEEE

gram running on the UNIX System.

Each of the attached processors runs the Meglos *kernel*, a real-time multitasking executive that oversees each processor's local resources. The kernel manages the allocation of processing cycles and communications bandwidth for programs running on the processor. For real-time response, there is a priority-based preemptive scheduler and an efficient mechanism for switching between processes. User programs may specify interrupt service routines, change the processor interrupt priority level, and directly access processor I/O space. Meglos provides a mechanism that integrates the scheduler with device interrupts and user-written device handlers. This allows users to customize the attached processors to control a variety of devices such as robot arms and video image processors.

The Meglos operating system allows several independent applications to run at the same time. One of our testbed systems has several processors connected to robot arms and other processors attached to cameras and image processors. Since Meglos makes it easy to write programs without embedded physical processor numbers, the testbed system may be shared by several researchers, each using some of the robotics equipment, without having to reserve the use of specific processors.

Since most of our users are already familiar with the UNIX system, we chose to design the Meglos programming environment as an extension of the UNIX environment. Since the single-board computers do not have disks, terminals or other standard peripherals, a satellite processing scheme is used: each time a program issues a UNIX system call, the call is passed to the UNIX System for execution. On completion of the system call, the results are sent back to the requesting program. This allows Meglos programs to access all files and other facilities on the host UNIX system in the same way as programs running on UNIX do.

II. An interesting application

An interesting robotics application that has been implemented on Meglos is a robot teaching system, MIMIC, in which a robot arm follows and mimics an operator's hand movements in real-time[9]. We refer to this application throughout the paper to illustrate how an application uses various facilities provided by Meglos. The application was motivated by the desire to overcome some of the limitations of teach boxes currently available for teaching robots a desired task. Conventional teach boxes can only teach points along a desired path. This can be difficult to convert into a trajectory. Another limitation is their inability to allow control of all joints simultaneously.

The operator moves a flat circuit board on which are mounted five LEDs in front of a camera. The three-dimensional position and orientation of the board are determined from the camera images and the actual motion is computed. The board's motion is converted into robot motion, by assuming a similar board is attached to the robot's gripper, as shown in Figure 1. The robot arm and gripper then reproduce or mimic this motion. All the calculations take place in real-time, i.e., as the operator moves the board around, the robot arm across the room performs the same movement with no noticeable delay.

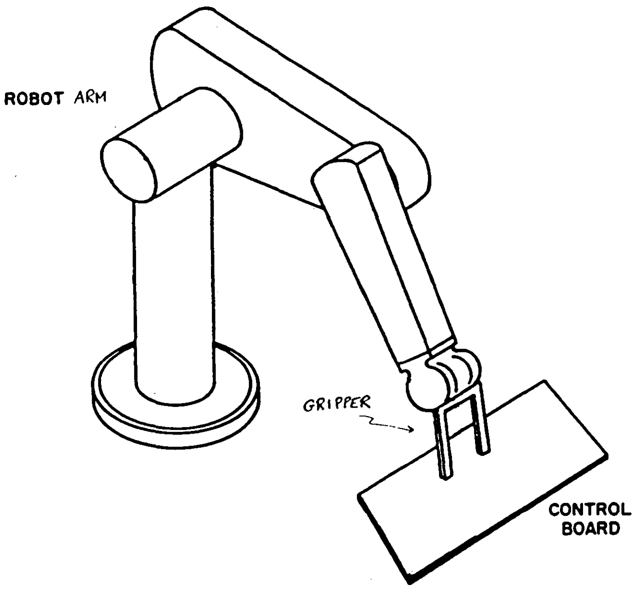


Figure 1. *Robot holding board.*

The application is broken into three modules, each of which runs on a separate processor. See Figure 2. The first module is a data acquisition system that processes data from a solid state camera that captures a frame each 17 msec. The module determines the image coordinates of each LED and sends them across a channel to the next module. Module two determines the position of the board in three space, calculates the board's motion, computes the angles through which the arm and gripper should move, and sends them to module three. This calculation requires 15 msecs. The third module, the robot controller, moves the arm and gripper in the requested fashion.

This application naturally breaks up into several modules executing on separate processors: two device handling modules and a central module that communicates with the others to coordinate their work. An advantage of partitioning the problem in this manner is that general purpose device handling routines can be designed for reuse by many applications. New applications that wish to track LEDs or control the robot need only understand the format for data sent to and from the device routines and need not concern themselves with the detail of handling the devices.

Like other robotics applications, MIMIC requires the use of customized hardware for its implementation. The data acquisition processor uses a commercially available

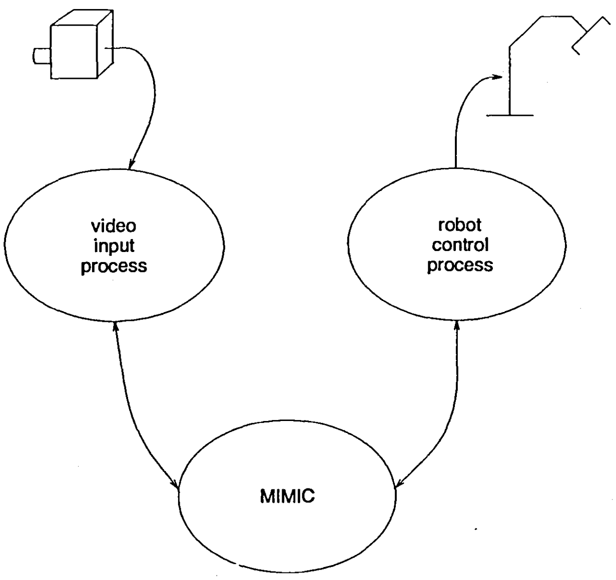


Figure 2. *Block diagram of teaching system.*

frame grabber and image processing system. The robot controller processor uses an in-house interface. Both connect directly to the Multibus. Building Meglos around an industry standard bus and allowing user-written interrupt service routines allows users the flexibility of customized processors. When a new device is to be attached, the kernel does not require being recompiled with a driver for the new device. This is another feature of Meglos that makes it useful for robotics applications.

III. Programmer interface

Like other message-based multiprocessor systems[10], applications are coded in a sequential programming language extended by primitives that provide communications and synchronization. An application typically consists of programs running on many processors that communicate amongst themselves to cooperatively perform the application. Users run and debug programs on the attached processors from terminals connected to the UNIX host. Programming is simplified because the entire "edit–compile–run" debugging cycle can be performed from a single terminal.

For example, to run a program that has previously been compiled into a file named mimic, the command:

```
tx mimic 10 20
```

is issued from the host UNIX system. It causes the file mimic to be downloaded to one of the processors, the arguments 10 20 to be passed to the program, and starts the program running on the single-board computer. If the running program executes a UNIX system call, the kernel on its processor passes the call to the tx program and tx actually executes the call. For instance, if the program issues a `write` to its standard output, the tx program responds by writing the data to its standard output. This exact emulation of the UNIX environment allows programs running under Meglos to make use of input and output redirection, pipes, and the other features of the UNIX System that contribute to its effectiveness as a program development environment[11].

For applications that make use of custom hardware such as robot arms, it is necessary that the program runs on a processor with the proper hardware. For example, the command:

```
tx3 mimic 10 20
```

causes the program to be run on processor number 3. A way to invoke an application that consists of many programs is to create a UNIX shell script[11] that starts each of the programs in the background with a separate tx command.

As an example of how these primitives are used, we show how MIMIC is run on one of our testbed Meglos systems. The data acquisition portion of MIMIC is compiled into the file trakleds and runs on processor 2, the processor with a camera, frame buffer and image processor. The robot controller, mimicarm runs on processor 5, the processor with the robot interface, and mimic, the program that does the calculations can run on any processor.

Processors are made available for use by Meglos programs with the txadd command and removed with txdrop. One typically drops a processor before powering it off, say to change a board in its Multibus. Before running MIMIC, the command

```
txadd -e 2 5
```

could be given. This causes processors 2 and 5 to be made available in the *exclusive* mode, indicating that no other users may run programs on the processors.

After the processors have been added, MIMIC may be started. A shell script that could be written to run MIMIC is as follows:

```
tx2 trakleds vision &
tx5 mimicarm 10 .3 &
tx  mimic vision &
```

This causes the trakleds and mimicarm programs to be started on the appropriate processors and mimic to be run on any processor. The mimicarm program requires two input parameters for setting internal constants within the robot controller. They are passed on the command line invoking the program. The "&" is part of the shell syntax that says to run the program in the background, concurrently with the other programs in the script.

The txstat command is used to display a list of available processors and what is running on each processor. Figure 3 shows the output of a txstat command obtained while MIMIC is running.

It indicates that user rdg is running the three programs comprising MIMIC on processors 1, 2, and 5 and that processors 3 and 4 are available for use. Other items displayed are the total memory on the processor, the memory currently available and the UNIX process id,

```
MACHINE  MEMSIZE  MEMFREE  USER       PID    MEMSIZE  PRIORITY  PROGRAM
      1    1152K    1056K
                             rdg     3733        96K       128  mimic
      2    1152K    1056K  (exclusive use: rdg)
                             rdg     3731        96K       128  trakleds
      3    1152K    1152K
      4    1152K    1152K
      5    1152K    1056K  (exclusive use: rdg)
                             rdg     3732        96K       128  mimicarm
```

Figure 3. txstat *output while MIMIC is running.*

memory usage and Meglos execution priority of each program.

IV. Channels

The communications requirements for multiprocessor robotics applications differ from other types of distributed systems, such as networks of workstations, in several fundamental ways, including requirements for communications performance, naming mechanisms, access control, and fault recovery.

Workstations typically use a client-server model of computation in which a processor (or groups of processors) offer some service, such as file system access, to the other workstations in a system. In contrast, the programs of a robotics application are typically peers and the relationship of client/server or master/slave cannot conveniently be assigned to the programs. This suggests that the Meglos communications mechanisms should be symmetric both in methods for establishing communications and for sending messages between processors.

In Meglos, programs establish communications channels between each other before they communicate and each channel connects only two programs. Channels provide for the transmission of data messages from the address space of one program to the address space of another and provide synchronization between readers and writers.

Communications via channels is independent of where the two programs are executing. They may be on the same or on different single-board computers or on the host UNIX System. It is this feature of channels that allows several independent applications to run on Meglos at the same time. To establish a channel between two programs, both must first agree on a name for the channel. The two programs then issue a `copen` system call. The channel name is passed to `copen` and it returns a channel descriptor for that channel. If a program attempts to read or write a channel before a second program does a `copen` for the channel, the read or write blocks until the other program does its `copen` and the channel is set up. The first read or write call may also return an error if a problem occurs while opening the channel. Note that no indication of channel direction is passed to `copen`. This is because channels are bidirectional and thus can always be either read or written by the program.

The performance of channels is sufficient to meet the demanding real-time performance requirements of robotics applications, yet they still provide transparent and automatic recovery from faults that are likely to occur while a program is running such as receiver buffer overflow or transient communications errors. Channels provide a good match with the communications patterns of robotics applications because such applications can often be structured as a set of programs whose communications topology is either fixed or changes slowly over time.

With channels, the operating system provides flow control that allows a program that is accepting input from more than one channel to read input from the channels in whatever order is most convenient for the program. Communications mechanisms without independent flow control do not allow a program to chose which channel data is to be received from. Instead, each time a program reads data, it gets whatever message has next arrived on any of its channels, so that a program may be flooded with messages that it is not ready to deal with. We did not use a more general scheme, such as ports that allow multiple readers and writers, because the protocols for flow-control and for the detection and recovery from transient communications failures can be implemented more efficiently when the reader and writers of messages are known in advance[12].

Programs send messages on a channel by issuing a `write` call. `Write` requires three arguments: a channel descriptor (obtained from a `copen` call), a pointer to the message to be sent, and the length of the message, in bytes. When the `write` returns, the sending program is assured that the reader has received the message with no errors.

To accept data from a channel, a program issues a `read` call. `Read` requires three arguments: the channel descriptor, a pointer to a local buffer, and the size of the buffer, in bytes. If a `write` has already been issued by the program at the other end of the channel, then the message sent on the channel is placed into the local buffer and the `read` call returns. If a `write` has not been issued, then the `read` blocks waiting for one. When the `read` call completes, it returns the length of the message that was placed in the buffer.

Programs sometimes need to coordinate data originating from several channels. A mechanism provided for such programs is a multiplexed input primitive. readn, to which an array of channel descriptors is passed. When a message arrives on any one of these channels, it is placed into the buffer specified in the readn call and the channel descriptor for that channel is returned to the calling program. If input is simultaneously available on more that one channel then the channel number that occurs first in the array of descriptors is returned. By using this mechanism, a program can implement either prioritized or round-robin service for input channels. Readn provides multiplexing capabilities that are similar to those of the 4.2BSD select call[13].

To close a channel, a program issues a close call. After a program closes its end of a channel, any attempts to read or write on that channel return an error. After reading the last data sent before the close, reads by the program at the other end of the channel return an end-of-file indication. Similarly, writes to a channel that is closed indicate end-of-file.

In a message-based system, two measures of communications performance, latency and throughput, provide programmers some idea of how their application would perform, and for robotics applications with real-time requirements, indicate whether they can be implemented at all. Latency is defined as the elapsed time between when a message is sent and when the sending processor is notified that the message is received by the destination processor. Throughput is defined as the number of bytes one program can send to another in one second. Guaranteed low latencies for communications between user programs are required to satisfy the real-time requirements of robotics applications and knowing the throughput of a system is useful to determine performance when sending large amounts of data between program.

We measured the performance of the system with no other activity on the interconnect. It takes only 750 μsec to send a 2 byte message and for a 1000 byte message the latency is only 3.3 msec. In the two byte case, the actual time the message spent going across the bus is only 2 μsec. For a 1024 byte message, the time spent on the bus is only 100 μsec out of the total time of 3.3 msec. This data shows that over 95% of the cost of communications can be attributed to the Meglos software and less than 5% to the hardware. This demonstrates why it is necessary to evaluate performance at the user level to obtain meaningful results.

Because writes do not return until an acknowledgement is received from the remote end, their times include all overhead for communication protocols and context switching. The latency for access to the S/NET depends on the number of other processors already waiting to send a message and the length of the messages that they send. Because the S/NET interconnect provides first-come first-served access to its bus, it provides a guaranteed maximum response time. In the extreme worst case, when each other processor is sending a long (1000 byte) message, the maximum hardware latency is 4.4 msec. So, the worst case transmission time for a short message is 5.2 msec and a for a long message is 7.7 msec.

The second measure, throughput, is obtained by dividing the number of bytes in a message by the latency for that size message. For messages longer than 1000 bytes, the throughput exceeds 300 kbytes/sec. Even at such high rates, only 3% of the S/NET's available bandwidth is used. Thus, the S/NET is capable of supporting up to thirty pairs of processors, communicating at their maximum rates, before the S/NET's bandwidth is used up.

When comparing the measurements reported here with the performance of other systems, one should be aware that some system designers only provide speed measurements of the hardware interconnect or measurements made at the kernel level. Such measurements usually indicate significantly better performance than can actually be obtained by user programs and are not useful for determining how well an application would run on the system. All our measurements were made between user programs and were done with a high-precision timer located on the individual processor boards.

The txstat command has an option for displaying the status of all the channels currently in use. For the example, the output from txstat with the channel information displayed is shown in Figure 4.

The figure shows the five channels used by MIMIC. A channel can be in any one of three states OPENING, OPEN, and CLOSING. The first line indicates that the channel vision is OPEN and connects trakleds executing on machine 2 to mimic executing on machine 1. This channel is used to transmit the LED image coordinates to mimic so it can determine the motion of the circuit board. The channel mimic_update_message is OPEN and connects mimic to mimicarm. This channel is used bidirectionally, to send joint rotations to mimicarm and to send the current robot position back to mimic. The channel mimic_clock_message is interesting in that it connects the mimicarm executing on machine 5 to itself. The use of this channel will be discussed in the forthcoming section on subprocesses. The last channel is OPENING because only one program has issued a copen for the channel. These channels are used to change internal parameters of the robot controller from a terminal.

V. Devices

Since each of the single board computer systems has its own Multibus cage, specialized devices such as robot arms and sensors may be connected to the Multibus and be used by the processor. Meglos allows programs running on a processor to use these devices and respond to their interrupts. All Meglos programs run with the processor in user mode. Most other operating systems require that programs that control devices to run in

```
MACHINE   MEMSIZE   MEMFREE   USER     PID    MEMSIZE   PRIORITY   PROGRAM
      1     1152K     1056K
                                rdg    3733        96K       128   mimic
      2     1152K     1056K   (exclusive use: rdg)
                                rdg    3731        96K       128   trakleds
      3     1152K     1152K
      4     1152K     1152K
      5     1152K     1056K   (exclusive use: rdg)
                                rdg    3732        96K       128   mimicarm

MACHNO    PID   LCHNO     STATE     MACHNO    PID   LCHNO   CHANNEL-NAME
     1   3733     2       OPEN         2     3731     2     vision
     5   3732     2       OPEN         1     3731     2     mimic_update_message
     5   3732     4       OPEN         5     3732     5     mimic_clock_message
     5   3732     7      OPENING                            mimicarm_info_external
```

Figure 4. txstat *output indicating channel usage.*

supervisor mode. This is potentially dangerous because a program in supervisor mode can overwrite the kernel or other programs running on a processor.

Every program running on Meglos may access Multibus I/O space. Programs access I/O registers of devices on the Multibus by reading or writing the address that corresponds to the register. Meglos allows programs to respond to device interrupts by calling a C language subroutine in the user's address space when an interrupt occurs. The name of the subroutine to call and the interrupt number are specified in the `onintrpt` call. `Onintrpt` also requires a user-written subroutine that contains code to disable the device. Meglos calls this subroutine when the program terminates either normally, or due to an exception such as a bus error. For applications using devices such as robot arms that can cause physical damage without proper program control, such a routine is required to stop the device when the program terminates abnormally because of a programming error.

There are certain limitations placed on interrupt service routines because they are not schedulable like subprocesses. This makes it impossible for an interrupt service routine to read or write on a channel or perform any UNIX system calls. A typical interrupt service routine performs the function needed by the interrupting device and then starts up a subprocess to handle any communications that may be required.

Programs that access hardware devices sometimes need a mechanism to prevent the interrupt service routine (specified in `onintrpt`) from executing during some sections of code. A program may delay the servicing of a hardware interrupt by raising the processor interrupt priority with one of the `spl` calls. The call `spl`*n* sets the interrupt priority level to *n*. It is the programmer's responsibility to return to priority level 0 with the `spl0` call at the end of the critical section of code. By default, interrupt service routines run at the hardware priority level of the interrupting device.

Mimicarm, the robot controller used by MIMIC, requires that joint angle updates occur at fixed intervals of 28 msecs. The interface board generates interrupts at this rate from the robot controller's internal 28 msec clock. This interrupt is tied to one of the Multibus's eight interrupt lines and the `onintrpt` call binds mimicarm's service routine to this interrupt. This routine need only reset the interrupt and resume the mimicarm update subprocess. This will be discussed in more detail in the next section.

Like any other device, Meglos allows programs to directly access the device registers for the S/NET interconnect. This allows programmers and language designers to experiment with communications primitives other than those provided directly by Meglos. For instance, the communications operations of the Linda language[14] and a queue-based communications scheme for the CEMU circuit simulator[15] are implemented in this fashion. Programs performing their own communications are required to send messages with a header that allows the kernel to distinguish these messages from those sent on channels. A variant of the `onintrpt` call is used to specify a handler to be called when these messages arrive at a processor. Though this general communications facility is available, all robotics applications implemented to date have found the channel mechanism to be sufficient for their needs.

The single board computers include a timer that can be set to interrupt at fixed intervals. The `maketimer` call makes use of this timer to call a C routine at a specified time interval. The timer interrupts at interrupt priority level 6 and the specified C routine runs at this priority level.

VI. Subprocesses

The portions of robotics applications that control devices often have strict real-time requirements. To simplify the design of such programs, Meglos allows a program to be subdivided into subprocesses with user-specified execution priorities. Subprocesses are a part of a user's process that may execute asynchronously with the rest of that process. Each subprocess is independently schedulable; the Meglos scheduler knows and understands that a process can contain more than one thread of execution. This can be viewed as a form of multitasking within a process. All subprocesses within a given process share the same address space and each subprocess has its own stack for storing the machine state information and its local variables.

Meglos includes a preemptive scheduler that allows execution priorities to be specified for each subprocess of a process. Since high priority subprocesses are always run in preference to those with lower priorities, strict real-time response may be assured by coding the portion of an application with real-time requirements as a high priority subprocess. The scheduler is referred to as *preemptive* because when a higher priority process becomes runnable (say, as a result of its input or output completing) while a lower priority process is running, the higher priority process preempts the execution of the lower priority process. The lower priority process will be restarted when the higher priority one terminates, blocks waiting for input or output to complete, or by issuing a `suspend` call or a `P` operation on a semaphore, Meglos does not provide time slicing between processes of the same priority, so that a subprocess that computes forever would prevent other subprocesses from executing.

Each process in Meglos consists of one or more subprocesses. All subprocesses of a process execute in the same address space, so that global variables are shared between subprocesses, Each subprocess has its own local variable stack. A process initially consists of a single subprocess and additional subprocesses may be created with the `creat_sbp` call. The name of a C subroutine that contains the code for this subprocess as well as the arguments describing the environment for the subprocess are specified in this call. `Creat_sbp` returns a subprocess number that is used to refer to the subprocess that was created. A process can temporarily stop its own execution by calling `suspend` and can restart another suspended subprocess by calling `resume` with the number of the subprocess that is to be resumed.

A subprocess is terminated when its procedure returns or can be terminated by another subprocess with `kill_sbp`. The main subprocess (i.e., the first subprocess of a process with the procedure name of `main`) should not return and no subprocess should call `exit` while other subprocesses are running. If they do, all subprocesses of the process are immediately terminated.

A subprocess that modifies a data structure shared among many subprocesses may have to prevent the other subprocesses from accessing that data structure while it is being modified. Like many other systems, Meglos provides semaphores[16] to allow programs to implement such controlled access.

For applications that consist of many subprocesses or processes on the same processor, it is important that the overhead incurred by Meglos for switching between tasks is small. The time for switching between two subprocesses within the same process is 200 μsec and the time for switching between two different processes is 250 μsec. The interrupt response time, the time that elapses between when an interrupt occurs and the user routine that services that interrupt starts to run, was measured at 50 μsec.

Two of the programs that are part of MIMIC, trakleds and mimic each consist of a single subprocess. The robot controller program, mimicarm, consists of three subprocesses and an interrupt service routine. One subprocess is used to provide synchronization and the second actually controls the robot arm. The third subprocess uses the mimicarm_info_external channel to connects to a program that reads from the user's terminal to allow changes to the internal controller parameters while the application is running. See Figure 5.

Every 28 msecs, the synchronization subprocess is resumed by the interrupt service routine and it sends a message across the mimic_clock_message channel and suspends itself awaiting another interrupt. The other subprocess is connected to both the mimic_clock_message channel and the mimic_update_message channel. Since messages arrive asynchronously on both channels, it uses `readn` to accept data from either channel.

A typical sequence of events is as follows: A joint angle update message arrives from mimic on the mimic_update_message channel. The subprocess stores the new joint angle rotation values, reads the current positions of each joint from the robot controller and sends these values back to mimic across mimic_update_message. It then blocks on the `readn` awaiting either another update message or a synchronizing message. When a synchronizing message arrives, the subprocess passes the joint angle rotation values it saved previously to the robot controller and again executes the `readn`. The system continues to function this way until the application is terminated by the operator.

VII. Debugging

The message-passing style of communications used in Meglos encourages programmers to break up their applications into functionally distinct modules with well defined communications interfaces. This often allows each program of a robotics application, like the trakleds, mimic, and mimicarm programs, to be written and debugged as separate modules in isolation from each other.

To help debug such programs, Meglos includes a symbolic debugger for C programs that allows programs to be run in an environment that can be monitored and controlled. Breakpoints can be set, either at a specified C statement, or at a machine instruction. Both C language variables and arbitrary memory locations may be displayed and modified. The debugger allows the programmer to access the program at both the C and

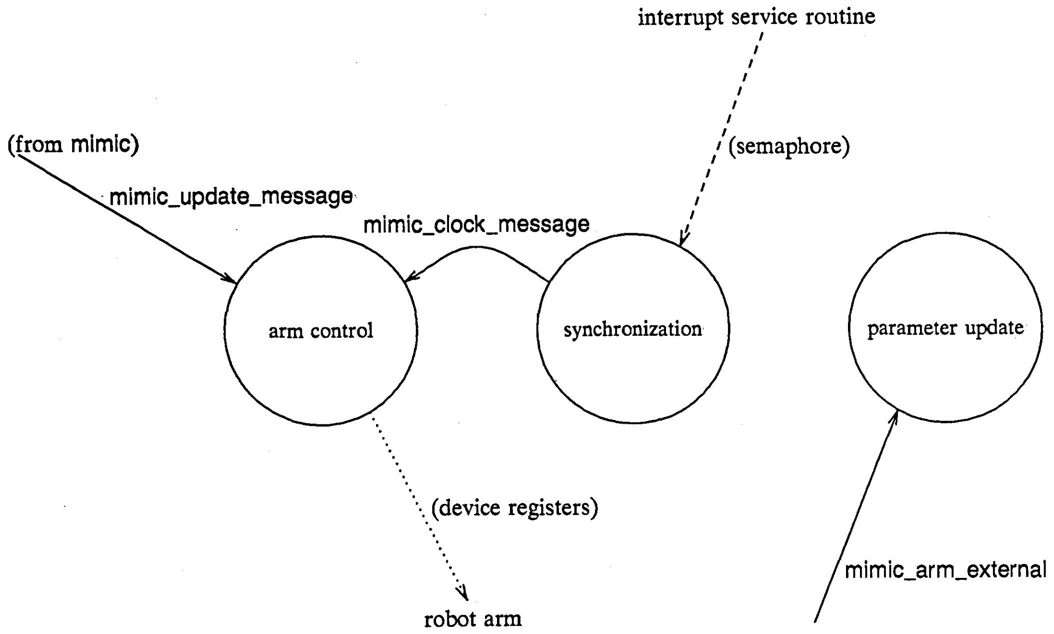


Figure 5. *Subprocesses of* mimicarm.

machine language levels and provides for easy translation between the two. The debugger's user interface is identical to that of the sdb debugger[17] which is part of the UNIX System.

If a program incurs an exception such as a bus error, Meglos aborts the program and produces a file on the host system with the contents of the program's memory and registers. The debugger is used to examine these images of aborted programs. It is also possible to abort a program at any time from the UNIX system, disable devices connected to that processor, and obtain a memory image by hitting a key on the keyboard. This is crucial when debugging robotics applications because it allows the operator to stop a robot before it crashes through a wall.

When it is necessary to simultaneously debug programs running on many processors at the same time, the windowing capabilities of a workstation or a terminal like the Teletype 5620[18] may be used. Each program of the application can be run with the debugger in a separate window, so that interactions between programs may be viewed.

The convenience of having full access to the UNIX system and good debugging facilities can be used by programs that have real-time requirements. Programs incur no extra overhead for accessing the host UNIX system except for the duration of the system call. While debugging, programs run at their normal speed until they are stopped at a breakpoint. Thus, the interactive debugger can be used to test most portions of real-time programs. For debugging time-critical sections of code, one can send debugging output to a file. If that is still too slow, debugging output can be written into memory and examined at a later time.

Far more sophisticated debugging tools that we provide would be useful in many situations. For example, a common symptom of a program bug is for an application to deadlock with all of the program of an application waiting for channel input. Examining core dumps of the programs provides little help, because they do not show how the processes got into the deadlocked state. A tool that should help in such situations is one which monitors communications and retains the last several messages sent on each channel, allowing them to be viewed by the programmer.

VIII. Conclusions

We have illustrated the user interface provided by the Meglos operating system. A real-time robot control application example was used to show some of the primitives and run-time information commands available to a user of the system.

Several prototype Meglos systems have been built and some are used to support multiprocessor robotics applications in our Robotics Research Department. The users' experiences demonstrate that Meglos is a good environment for developing and running robotics applications. The close coupling of the host UNIX System with the single-board computers allows real-time programs to access all the facilities provided by the UNIX System. The job of programming and testing an application is simplified by the availability of a source language debugger and by convenient access to the UNIX system. The primitives provided by Meglos for communications between programs, and for multitasking within a processor have proven to be easy to use and to perform adequately for robotics applications.

In the past, developers of robotics applications that require multiple processors have found it necessary to build specialized hardware and operating system software for each of their applications. We have shown that a single hardware and software system can be built to accommodate a wide range of applications, allowing robotics applications developers to concentrate their efforts on hardware and software designs specific to their applications.

REFERENCES

1. Klein, C. A., and Wahawisan, W., "Use of a Multiprocessor for Control of a Robotics System," *Int. J. Robotics Res.* **1**, 2, 1982, 45-59.

2. Gaglianello, R. D. and Katseff, H. P., "Meglos: An Operating System for a Multiprocessor Environment," *Proceedings of the Fifth International Conference on Distributed Computing Systems,* Denver, May, 1985, 35-42.

3. Gaglianello, R. D. and Katseff, H. P., "A Distributed Computing Environment for Robotics," *Proceedings of the 1986 International Conference on Robotics and Automation,* San Francisco, April, 1986, 1890-1896.

4. Ahuja, S. R., "S/NET: A High Speed Interconnect for Multiple Computers," *IEEE J on Selected Areas in Communications,* **SAC-1**, 5, November, 1983.

5. London, T. B., Ahuja, S. R., and Katseff, H. P., "Performance of an Interconnected Microprocessor System Designed for Fast User-level Communications," *IFIP WG 10.3 Workshop on Hardware Supported Implementation of Concurrent Languages in Distributed Systems,* University of Bristol, U. K., March, 1984, 125-134.

6. Pacific Microcomputers Inc., *PM68K User's Manual,* San Diego, California, 1982.

7. SKY Computers Inc., *Fast Floating Point Processor Integration Manual,* DOC #SE-IM-84-02.0, Lowell, Massachusetts, 1984.

8. Jarvis, J. F., Private communication.

9. Ganapathy, S., "Teaching Robots by Hand Movements," in preparation.

10. Seitz, C. L., "The Cosmic Cube," *Comm. ACM* **28**, 1, January, 1985, 22-33.

11. Ritchie, D. M., and Thompson, K., "The UNIX Time-Sharing System," *Bell Syst Tech J* **57**, 6, 2, July, 1978.

12. Gaglianello, R. D. and Katseff, H. P., "Communications in Meglos," *Software Practice and Experience,* to appear.

13. Leffler, S. J., et. al., "A 4.2BSD Interprocess Communication Primer," Unpublished Draft, University of California, Berkeley, California, 1983.

14. Carriero, N., and Gerlernter, D., "The S/Net's Linda Kernel," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles,* Orcas Island, Washington, U.S.A., December, 1985, 54-85.

15. Ackland, B. D., et. al., "MOS Timing Simulation on a Message Based Multiprocessor," *1986 IEEE International Conference on Computer Design: VLSI in Computers and Processors,* Port Chester, New York, October, 1986, to appear.

16. Shaw, Alan C., *The Logical Design of Operating Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1974.

17. Katseff, H. P., "Sdb: A Symbolic Debugger—Version 3.0," part of documentation for the UNIX^{TM} Operating System.

18. Pike, R., "The Blit: A Multiplexed Graphics Terminal," *AT&T Bell Laboratories Tech J* **63**, 8, 2, July, 1984, 1607-1632.

A Robot Force and Motion Server

Hong Zhang and Richard P. Paul

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

ABSTRACT

The design and implementation of a distributed robot manipulator controller based on a concurrent architecture is introduced in this paper. We consider the manipulator as a force and motion server (RFMS) to a robot system executing force and motion commands issued by the robot system coordinator. In the server, computations are distributed to a number of processors in order to minimize the time delay in responding to input requests. The "C" programming language is used for both system programming and the user interface.

1. Introduction

A robot manipulator might be considered to act as a force and motion server for a robot [1]. The robot, interpreting sensor information in terms of a task model and a task plan, issues requests to the manipulator to exert forces or move objects. The motion of the manipulator is modified by sensor feedback.

Sensors determine the state of a manipulation task in terms of their own coordinate frames. If this information is to be used to control the manipulator, it must be transformed into a common coordinate task frame where given constraints may be applied and information from various sensors integrated to form a best estimate of the task state. This information must in turn be transformed into the manipulator joint coordinates where control of the manipulator is exercised. The time delays involved in these transformations, the time from when a sensor observation is made until the time when the signal to the manipulator actuators is changed in response to the sensor observation, must not be of the same order of magnitude as the natural response time of the manipulator itself; otherwise, system instability will result. It must be either much slower, such as in welding sensor feedback, or much faster, such as in force or contact feedback. We are interested in working in the latter domain to provide a manipulator controller in which the response of the system is limited only by the manipulator itself and not by the control computer.

In order to minimize the time delay between a change in some Cartesian coordinate frame and a response at the actuator level, we perform as many calculations concurrently as possible. We do this by separating the kinematic and dynamic aspects of the computation. There is little point in utilizing a processor to perform manipulator joint control, another to perform input/output, another to perform kinematic transformations, and another to convert from accelerations to joint torques, as these processes must then be performed one after the other. This increases the rate at which the calculations may be performed but does nothing to minimize the processing time delay. In our design, we use one processor for each joint of the manipulator and perform as many operations concurrently as possible to reduce the computation time delay. In order to separate the kinematic and dynamic aspects of the task, we compute manipulator configuration dependent parameters at a rate related to change in manipulator configuration and then make use of these parameters in the control of the manipulator at the much higher control sample rate. For example, we compute the joint inertias as a kinematic process in background to simplify the conversion of joint accelerations to joint torques which must be performed at the control sample rate. This is to be contrasted to the Newton-Euler approach in which link accelerations are first computed recursively from the base out to the end effector and then torques are computed, once again recursively from the end effector back to the base, resulting in a pipelined process in which the rate of calculation may be increased but the time delay in performing the calculation is not reduced. Numerically, the worst case delay in the Newton-Euler approach corresponds to some 2000 arithmetic operations.

We are also judicious in determining those calculations that we will perform exactly, those we will approximate, and those we will ignore. We do not calculate, or take into account, centripetal acceleration or Coriolis forces as these occur only at high speed, since we are primarily interested in

This work is partially supported by National Science Foundation under Grand No. ECS-8411879 and by Intel Corporation. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of supporters.

the performance of the manipulator at the end of motions when the speed is low. In most cases where accurate path following at high speed is important these forces may be pre-calculated or simply learned. We make use of as much knowledge as possible; we do not try to learn the dynamics of the manipulator, as we have symbolic equations, but we do learn the mass of an object being carried. Finally we are interested in performance and not just in control formalism elegance.

We have tried to use commercially available components as much as possible in our design. Intel 86/30 single board computers are used to perform the bulk of the processing. Each 86/30 is provided with its own input/output interface to the joint in order to minimize time delays. Calculations are performed in floating point, since 16 bit integer does not have sufficient precision for manipulator control and the resulting programs are virtually unreadable. We program in "C" to facilitate program understanding and modification. While we are interested in performance, we are also interested in cost and believe that the system we are developing is economically viable for industrial robot control.

2. Background

The control of a robot manipulator deals with the relationships among objects and between the manipulator and the objects. A robot manipulator task can be defined in terms of relationships in positions and orientations from one coordinator frame to the next. These relationships may in turn be described with homogeneous transformations [4] where the operation of matrix multiplication corresponds to the composition of coordinate frames.

The robot manipulator must be able to position its end effector arbitrarily in its work space. The Cartesian position and orientation of the manipulator, commonly referred to as $\mathbf{T}_6$, is obtained as the composition of transformations describing each position in a task. Given a manipulator task as successive Cartesian positions, the control system of the robot manipulator must convert them to joint coordinates where a robot manipulator is actuated. The desired joint positions can then be used to determine joint errors to produce joint correction torques.

If force control is required, the joint torques or forces must be computed by a different method. This in general requires the computation of the Jacobian matrix and inverse Jacobian matrix corresponding to the current manipulator position. In control of a manipulator the dynamics of each joint are taken into account by computing the link gravity loading and the effective and coupling joint inertias.

Feedback from sensors is achieved by modifying an appropriate coordinate frame in the composition of transformations specifying $\mathbf{T}_6$. The sensors run concurrently with the manipulator on their own processors. The information communicated is purely geometric.

3. Classes of Processes

The computations performed by the RFMS can be classified into dynamic processes, kinematic processes, and static processes, based on their real time constraints. Dynamic processes must be performed at a rate considerably higher than the natural frequency response of the manipulator. A kinematic process is a function of the position of a manipulator; its rate of computation is dependent on the change of configuration. The execution of the kinematic process affects the control accuracy but not the stability. A static process is time independent; failure to execute a static task delays the execution of the task, as the manipulator will come to rest and wait for its completion.

In control system of a robot manipulator, the dynamic processes control the joints. These processes are performed at a sufficiently high rate to provide for stable, noise-free control of the manipulator. This rate is of the order of 250 hertz for a manipulator of approximately one meter reach. As the control rate is reduced, the sophistication of the control algorithm must be increased.

The kinematic processes in a RFMS correspond to such tasks as computing the Jacobian matrices and link inertias. When the manipulator is at rest, or when performing fine motions these processes do not need to be computed at all. The static processes, very much like the ones in a time-sharing multi-user computer system, are related to house-keeping tasks such as user interface or file creation where there is virtually no time constraint. In a control system, various processes coexist and priority of service is given in the order of dynamic, kinematic and static processes.

4. Process Definition

In general, the following processes must exist in a manipulator control system: a set-point process for position control, a force control process for joint torque generation, and a supervisor process for system coordination, although implementation details may vary from system to system and additional processes may be required. In our particular case, a communication process is also needed to communication to the robot coordinator. The tasks all these processes perform are specified in the following sections.

4.1. Set-point Process

This process generates a set of desired joint positions or set-points every servo period. Neighboring set-points are thus separated in time by one sample period, Δt. A sequence of set-points make up a position trajectory. This trajectory must meet certain continuity conditions to assure the stability of the system. Different methods exist for such a process, but we use the algorithm for trajectory generation described in [2]. This method is based on the homogeneous transformation representation of the relationships between coordinator frames and allows two fundamental modes of motion, *Cartesian* and *joint*. The amount of computation in each sample period depends on the location of the trajectory and the mode of motion. The worst case occurs when the trajectory is specified in Cartesian mode and a transition to another Cartesian motion segment needs to be computed.

A position in the robot work space is defined by a position equation that, in its simplest form, equates the $\mathbf{T}_6$ to another position as:

$$\mathbf{T}_6 = \mathbf{P} \tag{1}$$

In general more structure is needed, for example:

$$\mathbf{Z}\,\mathbf{T}_6\,\mathbf{Tool} = \mathbf{C}\,\mathbf{P}\,\mathbf{G} \tag{2}$$

which means a tool is attached to the end of last link, the base of the manipulator is **Z** from the world coordinator frame, the grasp position **G** is defined with respect to a position **P**, which is again defined relative to a reference coordinate frame **C**. In general any of these transforms can be functions of time.

Motion is defined not only by the end points but also by the manner in which it is achieved. One can specify such motion parameters as the segment time, velocity, mode of motion, etc. A mode structure, **M**, describes these parameters in detail.

The set-point process computes the set-point in two stages. First it solves for $\mathbf{T}_6$ from the current position equation $\mathbf{P}_i$ and possibly the next position equation $\mathbf{P}_{i+1}$ if transition is necessary. This requires matrix operations such as matrix inversion and multiplication. Obviously, the more complex the position equation, the more expensive this process is computationally. Eq(3) functionally defines this first stage:

$$\mathbf{T}_6 = \Gamma(\mathbf{P}_i, \mathbf{P}_{i+1}, \mathbf{M}, t) \tag{3}$$

The second stage of the set-point process is the inverse kinematics that solve for the joint positions from the $\mathbf{T}_6$. When the manipulator is simple, this can be performed symbolically, i.e., closed-form solutions exist to express the joint positions as a function of the $\mathbf{T}_6$.

$$\boldsymbol{\theta}_d = \Lambda(\mathbf{T}_6, \mathbf{M}, t) \tag{4}$$

Typically an iteration of this process requires between five to ten milliseconds on a VAX 11/780 minicomputer.

4.2. Force Control Process

This is a dynamic process responsible for computing joint torques and then driving the joints. Given the desired joint positions, $\boldsymbol{\theta}_d$, there are a number of methods of generating joint torques. A general form for torque at the *i*th joint is:

$$\tau_i = \mathbf{D}(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}, \ddot{\boldsymbol{\theta}}, \mathbf{D}_i, \mathbf{D}_{ij}, \mathbf{D}_{ijk}) + \mathbf{F}(\mathbf{f}_c, \mathbf{J}) + \mathbf{R}(\boldsymbol{\theta}_d, \mathbf{S}, \mathbf{J}) \tag{5}$$

where

D dynamics compensation due to the gravity loading $\mathbf{D}_i$, inertial acceleration $\mathbf{D}_{ij}$, and Coriolis and centripetal forces $\mathbf{D}_{ijk}$,

F joint bias force due to the Cartesian bias force $\mathbf{f}_c$, and

R the reaction torque due to joint errors $d\boldsymbol{\theta}$ with the Cartesian compliance specification S taken into consideration, where S is a 6x6 diagonal selection matrix with 1's for constrained directions and 0's for unconstrainted directions.

Eq (5) is based on [3] and reflects the general approach of all joint based force control methods such as the stiffness method and hybrid method. Note here that all the input parameters to Eq(5) are assumed to be available from the kinematic processes. This process generates joint torques in response to the joint position errors, to the compliance specification of the manipulator task, and to the requirement of dynamics compensation. In the case when there is no contact between the manipulator and the environment, the torques are simply generated from the joint position errors and dynamics compensation.

4.3. Background Process

Parameters such as the Jacobian matrix and its inverse and the dynamic constants in the dynamic equations are functions of the manipulator kinematics and are computed in a kinematic process in the background.

4.4. Communication Process

A process is provided to communicate between the force and motion server nd the robot coordinator. The robot coordinator issues messages to the RFMS and may request information such as current $\mathbf{T}_6$. The messages sent by the coordinator describe the task model and defines the task plan in terms of the following data structures: *transform* definitions, *position* definitions, *mode* definitions, and *action request*. In addition, special messages also are defined to deal with situations such as initialization and emergency. All the messages are handled by the communication process.

In *transform* definition, one can define a constant transform or a variable transform that is dependent on external signals such as sensors. If the transform is variable, a redefinition by the coordinator would result in a corresponding redefinition of its counterpart in the server. This mechanism enables a user to modify manipulator motions. A transform symbol table exists on the server and this communication process stores the transforms in this symbol table.

In a message for a *position equation*, there are two ordered lists of transforms corresponding to the left and right hand sides defining a position in the robot work space. The transforms used must have been defined prior to the position definition and, therefore, they must exist already in the transform table of the server. Using the transform symbol table, we link the transform equation on the server into a ring structure for easy operation[4] and then enter it into the position symbol table.

A *mode* structure specifies how each motion is to be made, with parameters such as segment time, acceleration time, velocity, compliance specifications, etc. One may define a number of modes in a program and associate motion segments with modes. It may sometimes be desirable to execute a motion one way in the beginning of a task and then execute the same motion later in the task with another mode specification. The definition of mode structures and their association with a motion make the programming of tasks simple and clear. When a mode message is received, the communication process stores it in a mode symbol table for later reference.

Another type of messages, *action requests*, relies on the defined data structures in the first three categories to initiate actions for the manipulator. An action request contains two pointers: one to a destination position with the current position being the default initial position, and the second pointer to a mode structure specifying how the motion is to be made. Upon receipt of an action request, the communication process stores the request in an action queue. The two pointers in an action request are identifiers to the position table and mode table. The use of action queue guarantees program synchronization.

5. Implementation

The RFMS has been constructed using Intel single board computers (SBC) to control a Unimation PUMA 260 robot manipulator. The system is illustrated in Figure 1. It contains a number of processors including one for each joint, one for the gripper, one as the supervisor, one communicating with the robot coordinator, one as the math processor, and one data channel processor. In addition, special hardware has been designed for the low level joint interface.

The system is tightly coupled by the Multibus system bus. The Multibus interface is a general purpose system bus structure providing for communication between system components. Memory on one board can be accessed by another through the Multibus; eight interrupt signals can be used to direct actions. To minimize the bus contention, Multibus usage by boards other than the supervisor is kept at the minimum in the system. The RFMS does not use an operating system but is driven by interrupts and handshaking operations.

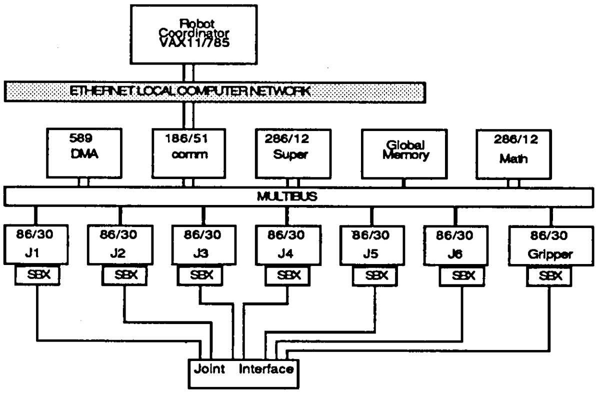


Figure 1. System Organization

5.1. Ethernet Communication

In this implementation the robot coordinator runs on a VAX 11/785 under Unix in a time-sharing environment. An Intel iSBC 186/51 communication controller in the RFMS facilitates the communication between the robot coordinator and the RFMS through the Ethernet local network. It is through this means that different agents such as sensors and RFMS in the robot system exchange information to update the world model. The speed of this communication is at the millisecond level, thus allowing the closing of a feedback loop through the Ethernet.

The communication between the coordinator and the RFMS takes place in the form of messages. The messages sent from the coordinator are first received by the communication controller, which stores the messages sequentially in a message list. Each entry of the message list contains a message identifier or index, a message type as described in Section 4.4, a flag for handshaking with the process reading the messages, and the content of the message. The communication controller appends the messages to the list to be processed by the supervisor. Depending upon the types of messages, this process can be either kinematic when handling sensor modifications or static when defining a task.

5.2. Supervisor

The supervisor runs on an Intel iSBC 86/30 with an 8087 floating point co-processor. The iSBC 86/30 contains 128K of dual-port memory accessible to both the local CPU and any other SBC on the Multibus, nine levels of interrupt control, two programmable timers, and serial and parallel I/O interfaces. The 8087 numeric co-processor executes floating point instructions at eight MHz and its instruction set provides for both arithmetic and trigonometric functions. The task of the supervisor is to maintain the global variables and control the system timing. Two separate processes run in the supervisor, one that reads the messages from the communication controller in background and one that coordinates the set-point generation in real time.

The background process reads the messages stored in the iSBC 186/51 and creates the internal data structures in terms of symbol tables and a motion queue as described in Section 4.4. The numeric values of the transformations elements are stored in the math processor memory where the matrix operations are performed.

When a position equation is processed, the two transformation lists representing the two sides of a position equation are used to create the ring structure, and a pointer to the structure is stored in a position symbol table. A mode record is treated similarly in a mode symbol table. Finally, an action request is queued in the action queue after its two pointers to a position equation and to a mode record are located in the respective symbol tables. The memory organization is summerized in Figure 2.

The real-time process on the supervisor initiates the computation of the next set-point upon a real-time clock interrupt from the programmed timer every sample period. The supervisor requests the math processor to compute the Cartesian set-point, i.e., the desired $\mathbf{T_6}$, and then requests the data channel processor to communicate $\mathbf{T_6}$ to the joint processors to compute the joint coordinates. The specific computation performed during each period depends upon the *state* of the manipulator or its trajectory. This process must finish before the next interrupt comes when another set-point must to be generated.

The most time-consuming computation in generating Cartesian set-point is the matrix multiplication required to compute $\mathbf{T_6}$. The math processor performs this operation. The math processor will eventually be replaced by a matrix multiplier device [5]. In each sample period when the matrix multiplications are needed, the supervisor simply generates a pointer list to those matrices to be multiplied, interrupts the math processor, and then continues with other operations. Upon completion of matrix multiplications, the math process interrupts the supervisor.

5.3. Joint Processors

Each joint is equipped with an iSBC 86/30 and an 8087 numeric processor. 64 kilobytes of its memory is configured global for interprocess communication. Joint processes run in parallel to control individual joints as described by Eq(4) and Eq(5). Working as slave processors to the supervisor, each of the joint processors computes its joint trajectory. The only process on a joint processor is real-time and interrupt driven. At the beginning of a sample period, it reads the current desired Cartesian position computed by the supervisor and solves for the joint position. Since the *i*th joint solution requires sines and cosines of the prior *i*-1 joints in general and this would cause considerable time delay if joints wait for solutions, we make use of the sines and cosines of joint variables computed in the last sample period so that all joints start computing simultaneously. During a transition, however, a joint process computes the coefficients of the transition polynomial and obtains solution by evaluating the polynomial. The *state* variable of the supervisor dictates the action of the joint processes.

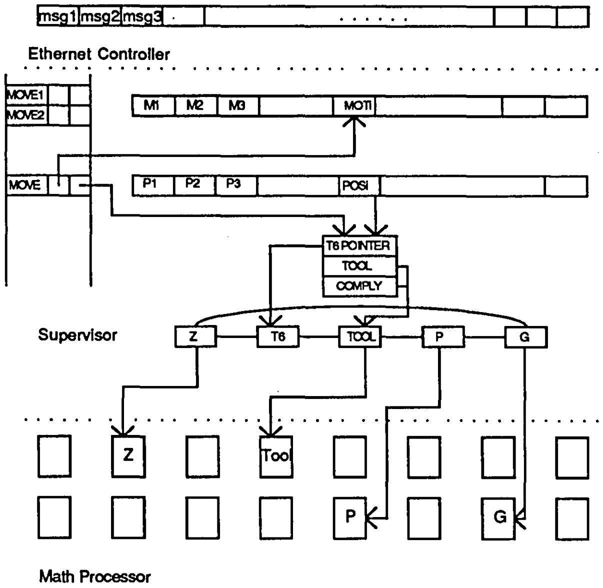


Figure 2. Memory Organization

The force control process is executed next to compute joint torque as in Eq(5). Information such as the Jacobian matrix and dynamics equation coefficients are broadcast to the joints when they are updated by the kinematic processes while the information such as errors and accelerations of other joints is collected from and distributed back to the joints as required in every sample period, all by the supervisor.

The low level interface to the joint is achieved through a specially designed hardware, iSBX multimodule board, attached to the standard SBX connector on each host 86/30. It provides the joint encoder interface and conversions between analog and digital signals. The board employs two digital to analog converters, two analog to digital converters, and an incremental encoder circuit. The first DAC outputs motor current, while the second allows us to specify a force set point to the joint. The ADCs read back the joint motor current and joint velocity, respectively. The incremental encoder tracks the position of the joint by observing the waveforms generated by the joint encoder. At the end of each sample period, the computed torque is converted to an equivalent joint current value before it is sent to the amplifier circuits to drive the joint motors.

5.4. Math Process

The math process is based on a iSBC 286/12 with an 80287 floating point co-processor. The iSBC 286/12 has 128k on board dual-port memory, a programmable interrupt controller, programmable timers, Multibus interface, and parallel and serial I/O interface, and the numeric co-processor 80287 similar to 8087 in its function. The process performs dynamics and Jacobian updates in background and, when needed, is interrupted to perform matrix operations for the set-point process.

5.5. Data Channel Processor

A DMA board iSBC 589 is added to the system to speed up the data transfer operations. Because the supervisor and the joint processors exchange information frequently in their real-time processes, this device considerably reduces the time for global memory read/write. At the time of initialization, destination and source addresses for different data transfers are stored in the parameter blocks of the DMA board. For each operation, the DMA simply needs to be written a wake-up byte and given the address of one of the parameter blocks.

5.6. Packaging

The system is housed in two connected iSBC 608 and iSBC 618 Cardcages with two iSBC 640 power supplies to provide a total of four double width and twelve single width SBC slots on the same Multibus for the system.

5.7. Performance Evaluation

The RFMS as described above has been evaluated for its real-time performance. Experiments have shown that the RFMS achieves a sufficiently high rate to update the Cartesian set-points while employing sophisticated control strategies for such applications as sensor-driven motions and force control. In order for the joint processes to evaluate joint set-points without waiting for the supervisor to finish computing T_6, th supervisor process pipelines its results to the joint processes, as is illustrated in Figure 3, where τ represents the sampling period, joints represents one of joint processes, and kinematic represents the background process computed on the math processor. By overlapping the supervisor and joint processes, we can shorten the minimum sampling period to obtain a servo rate over 250 Hz without performing any joint set-point interpolations. This servo rate is almost an order of magnitude improvement over the manipulator controllers that are being used today to control six degree of freedom manipulators.

We have achieved a throughput of 4 Kbytes per second data transmission rate through the Ethernet between the coordinator on a VAX 11/785 under normal working conditions and the RFMS. At this speed, we are able to send more than 100 transformations per second from the coordinator to the RFMS, as new observations are acquired by sensors, thus achieving real-time sensor feedback closed around the network.

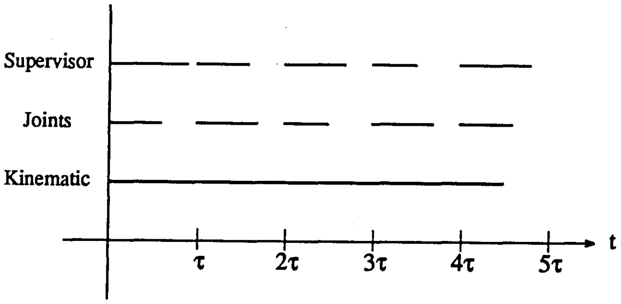


Figure 3. Process Scheduling

When homogeneous transformations are used to describe positions, the servo rate is limited by the complexity of the position equation associated with the current motion, since the more complex the position equation, the more matrix operations are required, and the longer sampling period would result. To eliminate the dependency of the sampling period on the complexity of the position equation, we are exploring solutions such as using the matrix multiplier [5] or high speed floating point processor to perform the matrix operations.

6. Discussion and Summary

We have endeavored to design a robot manipulator controller with sufficient computational bandwidth and precision so that the manipulator performance rather than the controller limits the performance of the system. We have done this in an economical manner using off-the-shelf components as much as possible. All coding is in the "C" language to facilitate understanding and future modification of the system.

The system provides for both position and force control implementing the hybrid force control system proposed by Craig and Raibert. The system is not based on a programming language but is specified in terms of network message formats. An Ethernet interface is provided so that the system may be directly interfaced with many other sensors and robot coordinator in a very simple manner, and that the motion can be controlled by sensors in real time.

References

[1] Paul, R.P., *et, al*, ''A Robust, Distributed Sensor and Actuation Robot Control System", ISRR, France, October, 1985

[2] Paul, R.P., *et, al* ''Robot Motion Trajectory Specification and Generation", ISRR Proceedings , Japan 1984

[3] Zhang, H., *et, al*, ''Hybrid Control of a Robot Manipulator", IEEE Conference on Robotics and Automation, St. Louis, MO 1985

[4] Paul, R.P., ''Robot Manipulators: Mathematics, Programming, and Control", MIT Press 1981.

[5] Nash, J.G., ''A Systolic/Cellular Computer Architecture For Linear Algebraic Operations", IEEE International Conference on Robotics and Automation, St. Louis, MO, March, 1985

Software Engineering for Rule-based Systems

Robert J.K. Jacob
Judith N. Froscher

Naval Research Laboratory
Washington, D.C. 20375

Abstract. Current expert systems are typically difficult to change once they are built. The objective of the present study is to develop a design methodology, which will make a knowledge-based system easier to change, particularly by people other than its original developer. The basic approach for solving this problem is to divide the information in a knowledge base and attempt to reduce the amount of information that each single knowledge engineer must understand before he can make a change to the knowledge base. We thus divide the domain knowledge in an expert system into *groups* and then attempt to limit carefully and specify formally the flow of information between these groups, in order to localize the effects of typical changes within the groups.

As the commercial promise for expert system technology grows, the problem of ongoing maintenance and modification of knowledge bases is becoming a significant concern. The designs of typical current knowledge-based systems are *ad hoc*, one of a kind, and difficult to maintain. The information in the knowledge base is interconnected in such a way that changing one part of the knowledge base may have unpredictable effects on other parts.

This research attempts to develop a design methodology similar to those used in software engineering,[1,2] which will make a knowledge-based production system easier to change, particularly by people other than its original developer. We have chosen to concentrate on production systems because they are the most widely-used type of knowledge representation in expert systems, particularly among those existing systems large enough and mature enough to have experienced the types of maintenance problems we hope to alleviate. In the future, we will attempt to extend the approach to suit other, newer knowledge representations, such as frames and semantic nets, as large systems begin to be written using them.

This paper describes the approach we are taking to build maintainability into production systems. It introduces a programming methodology for developing production systems. It discusses our study of structure and connectivity in already existing knowledge bases. It then presents algorithms we have devised for separating the information in a knowledge base and results obtained with them. Finally, it discusses tools for supporting the methodology.

Methodology

The basic approach we have taken for building maintainability into an expert system is to divide the information in the knowledge base and attempt to reduce the amount of information that each single knowledge engineer must understand before he can make a change to the knowledge base. We thus divide the domain knowledge in an expert system into *groups* and then attempt to limit carefully and specify formally the flow of information between these groups, in order to localize the effects of typical changes within the groups.

Production systems comprise extensive domain knowledge, expressed as if-then rules, and a relatively simple inference mechanism or rule interpreter. The interpreter tests the values of the facts on the left-hand side of a rule; if the test succeeds, new values for facts are set according to the right-hand side of the rule. In the present approach, we divide these rules into separate groups. The guiding principle for grouping two rules together is: *If a change were made to one rule, to what extent would the other rule be affected?* In this study, a *fact* refers to that part of the data representation that, if changed by one rule, would affect another rule in some way; in a simple production system where the data are represented as sequences of attribute-value pairs, a fact corresponds to an attribute. The knowledge engineer building the system would group together rules that use or produce values for the same sets of facts. With this arrangement, a fact in the knowledge base can be characterized either as being generated and used by rules entirely within a single rule group or else as spanning two or more groups. The latter will prove critical to future changes to the knowledge base, since they are the "glue" that holds the groups together.

Whenever rules in one group use facts generated by rules in other groups, such facts will be specially flagged, so that the knowledge engineer will know that their values may have been set outside this group. More importantly, those facts produced by one group and used by rules in other groups must be flagged too.

For each such fact, the programmer of the group that produces the fact makes an assertion, comprising a brief summary of the information represented by that fact. This assertion is the only information about that fact that should be relied upon by the programmers of other groups that use the fact. It is not a formal specification of the information represented by the fact, but rather an informal summary of what the fact should "mean" to outside users.

Given this structure, a programmer who wants to make changes to the system would assume the responsibility of understanding thoroughly and preserving the correct workings of a single group of rules (but not the entire body of rules, as with conventional systems). He or she would be free to make changes to the rules in the group provided only that he preserves the validity of the assertions associated with any facts that are produced by his group and used by other groups. Similarly, whenever he used a fact that was produced by another group, he would rely only on the assertion provided for it by the programmer of the other group and not on any specific information about the fact that might be obtainable from examining the inner workings of the other group.

Following our methodology, the developer would first divide the rules into groups. This can be done manually or automatically, as described below. One approach is to apply one of the automatic grouping algorithms to the initial prototype expert system and use the resulting grouping to guide the organization and development of the final production version. Then, a software tool will characterize each fact as inter-group or intra-group, and flag the former. The developer of a rule group that produces inter-group facts then provides an assertion describing each such fact. That description is the only information about the fact that should be used in the development of any other groups containing rules that use the value of the fact.

Thus, the set of rules will be divided into groups, the inter-group facts used and produced by each group will be identified, and descriptions will be entered for those produced by each group. Figure 1 shows the language used to provide this information, using an excerpt from a simple example knowledge base.[5] The figure illustrates the syntax for describing rule groups; a larger system would look exactly the same, except that it would list more rules, facts, and groups.

To modify a group, the maintenance programmer must understand the internal operations of that group, but not of the rest of the knowledge base. If he preserves the correct functioning of the rules within the group and does not change the validity of the assertions about its inter-group facts, the maintenance programmer can be confident that the change that has been made will not adversely affect the rest of the system. Conversely, if he wants to use additional inter-group facts from other groups, he should rely only on the assertions provided for them, not on the internal workings of the rules in the other group. (Of course, changes that pervade several groups would still have to be handled as they always have been, but the grouping is intended to minimize these.)

```
(GROUP isamammal
     (PRODUCES
          (mammal "is it a mammal,
               by conventional English usage"))
     (RULES
          (r1 (IF hair) (THEN mammal))
          (r2 (IF milk) (THEN mammal))))

(GROUP isabird
     (PRODUCES
          (bird "is it a bird, by English usage"))
     (RULES
          (r3 (IF feather) (THEN bird))
          (r4 (IF flies ovip) (THEN bird))

(GROUP isacarn
     (PRODUCES
          (carn "is it a carnivorous creature"))
     (RULES
          (r5 (IF meat) (THEN carn))
          (r6 (IF pointed claws fwdeyes)
                (THEN carn))))

(GROUP isungulate
     (PRODUCES
          (ungulate "is it an ungulate"))
     (USES
          (mammal))
     (RULES
          (r7 (IF mammal hoofs) (THEN ungulate))))

(GROUP giraffe
     (USES (ungulate))
     (RULES
          (r10 (IF ungulate longn longl darksp)
                (THEN giraffe))))
etc....
```

Figure 1. Example of a grouped set of rules.

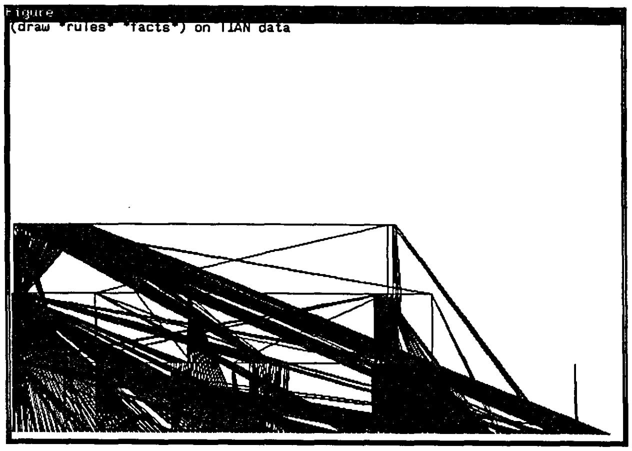
Partitioning the Knowledge Base

To decide whether partitioning a knowledge base is a feasible approach, we are analyzing existing production systems to determine how the rules in the system are related to each other. We have developed a software tool that analyzes the connections between the rules of a production system. The input to the tool is a set of rules expressed in an abstract form.

We are using the tool to determine whether the rules are indeed thoroughly intertwined or sufficiently separated that they could be divided into groups. To date, we have analyzed several knowledge bases and found that there is considerable separability and latent structure to the relationships between the rules in these systems, which could be exploited to improve maintainability.

Next, we are attempting to divide. the rules of existing systems into appropriate groups automatically, using several new approaches. By grouping the rules of existing production systems, we hope to determine whether such systems *could* have been cast in the mold required by the new method or whether it would have imposed excessive restrictions and unnatural structure on the developer. Based on the latent structure in rules found thus far, initial results suggest that the present approach can be imposed on many rule-based systems. They also suggest that an ideal, but not always attainable, grouping of rules is one in which each group of rules sets the value of only one fact that is used by rules outside this group.

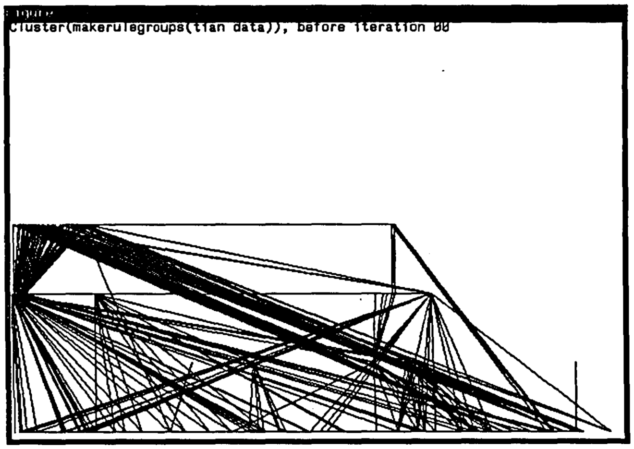
Rules are related to each other through the facts whose values they use or modify. First, we depicted these relationships in a graph, showing the inference hierarchy for the system. Figure 2 shows such a graph for an expert system developed by Reggia at the University of Maryland, using the KES language;[3] it is used to diagnose stroke and related diseases. In Figure 2, each node, or point, represents a rule and each link, or line, between two rules represents a fact whose value is set by one rule and used by the other.



(draw *rules* *facts*) on TIAN data

Figure 2. Plot of individual rules of an expert system.

The first algorithm considered attempts to partition this graph into a collection of rooted trees of rules, where the "root" of each such tree is a fact. That is, divide the rules into groups such that each group of rules produces only one fact that is used by other groups. This provides the desideratum mentioned above, since each rule group sets the value of only one external fact. Figure 3 shows the same system as Figure 2, after such an algorithm was applied. Each node now represents a group of rules, and each link represents a fact that is produced by rules in one group and used by those in another group. Facts that are produced and used entirely within a single group do not appear in the graph.

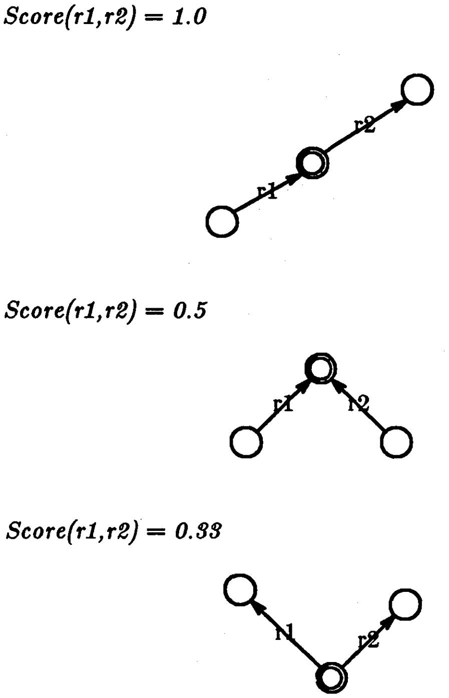


Cluster(makerulegroups(tian data)), before iteration 80

Figure 3. Rules of expert system, grouped into trees.

This algorithm tends to divide the knowledge base into many small groups. Each such group contains a collection of rules whose effect on other groups is entirely summarized by the fact at the root of the tree. Hence rules in these groups intuitively belong together under any grouping scheme. The problem is that the many small groups now must be combined into larger agglomerations. One alternative tested was to weaken the criterion for being a "rooted tree." That is, divide the rules into groups such that each group produces no more than **n** external facts, where **n** is now greater than 1. However, as **n** was increased, this approach did not appear to expose any natural structure in the knowledge bases.

Next, an approach based on cluster analysis was developed. Given a collection of objects, a clustering algorithm partitions them into groups of like objects. To use such an algorithm, though, a measure of distance or "relatedness" between rules must be defined. Since our ultimate concern is for a programmer making changes to the knowledge base, the similarity between two rules should measure: *If one rule were changed, how likely is the other rule to have to be changed also.* Rules affect each other through the facts they have in common. Thus a simple measure of the "relatedness" of two rules is the number of facts that are mentioned in the left or right hand sides of both rules. Since there are several ways in which two rules could refer to the same fact, we decided to weight this count. The two rules `if A then B` and `if B then C` share fact `B` in common; so do the two rules `if A then B` and `if C then B`. The rules of the former pair seem to have a greater programming effect on each other than the latter pair, and hence should be more "related." Figure 4 summarizes the three ways in which two rules can share a fact, and the weights given to each. The total "relatedness" measure between two rules is, then, a weighted count of the facts shared by both rules, where each fact is weighted by the score that indicates in which of the three possible ways the two rules use the fact.


Score(r1,r2) = 1.0

Score(r1,r2) = 0.5

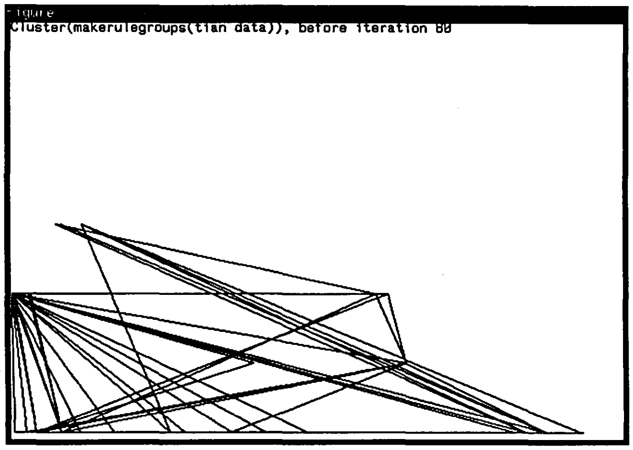
Score(r1,r2) = 0.33

Figure 4. Components of "relatedness" measure between two rules.

Given such a measure, we can proceed with a straightforward clustering algorithm. First, measure the similarities between all pairs of rules, select the closest pair, and put those two rules together into one cluster. Then, repeat the procedure, grouping rules with each other or possibly with already-formed clusters. In the latter case, we must measure the "relatedness" between a rule and a cluster of rules. This is simply defined as the mean of the similarities between the individual rule and each of the rules in the cluster, corresponding to an average-linkage clustering procedure. The algorithm proceeds iteratively.

The clustering algorithm can also be started with the small groups found by the rooted-tree algorithm above, instead of starting with individual rules. Since the tree groups appeared promising, but just too numerous, this is a reasonable alternative, and it appears to produce slightly better results. Figure 5 shows the rule groups of the system of Figures 2 and 3 after clustering in this fashion.

Thus far, this method appears to do the best job of partitioning a set of rules in an intuitively reasonable way. One drawback to the algorithm is that on each iteration it makes the best possible agglomeration of two groups, but it never backtracks, in case there might be a better grouping for the system considered as a whole. Also, like most clustering algorithms, if it runs for enough iterations it will eventually group all the rules into one large group; a stopping rule is thus needed.



Cluster(makerulegroups(tian data)), before iteration 80

Figure 5. Rules, grouped into trees then clustered.

Software Tools and Measurements

We have developed software tools to support this programming methodology. The developer can define the grouping of rules and input the knowledge base in the form shown in Figure 1, or he can run one of the grouping algorithms discussed to produce the grouping. These groupings have no impact on system performance since they are invisible to the rule interpreter. Given such a grouping, the software then automatically identifies the intra-group and inter-group facts. It flags all inter-group facts produced by a group, so the programmer can provide assertions for them; it flags all inter-group facts used by a group and retrieves their descriptions, so the programmer can rely on them when using such facts. Other software tools can trace all effects of changing a given rule and can find any unused rules or groups.

The division of a set of rules into groups should attempt to minimize the amount of coupling between the groups and maximize the amount of cohesiveness within each group.[4] Defining measures for these notions will provide data to help compare alternative groupings of a given set of rules. Once a set of rules is divided into groups, each fact in the system can be characterized as being used and produced by rules entirely within a single group or else as being used or produced by rules in more than one group. One simple measure of coupling is the proportion of facts in the second category, while cohesion is represented by the proportion of facts in the first.

Another approach to these measures is also being investigated. For coupling, it uses the average "relatedness" between all pairs of rules, where members of the pairs lie in different groups. For overall cohesion, it uses the average relatedness of every pair of rules that lie in the same group. For the example shown above, these quantities are 0.0798 average coupling and 0.9238 average cohesion, suggesting a far better than random organization.

Conclusions

By studying the connectivity of rules and facts in several typical rule-based expert systems, we found that they indeed have a latent structure, which can be used to support a new programming methodology. We have developed a methodology based on dividing the rules into groups and concentrating on those facts that carry information between rules in different groups. We have also studied several algorithms for grouping the rules automatically. Finally, we have developed a simple language and some software tools and measures to support the new method.

The resulting programming methodology requires the knowledge engineer who develops a rule-based system to declare groups of rules, flag all between-group facts, and provide descriptions of those facts to any rule groups that use such facts. The knowledge engineer who wants to modify such a system then gives special attention to the between-group facts and preserves or relies on their descriptions when making changes.

An interesting aspect of this approach is that it draws distinctions between the facts contained in working memory of a production system. Certain facts are flagged as being important to the overall software structure of the system, while others are "internal" to particular modules and thus less important. Programmers can be advised to pay special attention to rules that involve the "important" facts. This is in contrast to the homogeneous way in which the facts of a rule-based system are usually viewed, where they must all command equal attention or inattention from the programmer.

To determine how well this programming methodology will work, we are attempting to retro-fit several existing knowledge bases with this approach. This will help determine how well the structure implied by the new programming methodology can fit the structures observed in actual rule bases. We will use the grouping algorithms to divide the rules and then use measures of coupling and cohesion to compare alternative groupings. Next we will attempt to measure the extent to which the new method helps or hinders maintenance of an expert system. We will attempt to make changes both to a conventional expert system and to one divided into groups following the proposed method and compare the results. Based on our preliminary results, the method does not impose unreasonable restrictions on the developer nor does it lead to unnatural structures.

References

1. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* **15** pp. 1053-1058 (1972).
2. D. L. Parnas, "Software Engineering Principles," *INFOR Canadian Journal of Operations Research and Information Processing* (November, 1984).
3. J. Reggia, "Knowledge-based Decision Support Systems: Development through KMS," *Department of Computer Science, University of Maryland* (1981).
4. W.P. Stevens, G.J. Meyers, and L.L. Constantine, "Structured Design," *IBM Systems Journal* **13** pp. 115-139 (1974).
5. P.H. Winston and B.K.P. Horn, *LISP*, Addison-Wesley, Reading, Mass. (1980).

AN ORGANIZATIONAL FRAMEWORK FOR COMPARING ADAPTIVE ARTIFICIAL INTELLIGENCE SYSTEMS

Teresa A. Blaxton and Brian G. Kushner

The BDM Corporation
7915 Jones Branch Drive
McLean, Virginia 22102

One of the most interesting topics of investigation in the field of AI is machine learning where systems are being made to automatically "adapt" or learn over time. The following paper presents a common framework for organizing several diverse adaptive AI systems. Nine systems are discussed with regard to: (a) the types of knowledge representations they employ; (b) storage; (c) retrieval mechanisms; (d) conflict resolution principles; and (e) means of adapting knowledge and control structures as a function of changing experience.

There is a new trend evolving in artificial intelligence which directly impacts the work being done on expert systems. More and more, people are becoming unsatisfied with constructing static knowledge bases which are obsolete almost as soon as they are completed and must be updated on a continual basis. The alternative being explored is that of creating systems which adapt with the addition of new knowledge, not only learning new facts but actually changing their own memory organizations and control structures to accommodate the new data more efficiently.[1,2,3] Such systems are potentially more user oriented, more flexible to operate, and are less costly from a software/life-cycle support viewpoint.

Although several researchers have attempted to build adaptive systems (ASs), they have unfortunately done so without the benefit of any guiding theory. Those frameworks that have been offered have not been sufficiently general to encompass the wide variety of systems that have been developed.[4,5,6] Consequently, the literature addresses ASs from multiple perspectives and with inconsistent terminology, making it difficult to compare the accomplishments of these systems with one another. In this article, an organizational structure will be suggested for this body of research, hopefully lending some insights into directions for future development.

This organizational framework has several parts, all focusing on issues relating to the incorporation and utilization of knowledge in ASs. In terms of distinctions between various ASs, the type of knowledge acquired by these systems, as well as how this knowledge is represented, are the aspects most similar to those of traditional expert systems. Other familiar elements of this framework include the retrieval operations used by the AS to access previously stored information, and the mechanisms used to control the processing in the system. But that is where the similarity ends, since both the mechanisms by which incoming knowledge is automatically stored in the system, and the strategies for modifying or adapting the knowledge base and control structure, are by necessity unique features of these adaptive systems. These criteria will be discussed in greater detail in subsequent sections of this paper, where we will in turn (a) establish the need for building ASs; (b) outline a framework of the concerns one might face when trying to build an AS; (c) compare and contrast several already existing ASs; and (d) lay out some guiding principles for building an "ideal" AS.

Problems With Traditional Expert Systems

Many have argued that expert systems are the one area in which the AI enterprise has been truly successful.[8,9] These claims are based upon the proven utility of some computerized expert systems that are used to aid human experts in solving problems within limited application domains.[10,11] Despite these successes, there are still many nontrivial problems associated with building and maintaining expert systems, a few of which will now be enumerated.

To begin, the process of knowledge engineering, whereby the knowledge base of an expert system is acquired, is fraught with difficulties.[12] At worst, the availability of the main expert or experts may be limited or inadequate during the knowledge base construction period. Barring this eventuality, the experts that are available may disagree as to what knowledge to include, leaving the knowledge

CH2345-7/86/0000/0190$01.00 © 1986 IEEE

engineer at a loss as to how to resolve the dilemma. Perhaps more seriously, the knowledge engineer is faced with the task of obtaining procedural knowledge from the expert who can report only declarative knowledge. That is, the expert can declare that something is true, but cannot accurately describe how to do it.

Aside from problems encountered in the construction of traditional expert systems, a host of difficulties arise once the system is completed. Foremost among these is the challenge of updating the knowledge base to keep it current while maintaining truth value in the system. This becomes particularly acute as the size of the knowledge base increases. The practice of modularizing the knowledge base into separate sections and assigning each to a different knowledge engineer has eased this situation somewhat.[13] However, most would agree that there is still room for improvement, since there are still problems associated with maintaining knowledge consistency and coordination across these modules. Perhaps even a larger concern is that most expert system formulations offer no provision for automatically augmenting the knowledge base and control structure as the need arises. These difficulties, coupled with the often limited domain of applicability of most systems, suggest that investigation of an alternative approach is warranted.

Recent Advances: Adaptive Systems

As was already mentioned, recent work in the area of adaptive systems has led to theoretical advances over traditional expert system formulations.[14] If realized on a large scale, AS would be preferred to expert systems for several reasons. First ASs may be implemented without access to an "expert" per se. That is, the system may start out at a novice knowledge level, and gradually build up to a higher level of expertise through experience and through interaction with some external "knowledge sources." Such a system would be useful in so-called "cutting edge technology" domains where the current knowledge base is small, but expected to grow.

In terms of the user/system interface, ASs may be particularly beneficial in the development of individualized systems where commands accepted by the system are customized for a small set of users. In addition, one might imagine that an AS that has itself progressed from the novice to expert stages would provide more understandable responses to queries made by a novice user than would a traditional expert system.[14]

In a more global sense, ASs are preferable to traditional expert systems for the simple reason that intelligence is dynamic, and in any domain of interest the knowledge and heuristics utilized are bound to change with time. A major amount of effort has been expended in the past to find domains which are suitably narrow and static for building expert systems. In spite of this, the systems that have been developed must still be updated and augmented fairly frequently, and hence require costly, manpower intensive, support tails. One way to avoid this pitfall is to build systems that adapt. Therefore, it is our belief that the performance of the system will be improved to the degree that the system and its knowledge base adapt to the gradual evolution of the domain itself. *

Much of the pioneering work on ASs has been conducted by cognitive psychologists interested in modeling different aspects of human cognition. These researchers have experimented with the application of learning principles to AI systems in domains as diverse as number categorization, puzzle solving, language acquisition, and manipulation of geometric figures. Due to the disparity of these topics, readers of this literature may sometimes find it difficult to apprehend relevant similarities and differences among these systems. The organizational framework presented here is intended to bring a certain order to this chaos by providing points of comparison common to all of these systems, varied though they may be. For purposes of the present paper, the framework will be discussed in regard to only a subset of ASs. ** Before presenting the framework, each of ASs to be discussed will first be briefly described.

Representative Crossection of Adaptive Systems

The first AS to be described is called ACT* and was published by John Anderson.[1] Intended as a comprehensive theory of human cognition, ACT* models such complex activities as rotating mental images, solving geometry proofs, retrieval processes involved in reading, and language acquisition. It is by far the most fully developed system to be described in this article. Since ACT* was designed to model human cognition, it may embody some constraints that are unattractive in the realm of AI applications. Nevertheless it is argued that there is much to learn about building ASs from the study of such a system.

* It has been argued elsewhere that building ASs is not necessarily a worthwhile enterprise in that learning can be a long and iterative process, perhaps requiring more effort than is merited.[33] it is our position, however, that the long term benefits realized from the construction of ASs in dynamic domains will far outweigh any initial startup costs.

** As the reader may notice, the nine systems described in this article constitute only a small number of those programs presented elsewhere as ASs. Some of those other systems have been eliminated from our present discussion because they are better labeled as frameworks for knowledge representation rather than as full blown systems Still others were omitted because they are not truly adaptive in the sense of having mechanisms responsible for reorganizing memory and control structures. Finally, those deemed too narrow in scope to be of general interest were not included for discussion.[37]

The BACON.5 program written by Langley, Bradshaw, and Simon [15] was intended for a very different purpose. Given numeric data and variable specifications it notices patterns, abstracting out regularities among combinations of variables as "concepts". BACON.5 has discovered the Ideal Gas Law, Ohm's Law, and the law of gravitation, among others. A program somewhat similar to BACON.5 is UNIMEM.[16] It categorizes and makes generalizations from numeric data without using any statistical heuristics such as those employed in the BACON.5 system. For example, given data on areas and populations of states, it categories them into "large" and "small" classes and forms appropriate generalizations about those classes. For instance, it generalizes that small states have small populations without incorrectly inferring that large states necessarily have large populations (e.g., Alaska).

The next three systems are similar to one another in that they all start out with little or no knowledge about a given topic and progress up to an expert level. Kolodner's [17,2,18] Computerized Yale Retrieval and Updating System (CYRUS) learns facts about Cyrus Vance and Edmund Muskie's terms as Secretary of State. The Integrated Partial Parser, IPP, learns about international terrorism from newspaper articles. Finally, a follow-up to IPP called RESEARCHER [19] learns and answers questions about patent abstracts.

For purposes of the present paper, the most notable aspects of all of these systems are that they have mechanisms for organizing and updating their knowledge bases such that interesting generalizations and discriminations result. For example, after learning about a number of instances in which kidnap victims in Italy happened to be businessman, IPP is able to generalize that a new (unidentified) Italian who is kidnapped is likely to be a businessman. IPP does not make the error of the converse, however, which would be to infer that a given kidnapping took place in Italy simply because the victim is a businessman.

An example of an AS in a very different domain is the Blocks World program, first introduced by Winograd [20] and then updated by Winston.[21] This system learns about various possible configurations of a set of geometric figures. For example, Winograd's [20] system can remember sequences of moves for any given object in the blocks world and infer procedures necessary for those sequences to have been performed, even though that information was not explicitly stored.

The last two ASs learn rules about how to solve some problem. The highly acclaimed Meta-DENDRAL [22] is a program which sits over the expert system DENDRAL and learns cleavage rules used by mass spectrometers. The second version of the Strategic Acquisition Governed by Experimentation system, SAGE.2 [3] learns rules for solving the Tower of Hanoi puzzle task. Starting out with a small number of heuristics this program quickly acquires the knowledge necessary to solve the puzzle in the minimum number of moves.

Having read these descriptions the reader should now have an appreciation for how diverse these ASs really are. Nevertheless, it is argued that an organizational framework may be used to view these systems which will allow their simultaneous comparison on a number of dimensions. That framework will now be presented.

Framework for Comparing Adaptive Systems

The ASs listed just described may be compared with regard to the following features: (a) the types of representation schemes employed; (b) the mechanisms by which incoming knowledge is stored in the system; (c) retrieval operations used to access previously stored information; (d) mechanisms used to control information processing in the system; and (e) strategies for adapting the knowledge base and control structure to accommodate the changing environment. In this section, our set of ASs will be discussed with regard to each of these features. Following this, an evaluation of the different designs employed in building these ASs will be presented.

A. Type of Knowledge Representations Used

1. Production Rules

The first type of representation listed in Table 1 is the production rule. Production rules are if-then or condition-action pairs which invoke a particular action to be carried out when the contents of working memory match a specified condition. By this description it is clear that production rules embody procedural knowledge, but they are closely tied to declarative knowledge as well. Consider the following example of a production rule in the BACON system: [23]

> If you see a number of descriptions at Level L in which the dependent variable (D) has the same number value (V),
>
> Then create a new description at level L+1 in which the value of D is also V and which has all conditions common to the observed descriptions.

This rule results in the creation of another production, which will, in some sense, represent declarative knowledge within its own conditions. When these new conditions are matched by the contents of working memory, the new rule will be eligible for implementation. Production rules are used in ACT*, BACON.5, Meta-DENDRAL, and SAGE.2.

2. Memory Organization Packets (MOPs)

Following in a tradition somewhat similar to that of Minsky's frames, [24] Schank and Abelson's scripts, [25] and Schank's dynamic memory, [26] UNIMEM, CYRUS, IPP, and RESEARCHER all use memory organization packets, or MOPs. MOPs are a type of semantic network within which knowledge is arranged hierarchically by topic.[27] Information that constitutes a specific exception to the theme of the MOP is given a separate index, or label, which may be used to locate it more quickly during the retrieval process. When two or more elements are organized under one index, a new sub-MOP is formed, thus preserving the modularity of the memory network.

3. Propositions

Propositions have become a fairly common means of abstract knowledge representation in AI systems. They preserve semantic relationships between arguments or objects. For instance, the proposition "(kiss Sue Bill)" preserves the relation "kiss" between the arguments "Sue" and "Bill". Notice that the proposition's structure is independent of information order.[1] That is, "(kiss Sue Bill)" does not encode the difference between "Sue kissed Bill" and "Bill was kissed by Sue". Only the semantic relation is represented. Propositional representations are used both in ACT* and the Blocks World system.

4. Temporal Strings

Temporal strings are a type of representation used in the ACT* system to maintain order information. Knowledge about order is important to many tasks, one of which is the analysis of linguistic information. Temporal strings provide a more efficient means of storing order than do propositions, but cannot be used in the place of propositions since they do not incorporate information about meaning as effectively.

5. Spatial Images

This final type of representation preserves the configuration of elements in a spatial array. This construct is used in the ACT* system to model pattern recognition behavior in humans. The important point to note here about spatial images is that they preserve only information about the relative physical positions of objects in an array, and not necessarily information about what these objects actually look like.

B. Information Storage Strategies

Having established the tools for representing knowledge in our set of ASs, it is now of interest to explore the strategies used to store new information in these systems.

As shown in Table 2, one strategy for adding new information to the system is to learn every new stimulus as it is presented. This simple approach is used in the UNIMEM, CYRUS, Blocks World, and Meta-DENDRAL systems. Another alternative is to be more discriminating and store new information in the network only after it has been shown to have some importance, for instance after it has occurred several times. The ACT*, BACON.5, IPP, RESEARCHER, AND SAGE.2 systems all employ this type of approach.

Once the decision has been made to add new information to the system, the question arises as to how it will be stored in relation to already existing memory elements. Of course the new data could simply be added randomly. However, a more strategic approach is to store related items "near" one another in the network. The ACT*, UNIMEM, CYRUS, IPP, and RESEARCHER systems all employ this design, linking together items which are semantically or contextually related.

This method of storage serves two useful purposes. First, provided one uses the right type of knowledge representation, it will not be necessary to explicitly store all semantic features associated with every new element since some of those may already be represented in nearby (subsuming) structures. Second, retrieval of information is facilitated when memory is organized thematically. that is, one need only get to the right area in memory and look there rather than exhaustively searching the entire network.

Another useful tactic, implemented in UNIMEM, CYRUS, IPP and RESEARCHER, is to mark new elements with special indices which denote the way in which their themes differ from those of their "parent" nodes in the network. This approach has the same advantages as storing related items near one another in memory. It is particularly useful in retrieval as will be illustrated in the next section.

C. Retrieval Mechanisms

The three types of retrieval mechanisms employed by our set of ASs are listed in Table 3. The first of these, pattern matching, is a method whereby an item is retrieved from memory if the physical features of its representation match those of some retrieval cue. Pattern matching is commonly used in production systems where rules are selected for use based on the match between their conditions and the elements in working memory. The implementation of this mechanism usually involves exhaustive memory search. The ACT*, BACON.5, Blocks World, Meta-DENDRAL, and SAGE.2 systems all employ pattern matching.

A very different approach to retrieval is to search the network by traversing indices associated with individual categorical memory structures. Recall that the MOPs used to represent knowledge in the UNIMEM, CYRUS, IPP, and RESEARCHER systems have indices, or labels, associated with the. There are indices which denotes the thematic content of an MOP and other indices which tag elements involving exceptions to these themes. Retrieval begins with an index which is compared to other indices in the network. When a match is found, the memory elements subsumed underneath that index are searched. The result is that the scope of the retrieval process is limited to only those areas of memory which are the most relevant, a more efficient strategy than exhaustive search.

Another way of facilitating economic search is through spreading activation, as implemented in ACT*. At any given time, the memory elements that match the contents of working memory are temporarily activated, and this activation spreads to "nearby" connected elements in the network. Since the storage strategy in ACT* is to form connections among related items as they are learned, the nearby items that get activated will also be related to the contents of working memory. That is, all activated elements will be potentially relevant to the current context. Search is quite efficient since the retrieval process is directed only to the areas of the network that are activated, rather than to the network as a whole.

D. Principles of Control and Conflict Resolution

As might be expected, the search mechanisms just described often result in the retrieval of more than one acceptable memory element. The problem then arises as to how to choose among the potential candidates, selecting the most appropriate one for instantiation. A set of principles used to resolve these conflicts is presented in Table 4.

The most obvious means of deciding among potential elements is to choose the one which most closely matches the retrieval cue (or contents of working memory). this strategy is adopted in all of the ASs we are considering. In practice, however, one might find that as systems get more and more complex, this simple tactic will not always work well. That is, there may be several elements which match to the same degree, in which case further means for choosing among elements must be provided.

The ACT* and SAGE.2 systems employ several strategies for resolving these conflicts which rely on analysis of features of the operators themselves. For instance, each operator in these ASs has some strength associated with it which is either increased or decreased after each instantiation, depending on the outcome. Competing operators are then selected depending upon their relative strengths or histories of being useful. The field of potential candidates can be narrowed further using a principle called data refractoriness whereby no one operator can serve in two patterns simultaneously. Finally, the ACT* program relies on a principle of specificity whereby the most specific of two operators which match equally well will be chosen. For instance, if the pattern "barn" were being matched and the two elements "ba" and "bar" were competing, "bar" would be chosen since it is the most specific.

In addition to relying on traits of the individual operators to resolve conflicts, system behavior may be controlled in both ACT* and SAGE.2 by contextual constraints. Context is used to govern choice of operators in SAGE.2 through the "recency of use" rule. That is, the most recently used operator is chosen over others on the assumption that it must be the most relevant to the problem at hand. This choice will have nothing to do directly with the strength of the operator or whether the pattern is already represented elsewhere.

Perhaps a more interesting mechanism is used in the ACT* program. Problem solving in ACT* is goal-driven. That is, a large task having one ultimate goal can be broken down into several subtasks, each having a goal of its own. The current goal of interest is represented in working memory and, as such, disallows the instantiation of any operators not directly related to its completion. This mechanism greatly reduces the field of potential candidates from the start, thus eliminating many conflicts that might otherwise arise.

E. Mechanisms for Adaptation

Thus far the types of mechanisms described are not in any way peculiar to systems under consideration here. For instance, any traditional production system has to have some means of retrieving information from memory (usually pattern matching), and some way to resolve conflicts among competing memory elements (usually degree of match and specificity). The trouble with these typical expert systems, however, is that when they "learn" they do so simply by adding new facts or rules. These additions are, for the most part, made without regard to the integration and reorganization of this new information with existing knowledge.[28]. What sets adaptive systems apart from traditional expert systems is their ability to modify their own knowledge an control structures with experience in some meaningful manner. The ways in which adaptation occurs are listed in Table 5.

The most common way of incorporating experience into the system is to strengthen operators that have either been presented a number of times or have somehow proven useful in the past. In fact, every AS in our set of nine except Blocks World employs this method. However, the converse of this strategy, which is to weaken an operator whose instantiation has produced undesirable results, is not as popular. Only the ACT* and SAGE.2 systems employ this tactic. A method more drastic than weakening the strength of an operator is to remove it from the network altogether when if ceases to be appropriate for current applications. This is sometimes done with production rules in ACT*, but only very conservatively.

In addition to changing the relative strengths and weaknesses of operators in the system, it is possible to actually create new ones using knowledge embedded in previously existing representations. In particular, generalization involves the formation of new elements which embody common features of several representations already in the network. The new operator applies in more cases since it is less specific than its predecessors. This mechanism is implemented in every AS except Meta-DENDRAL and SAGE.2.

In contrast to generalization, new representations may be created through the process of discrimination. Discrimination occurs when extra delimiting features are added to an operator which restrict the contexts in which it can apply. In other words, discrimination creates constructs that are special cases of already existing ones. This is the type of process that occurs in CYRUS, for instance, when an index is added beneath the top level of an MOP to signify that an element contains information in exception to the theme of the parent MOP. Although potentially quite powerful, this strategy is employed only in the ACT* and CYRUS systems.

The discussion up to this point has focussed on the comparison of our nine ASs with regard to the types of knowledge they acquire, along with the types of representations, storage mechanisms, retrieval strategies, control principles, and means of adaptation used by each. Now that the ASs have been couched in terms of this organizational framework, it is of interest to determine whether any guiding principles may be used in conjunction with this framework to help researchers both (a) build better ASs and (b) evaluate our nine Ass in relation to one another.

Building the "Ideal" Adaptive System

The remainder of this paper will outline some general guidelines for building what we call the "ideal" AS. These guidelines will be cast in terms of the organizational framework already presented. It is' hoped that these principles will not only aid in the design of future ASs, but might serve as a set of metrics by which to evaluate already existing systems.

First it is our feeling that there is no way to select the "best" representation a priori. In fact some might argue that, as far as the outward behavior of the system is concerned, any deficiencies associated with the choice of representation structure may be compensated for in terms of the procedures implemented to operate on those structures.[29] We will concede that some representations may lend themselves to certain types of problems more easily than others, although this will be a matter of convenience.

Having dispensed with the question of representation, we will now proceed with recommendations concerning the other issues of storage, retrieval, control, and adaptation. After reading through the earlier comparisons of the ASs on each of the dimensions just listed, the reader may feel that a "more is better" rule is applicable. That is, it might appear that variety is the key, with the better systems being those which incorporate several different types of capabilities in each of these areas. Although this is true to a certain extent, it is not necessarily the best prescription for success.

Rather than a "more is better" methodology, we advocate that researchers adopt an approach whereby a combination of both top-down and bottom-up strategies are implemented in the system design. Bottom-up strategies are those for which properties of individual operators are important, whereas top-down strategies involve overall system behavior. Perhaps this concept is best illustrated with an example.

In terms of storage mechanisms, a top-down strategy is to store related items near one another in the network. This type of strategy impacts directly on the global organization and performance of the system. Similarly, the approach of storing items in a hierarchial organization such that concepts at a given level subsume those below and are subsumed by those above will later affect global retrieval operations.

In contrast, a bottom-up storage strategy is to learn every new stimulus as it is presented, regardless of its semantic content--that is, regardless of its eventual position in the overall memory network. This strategy has not been employed as much as it could have been, probably due to the semantic primacy bias in the human memory literature.[30] The traditional wisdom has been that people primarily remember semantic content in lieu of lower level information involving physical features of stimuli. Recent experiments on human memory have shown, however, that we do indeed remember this low level information, often better than the semantic content.[31] To the degree that the human is accepted as a useful model for designing intelligent systems, the importance of bottom-up storage strategies should not be ignored when building ASs.

Assessing the relative merit of combinations of storage mechanisms of the ASs from Table 3, it may be seen that only the UNIMEM and CYRUS systems employ both top-down and bottom-up strategies. Specifically, both are designed to learn every new stimulus as it is presented and to store related items near one another in memory. In addition, both of these systems use the MOP representation from which a hierarchical organization is created.

Applying the top-down/bottom-up distinction to retrieval mechanisms, it may be argued that memory search aided by either index traversal or spreading activation is top down since these mechanisms are implemented with regard to the memory network as a whole. On the other hand, pattern matching is a bottom-up retrieval tactic since it depends only on characteristics of individual operators. Again, research on human memory has shown that both bottom-up and top-down processes play important roles in the determination of retrieval performance [32] lending credence to the assertion that this combination might be useful in the design of ASs. In Table 4 we see that the only system employing both top-down and bottom-up retrieval strategies is ACT*. Elements may be retrieved from memory in ACT* using either spreading activation or pattern matching.

As with storage and retrieval, there are principles of conflict resolution which embody both top-down and bottom-up features. For example, a top-down strategy for delimiting the field of competing operators is to impose a goal hierarchy on the system. The choice of priorities rather than the traits of any individual operators. A similar argument may be made for the strategy of choosing the most recently used operators over others that match as closely.

All other conflict resolution principles discussed earlier involve bottom-up analysis. These include degree of match, specificity, data refractoriness, and operator strength. Notice that all of these require that features of individual operators dictate which will be selected for implementation as opposed to overall contextual constraints.

Looking to Table 5 again, it may be seen that the only two systems which employ top-down and bottom-up strategies for controlling choice of operators are ACT* and SAGE.2. Both use several bottom-up strategies. Of greater interest is that the top-down approach to control in ACT* is implemented in a goal hierarchy, whereas a recency criterion serves as a context mechanism in SAGE.2.

The final dimension to be considered from the organizational framework is the manner in which systems are allowed to adapt. Of the mechanisms presented earlier, adaptation through generalization and discrimination are top-down in nature because they occur only when similarities or differences among several structures are noticed simultaneously in one context. On the other hand, the strengthening, weakening, and unlearning of individual elements all affect only one structure at a time. Since they do not directly impact on the status of other contextually-related elements, these strategies are classified as being bottom-up. All of the ASs invoke both top-down and bottom-up adaptation strategies except Meta-DENDRAL and SAGE.2 which employ only bottom-up procedures.

Conclusions

The above discussion focused on developing an organizational framework for comparing adaptive systems. These systems are of interest relative to traditional expert systems in that they (a) do not have to be updated by hand as knowledge in the domain of interest changes; (b) can start out at the novice level without explicit access to a human expert; (c) are useful in "cutting edge" domains in which few, if any, experts exist; and (d) might provide better interfaces for novice users since their own memory structures have evolved from the novice level. Nine systems were discussed with regard to an organizational framework for comparing ASs. This framework was used to compare the systems on such dimensions as the type of knowledge acquired by each and representations used; storage and retrieval strategies, control mechanisms, and methods used for adaptation. As we would expect, based on our experience with the latest generation of expert systems, we have recommended that ASs be designed using a combination of top-down and bottom-up mechanisms in each of these areas.

The development of this framework for ASs systems may also have several additional benefits. As the most near term example, this framework can aid in our understanding of AI system performance. This may allow researchers to address some of the perennial AI questions, such as how one measures the robustness of a given AI system, what are valid benchmarks for AI systems, and how an AI system can be designed for ease of testing. On this latter point, research in AS methodologies could expedite the rapid prototyping of AI systems, in a manner analogous to the techniques used in the electronics industry. Finally, continued development of ASs, particularly through rapid prototyping, can lead to a faster incorporation of AI systems into the marketplace.

Acknowledgments

The authors would like to thank Drs. L. H. Reeker, R. A. Geesey, C. B. Friedlander and C. T. Butler for their timely insights and constructive criticism throughout the development of this effort. This work was supported in part by DARPA under contract number MDA903-85-C-0056 and by DARPA/AFOSR under contract number F49620-86-C-0036.

REFERENCES

(1) Anderson, J. R. (1983) The architecture of cognition, Harvard University Press: London.

(2) Kolodner, J. A. (1983a) Maintaining organization in a dynamic long term memory. Cognitive Science, 7, 243-280.

(3) Langley, P. (1985) Learning to search: From weak methods to domain-specific heuristics. Cognitive Science, 9, 217-260.

(4) Bundy, A., Silver, B., & Plummer, D. (1985) An analytical comparison of some rule learning programs. Artificial Intelligence, 27, 137-181.

(5) Carbonell, J. G. (1983) Learning by analogy: Formulating and generalizing plans from past experience. In Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (Eds.) Machine learning: An artificial intelligence approach. Tioga Publishing Co., Palo Alto, CA.

(6) Michalski, R. S. (1983) A theory and methodology of inductive learning. In Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (Eds.), Machine learning: An artificial intelligence approach. Tioga Publishing Co., Palo Alto, CA.

(7) Rychener, M. D. (1983) The instructible production system: A retrospective analysis. In Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (Eds.), Machine learning: An artificial intelligence approach. Tioga Publishing Co., Palo Alto, CA.

(8) Bobrow, D. G., & Hayes, P. J. (1985) Artificial intelligence--Where are we? Artificial Intelligence, 25, 375-415.

(9) Hayes-Roth, F. (1985) Engineering systems of knowledge: The great adventure ahead. In (K. N. Karna, Ed.) proceedings of the Expert Systems in Government Symposium, pp. 678-692.

(10) Pople, H. (1982) Heuristic methods for imposing structure on ill-structured problems: The structure of medical diagnostics. In P. Szolovits (Ed.) Artificial intelligence in medicine. Westview Press Co.

(11) Shortliffe, E. (1976) Computer-based medical consultations: MYCIN. American Elsevier, New York.

(12) Hayes-Roth, F. (1984) The knowledge-based expert system: A tutorial. IEEE, September, pp. 11-28.

(13) Frosher, J. N., & Jacob, R. J. K. (1985) Designing expert systems for ease of change. In Proceedings of the IEEE Computer Society Expert Systems in Government Symposium, K. N. Karna (Ed.), Computer Society Press.

(14) Riesbeck, C. K. (1984) Knowledge reorganization and reasoning style. In M. J. Coombs (Ed.) Developments in expert systems, Academic Press: London.

(15) Langley, P., Bradshaw, G. L., & Simon, H. A. (1981) BACON.5: The discovery of conservation laws. In proceedings of the 7th International Joint Conference on Artificial Intelligence, 121-126.

(16) Lebowitz, M. (1985) Categorizing numeric information for generalization, Cognitive Science, 9, 285-308.

(17) Kolodner, J. A. (1983a) Maintaining organization in a dynamic long term memory. Cognitive Science, 7, 243-280.

(18) Kolodner, J. A. (1983c) Retrieval and organizational strategies in conceptual memory: A computer model. Lawrence Erlbaum Associates, Hillsadale, N.J.

(19) Lebowitz, M. (1986) An experiment in intelligent information systems: RESEARCHER. Unpublished manuscript.

(20) Winograd, T. (1972) Understanding natural language. New York: Academic Press.

(21) Winston, P. (1975) Learning structural descriptions from examples. In P. H. Winston (Ed.), The psychology of computer vision, New York: McGraw-Hill.

(22) Buchanan, B. G., Smith, D. H., White, W. C., Gritter, R. J., Feigenbaum, E. A., Lederberg, J., & Djerassi, C. (1976) Applications of artificial intelligence for chemical inference XXII: Automatic rule formation in mass spectormetry by means of the META-DENDRAL program. Journal of the American Chemical Society, 98, 6168-6178.

(23) Langley, P., Bradshaw, G. L., & Simon, H. A. (1983) Rediscovering chemistry with the BACON system. In Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (Eds.), Machine learning: An artificial intelligence approach. Tioga Publishing Co., Palo Alto, CA.

(24) Minsky, M. (1975) A framework for representing knowledge. In P. Winston (Ed.) The psychology of computer vision. McGraw-Hill.

(25) Schank, R. C., & Abelson, R. (1977) Scripts, plans, goals, and understanding. Hillsdale, NJ: Erlbaum.

(26) Schank, R. C. (1982) Dynamic memory: A theory of reminding and learning in computers and people. Cambridge University Press: London.

(27) Chandrasekeran, B., & Mittal, S. (1984 Deep versus compiled knowledge approaches to diagnostics problem solving. In M. J. Coombs (Ed.) Developments in expert systems, Academic Press: London.

(28) Kolodner, J. A. (1984) Towards and understanding of the role of experience in the evolution from novice to expert. In M. J. Coombs (Ed.) Developments in expert systems, Academic Press: London.

(29) Anderson, J. R. (1978) Arguments concerning representations for mental imagery. Psychological Review, 85, 249-277.

(30) Sachs, J. S. (1967) Recognition memory for syntactic and semantic aspects of connected discourse. Perception and Psychophysics, 2, 437-442.

(31) Kolers, P. A. (1976) Reading a year later. Journal of Experimental Psychology: Human Learning and Memory, 2, 554-565.

(32) Roediger, H. L., & Blaxton, T. A. (in press) Retrieval modes produce dissociations in memory for surface information. In D. S. Gorfein and R. R. Hoggman (Eds.) Memory and cognitive processes: The Ebbinghause Centennial Conference. Hillsadale, N.J.

(33) Buchanan, B. G., Barstow, D., Bechtal, R., Bennett, J., Clancey, W., Kulikowski, C., Mitchell, T., & Waterman, D. A. (1983) Constructing an expert system. In F. Hayes-Roth, D. A. Waterman, & D. B. Lenat (Eds.) Building expert systems. London: Addison-Wesley.

(34) Brachman, R. J., & Schmolze, J. G. (1985) An overview of KL-ONE knowledge representation system. Cognitive Science, 9, 171-216.

(35) Feigenbaum, E. A., & Simon, H. A. (1984) EPAM-like models for recognition and learning. Cogntive Sciences, 8, 305-336.

(36) Lenat, D. B. (1983) The role of heuristics in learning by discovery: Three case studies. In Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (Eds.), Machine learning: An artificial intelligence approach. Tioga Publishing Co., Palo Alto, CA.

(37) Quinlan, J. R. (1983) Learning efficient classification procedures and their application to chess and games. In Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (Eds.), Machine learning: An artificial intelligence approach. Tioga Publishing Co., Palo Alto, CA.

Table 2

STRATEGIES FOR STORING INFORMATION

	Learn every new stimulus	Learn only "important" stimuli	Store related items "near" one another	Index for ease of search
ACT*		X	X	
BACON.5		X		
UNIMEM	X		X	X
CYRUS	X		X	X
IPP		X	X	X
RESEARCHER		X	X	X
BLOCKS WORLD	X			
META-DENDRAL	X			
SAGE.2		X		

Table 3

MECHANISMS USED FOR INFORMATION RETRIEVAL

	Pattern Matching	Index Traversal	Spreading Activation
ACT*	X		X
BACON.5	X		
UNIMEM		X	
CYRUS		X	
IPP		X	
RESEARCHER		X	
BLOCKS WORLD	X		
META-DENDRAL	X		
SAGE.2	X		

TABLE 1

CONSTRUCTS USED FOR KNOWLEDGE REPRESENTATION

	Production Rules	MOPs	Propositions	Temporal Strings	Spatial Images
ACT*	X		X	X	X
BACON.5	X				
UNIMEM		X			
CYRUS		X			
IPP		X			
RESEARCHER		X			
BLOCKS WORLD			X		
META-DENDRAL	X				
SAGE.2	X				

Table 4

PRINCIPLES OF CONTROL

	Degree of Match	Operator Strength	Data Refractoriness	Specificity	Recency of use	Goal Dominance
ACT*	X	X	X	X		X
BACON.5	X					
UNIMEM	X					
CYRUS	X					
IPP	X					
RESEARCHER	X					
BLOCKS WORLD	X					
META-DENDRAL	X					
SAGE.2	X	X	X		X	

Table 5

MECHANISMS FOR ADAPTATION

	Strengthening	Weakening	Unlearning	Generalization	Discrimination
ACT*	X	X	X	X	X
BACON.5	X			X	
UNIMEM	X			X	
CYRUS	X			X	X
IPP	X			X	
RESEARCHER	X			X	
BLOCKS WORLD				X	
META-DENDRAL	X				
SAGE.2	X	X			

AN OBJECT/TASK MODELING APPROACH

BASED ON DOMAIN KNOWLEDGE AND CONTROL KNOWLEDGE REPRESENTATION

Qiming Chen

National Land Information System
Research Institute of Surveying and Mapping, Beijing, China

ABSTRACT

The need for developing knowledge-based information systems has given rise to studies on integrating AI systems and DB systems. An object/task modeling approach based on knowledge representation is proposed which is characterized by specifying object behaviors and domain rules in terms of object-oriented logic programming, and specifying tasks and meta-rules in terms of network-oriented formalism. This approach may be viewed as a step to the integration of object-oriented programming, logic programming, semantic modeling and event modeling, and to the combination of forward chaining and backward chaining techniques. It can enhance the flexibility and extendability of knowledge based systems to accommodate applications in multiple domains.

INTRODUCTION

The need for developing the next generation information systems has given rise to studies on integrating database systems and Artificial Intelligence (AI) systems [Ullman 85]. The fact that both Logic Programming (LP) and Relational Database (RDB) are subsets of first order predicate calculus suggests a resemblence between them. On one hand, LP provides powerful formalism for representing database rules and deductive retrievals, on the other hand, RDB provide adequate means of storing and managing knowledge [Gall 84] [Parker 84] [Fuchi 85] [Zani 84] [Chen 86a].

However, most of the knowledge based systems and decision support systems currently in operation, support only a single model or a class of models. They are special application oriented and self-contained, their configurations are both static and intolerant from a model perspective. In other words, such systems lack the flexibility of being adaptable to the possible alterations in the problems they were originally designed to deal with, and cannot be utilized in multiple problem domains, even if those domains are closely related.

This situation is caused, to a certain extent, by lacking the formalism for modeling objects and representing control knowledge within a system framework.

In fact, to manage a knowledge base for supporting tasks, we must take into account the follows :

(1) Domain knowledge representation and control knowledge representation.

(2) The sharing of objects and their properties by multiple tasks.

Thus the Object/Task modeling contains three levels : the object level, the association level, and the task level.

The proposed approach is characterized by

(1) A rule-based object modeling mechanism based on the integration of object-oriented programming paradigm and logic programming.

(2) Modeling the control structure of a problem-solving or decision-making task as a net, and then exploring a rule-based, linguistic and precise net specification formalism and a multiple level, task-oriented constraints handling mechanism.

(3) Treating task as system objects through which a task liberary may be constructed.

Under this object/task modeling approach, database applications are specified as localized domain knowledge representation and explicit control knowledge representation. Differnt roles an object plays in different tasks can be identified distinctly.

This approach is being designed and partially implemented on top of System G, an extended experimental RDB system [Melk 83ab] [Chen 85b] on a VAX computer.

CH2345-7/86/0000/0200$01.00 © 1986 IEEE

THREE LEVEL OBJECT/TASK MODELING

To illustrate the proposed concepts we first introduce a decision-making example concerning a simplified manufacturing order processing system for an instrument assembling company. The information needed is stored in certain relations (predicates), includes "order" (states orders), "asmb-des" (lists the parts needed for each model of machines, as well as the assembly time); "parts-inv" and "products-inv" (describe the inventories of parts and ready-made machines), and "parts" and "machines" (descride models and costs), whose instances are viewed as facts of the following form :

```
order (order_id, model, qty, finish_qty).
parts (code, cost).
machines (model, cost).
parts_inv (code, name, amt).
products_inv (model, name, amt).
asmb_des (model, asmb_hrs,
   config(senser, cpu, monitor, frame)).
```

Level 1 Object Modeling

The rule-based object/task modeling approach described in this paper is characterized by the integration of object-oriented programming, LP and RDB.

In the object-oriented semantic modeling paradigm, reality is represented in terms of objects as well as their relationships [Cox 84]. Each object has an associated set of procedures called methods denoting its dynamic behaviors. The manipulation of objects is made by applying the methods on them, referred to as "sending the objects a message".

Integrating object-oriented programming paradigm and logic programming, generally we identify an object structurally as :

(1) a primitive one represented by an LP predicate without any nested predicate as its argument.

(2) a compound one composed in a nested or recursive fashion from two or more objects, represented by an LP predicate with at least one predicate as its argument.

Then each method is stated as an LP clause. Passing a message is specified by using the infix operator ":", (the number of arguments may be zero). We classify two kinds of methods:

(1) i-method (the individual-oriented method), which is defined on a single object instance (universally quantified), as

```
i-method : object (arg1, arg2,...)
```

(2) c-method (the class-oriented method), which is defined on a class of object instances (the simplest example is a relation with multiple tuples). Its application is represented as

```
c-method : object ()
```

(3) mc-methods (the multi-class-oriented methods).

An object type is the structural specification of a class of objects, which describes its name, structure, methods and constraints, where the structure defines the attributes and constituents possessed by that object; methods consist of the rules applicable to the objects in term of Horn clauses; integrity constraints are assertions that instances of objects are compelled to abey, which can be classified according to various criteria, such as type constraints, dependency constraints and so on. In our approach a constraint is specified similar to the body of a Horn clause.

Thus for example the object type "order" is specified as :

```
object_type  order (ORDER_ID, MODEL, QTY, FINISH_QTY)
 {
  structure      { ORDER_ID    : c4,
                   MODEL       : c8,
                   QTY         : i4,
                   FINISH_QTY  : i4.
                   key : ORDER_ID
                 }
  constraints    { MODEL [ "hc20, hc40, hc60",
                   QTY > 0, QTY < 10000,
                   QTY > FINISH_QTY.
                 }
  i-methods      { what_kind_order ([instrument,MODEL]):-
                         order (_, MODEL, _, _).
                   unfinish_qty (X) :-
                         order (_, _, QTY, FINISH_QTY),
                         X is QTY - FINISH_QTY.
                 }
  c-methods      { diff (IDa, IDb, X) :-
                         order (IDa, _, QTYa, _),
                         order (IDb, _, QTYb, _),
                         X is QTYa - QTYb.
                 }
 }
```

An instance of an object is a predicate with all the arguments been instantiated with specific values. Thus the goal

```
?- unfinish_qty (X) :
   order (0056, hc40, 30, 10).
succeeds with X = 20.
```

The interpretation of a message consists in the unification of the objects, the unification of the method and the proof of the method. If an object is declared by "isa" as an sub-object of others and therefore inheriting their behavioral properties, then the unification of the method with the methods associated with the ancestors of the object in the "isa" network, is necessary. When an object has more then one ancestors, we first determine the order of the ancestors, then use a so called "upward-first" search strategy, starting from checking the first available ancestor, observe to see whether it is a root (no further ancestor remained), if not, move upward until either the unification succeeds or back to try the next possible ancestor at the highest possible ancestry level.

Passing message implies that certain action may be taken ruled by the specified method clause, then the resulting predicate (on the left side of the clause) can be reasonably viewed as a virtual object (VO). The instance of a VO is derived from the instances of other(source) objects, and instantiated by passing message thus represents the results of the message passing. Thus VO's may be viewed as a link between data access and data processing.

The declaration of a VO contains structure (which can be the same as for the usual actual object), source objects, mapping rules and so on. The source objects can be actual or VO or a mixture of both, thus a VO may be defined hierarchically. We list below a VO specification "partsreq", in the order processing example, where one of its source object "orderproc" itself is a VO.

```
Virtual_object_type
 parts_req (ORDER_ID, req(CODE, QTY))
 {
  structure  { ORDER_ID : c4,
               req (CODE, QTY)
                  { CODE : c4,
                    QTY  : i4.
                    key  : CODE
                  }
               key : ORDER_ID
             }
  source     { asmb_des, order_proc
             }
  mapping rules
    { parts_req (ORDER_ID, req(CODE, QTY)) :-
      order_proc (ORDER_ID, _, _, _, 0, _),  !, fail.

      parts_req (ORDER_ID, req(CODE, QTY)) :-
      order_proc (ORDER_ID, MODEL, _, ASMB_QTY, WAIT_QTY, _),
      asmb_des (MODEL, _, config(CODE_S,CODE_C,CODE_M,CODE_F)), !,
      parts_inv (CODE, _, AMT),
      (CODE = CODE_S; CODE = CODE_C; CODE = CODE_M; CODE = CODE_F),
      QTY is ASMB_QTY + WAIT_QTY - AMT,
      write (parts_req (ORDER_ID, req(CODE, QTY))), fail.
    }
 }
```

Unification is the way to make VO's physically accessible. In the case that a VO has virtual source objects, the latter must be instantiated in advance according to their own definitions, and such a process may spread upward in the virtual object network until all the actual and VO's involved are fully instantiated. Syntactically, the instantiation of a VO, such as "partsreq", is represented by the clause

```
mapping (parts_req).
```

The instance of a VO has a life time which extends over a single task. Duplicate unification for VO's in a task can be reduced unless its source objects are altered.

Although a VO mapping is just one or a cluster of rules, a VO is a more feasible entity than a rule. a VO has a structure definition, can be directly queried at a high level, with a printing format; It can, flexibly, either be used as a source object for other VO's, or be involved in tasks.

level 2 Heuristic Association

Although in general everything could be viewed as an object [Wilson 83], It is best to consider only the semantically meaniningful objects. In fact, the need to represent associative behaviors of a group of different type objects, without creating any "fuzzy" object, does arise in many situations. For this purpose we introduce a concept called "association", which describes the non-hierarchical, heuristic relationships among classes, typically represented by mc-methods.

In fact, an association neither introduces any new object, nor reffers to any storage structure, it only mentions the dynamic behaviors caused by a group of different types objects, and is more suitable to be interpreted as a schema, which provides a link from the problem space to the solution space.

Level 3 Task Modeling

The task modeling concerns not only the domain knowledge (about objects), but also the control structure (about scheme) whose description may be viewed as certain meta-rules, determining the usage of domain knowledge and the linkage of actions which make up an operational scheme with desired run-time control [Mel 83a].

Since the task scheme is basically a forward chaining network with a number of paths, our proposed task specification approach is founded on a rule-based linguistic net description formalism which was presented in author's another paper [Chen 85b], involving three basic kinds of components : action, linker and net.

In the object-oriented paradigm, actions are denoted by the messages (methods on objects) to be passed. An action specification gives the correspondence between the statically described message and the action as its execution. A message passing at different stage of a task must be identified separately, except in the case of a loop. There is a one-to-many mapping from the space of methods to the space of actions.

Linkers describe the conditions for each possible path in the net denoting the task scheme, that is, the connection rules. Each linker has four arguments (attributes), as linker (LINKERNAME, LINKMODE, TERMINATEMODE, CONDITION). The LINKMODE may be 'r' (regular) or 'c' (conditional branch). The TERMINATEMODE includes 'h' (hard termination with system state recovery, if errors are detected), and 's' (soft termination without state recovery).

A net definition specifies the internal linkage among the actions and the control flow (not data flow, as a rule, all inter-action communication must be carried out by accessing data/knowledge base, which may be viewed as "mailboxes").

The major feature of the proposed net specification approach consists in lifting net specification fron the structure level to the function level. This is characterized by introducing the concepts of net-structures and net forming operations called path forms for constructing new nets from existing ones, which map net-structures to net-structures in general.

A path form is an expression denoting a combination of net-structures which depend on the actions, linkers and sub-nets that form the parameters of the expression.

Examples of parth forms are

A > B (Composition),

A > (P)B (conditional composition through a linker P),

[A; B] (serialization),

(P){A; B; C} (conditional branch),

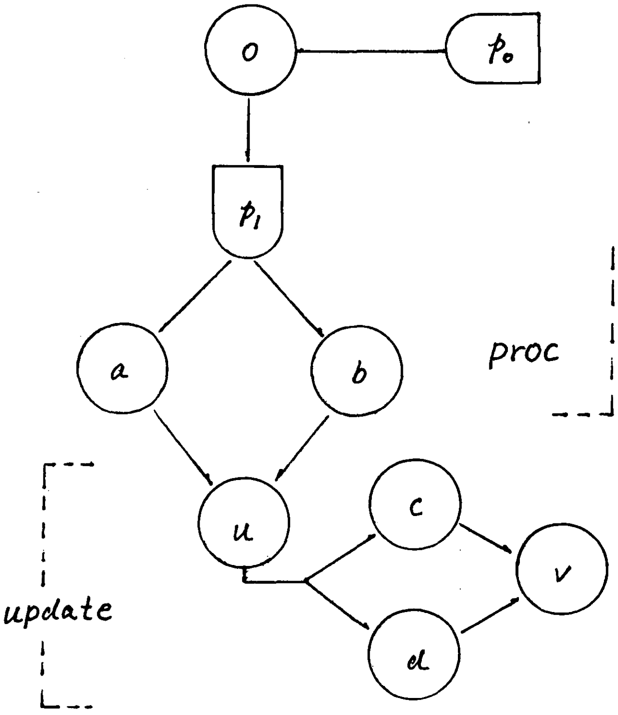
A >> {B; C; D} (compose-to-all, means [A>B; A>C; A>D]).

These path forms can be utilized hierarchically or recursively for representing a complex net as the combination of certain simpler (or lower level) nets. And in fact more additional path forms may be defined and implemented.

Loops are handled by a naming mechanism, the first-meet rule, describing how to handle a symbol for an action, a linker or a sub-net, which appears more than once in a net-expression, by going back to the leftmost position in the expression where the symbol appears.

To illustrate the proposed approach, we will give a complete rule-based task specification for the order processing example mentioned before, as following.

Figure 1.



```
task order_processing
  {
   object_specification
     { actual_objects  { order,products_inv,parts_inv,asmb_des,parts,machines
                       }
       virtual_objects { order_proc, parts_req, wk_ticket
                       }
       mc-methods  { update_inv (MESSAGE) :-
                      order_proc (ORDER_ID, MODEL, PICK_QTY, ASMB_QTY,
                                  WAIT_QTY, SALES_VALUE)),
                      asmb_des (MODEL, _, config(CODE_S,CODE_C,CODE_M,CODE_F)),
                      retract (products_inv (MODEL, NAME, AMT)),
                      NEW_AMT is AMT - PICK_QTY,
                      assert (products_inv (MODEL, NAME, NEW_AMT),
                      retract (parts_inv (CODE_S, NAME_S, AMT_S)),
                      retract (parts_inv (CODE_C, NAME_C, AMT_C)),
                      retract (parts_inv (CODE_M, NAME_M, AMT_M)),
                      retract (parts_inv (CODE_F, NAME_F, AMT_F)),
                      NEW_AMT_S is AMT_S - ASMB_QTY,
                      NEW_AMT_C is AMT_C - ASMB_QTY,
                      NEW_AMT_M is AMT_M - ASMB_QTY,
                      NEW_AMT_F is AMT_F - ASMB_QTY,
                      assert (parts_inv (CODE_S, NAME_S, NEW_AMT_S)),
                      assert (parts_inv (CODE_C, NAME_C, NEW_AMT_C)),
                      assert (parts_inv (CODE_M, NAME_M, NEW_AMT_M)),
                      assert (parts_inv (CODE_F, NAME_F, NEW_AMT_F)),
                      MESSAGE = invetories_updated.

                     update_order (MESSAGE) :-
                      order_proc (ORDER_ID, MODEL, PICK_QTY, ASMB_QTY,
                                  WAIT_QTY, SALES_VALUE)).
                      retract (order (ORDER_ID, MODEL, QTY, FINISH_QTY)),
                      NEW_FINISH_QTY is FINISH_QTY + PICK_QTY + ASMB_QTY,
                      assert (order (ORDER_ID, MODEL, QTY, NEW_FINISH_QTY)),
                      MESSAGE = order_updated.
                   }
       constraints { order with_cons {MODEL [ "hc40, hc60"}
                     products_inv  with_cons  {products_inv (MODEL, NAME, AMT),
                                               machines (MODEL, COST),
                                               COST * AMT < 500,000}
                   }
    scheme_specification
     { action    { o  is  mapping (order_proc).
                   a  is  mapping (wk_ticket).
                   b  is  mapping (parts_req).
                   u  is  update_inv (MESSAGE) : parts_inv(), products_inv(),
                                                 asmb_des (), order_proc ().
                   c  is  listing (parts_inv).
                   d  is  listing (products_inv).
                   v  is  update_order (MESSAGE) : order (), order_proc ().
                 }
       net       { order_processing   is  (p0) @proc > @update .
                   proc               is  o > (p1)[a; b] .
                   update             is  u > [c; d] > v .
                 }
       linker    { p0  is  (r, h, (order with_cons {QTY > 10})) .
                   p1  is  (r, h, (order_proc with_cons {ASMB_QTY > 0},
                                   parts_inv with_cons {QTY > 0})) .
                 }
     }
  }
```

We assume that all the anticipating objects are previously declared. Within this task, the virtual objects only need to be instantiated once (by "mapping"). In fact, the three virtual objects, holding the resulting data of the order processing task, can also be simply queried outside this task, just for estimation, without updating the system. This is an evidence showing the flexibility offered by our multiple level object/task modeling approach.

The net representation for the task is illustrated in Figure 1. Its performance can be described as follows: Action o, a, and b carry out the order processing in terms of computing virtual objects "orderproc", "workticket" and "partsreq", and convey the resulting data to users; actions u and v update the inventories and the order; action c, d display the new state of inventories after processing the current order. Certain control rules are specified at linkers p0 and p1.

TASKS AS SYSTEM OBJECTS

Tasks can be viewed as system objects as well, the meaning for this is two folds :

(1) In this approach, the "task" is treated as a special type of system objects with a set of operations defined on it, such as query, update, execute,...etc. Thus a knowledge based "Task Libraries" could possibly be built easily.

(2) A task may correspond to a cluster of special VO's which hold the resulting data of the task running.

The concept of Generalized Virtual Object (GVO) is developed for integrating action capabilities and operational dynamics, conveying therefore the execution results of complex decision/action schemes to end-users in terms of high-level data accessing.

Different from the approach for handling VO's, the GVO computing is characterized by active (vs passive) and dynamic (vs static) which may invoke any transactions, bring broad state changes and, on the other hand, be contributed by the feed-back effects of above state alterations. A process handling approach is also developed which can directly translate a net-oriented decision/action scheme making up a GVO mapping into an implementation.

CONCLUSIONS

Logic Programming and Functional Programming have set a bridge between AI and other fields [Chen 85a]. For developing more intelligent information systems, in this work we have combined object-oriented paradigm, logic programming, semantic modeling and event modeling, providing therefore complementary benifits in inference, deductive query support, integrity control, and explicit control knowledge representation, towards a generalized management of data, action and operational schemes.

REFERENCES

[Chen 86a] Q. Chen, "A Rule-based Object/ Task Modeling Approach", Proc. of ACM-SIGMOD 86 International Conference, Washington D.C. 1986, USA.

[Chen 86b] Q. Chen, "The management of Dynamically Distributed Data Base Windows", Proc. of International Conference on Very Large Data Bases (VLDB'86), 1986, Kyoto, Japan.

[Chen 85a] Q. Chen, "Extending the Implementation Scheme of Functional Programming System FP for Supporting the Formal Software Development Methodology", Proc. 8th International Conference on Software Engineering, London, 1985.

[Chen 85b] Q. Chen, "Toward A Generalized Data/Action Management : An Approach for Specifying and Implementing Operational Schemes", Proc. 1st Pan Pacific Computer Conference, Melbourne, Australia, Sep. 10-13, 1985.

[Cox 84] B. J. Cox, "Message/Object, An Evolutionary Change", IEEE Trans. On SE, pp. 50-61, Jan. 1984.

[Gall 84] H. Gallaire, J. Minker and J. Nicolas, "Logic and Databases : A Deductive Approach", Computing Surveys, Vol. 16, No. 2, 1984.

[Melk 83a] M. Melkanoff and Q. Chen, "An Experimental Database Which Combines Static and Dynamic Capabilities", Proc. of Engineering Design Applications, ACM-SIGMOD'83/Database Week, 1983.

[Melk 83b] M. Melkanoff and Q. Chen, "Integrating Action Capabilities into Information Databases", Proc. 2nd International Conference on Databases (ICOD-2), Cambridge, UK, 1983.

[Parker 84] D. Parker et al., "Logic Programming and Databases", Proc. Int. Workshop on Expert Database Systems, 1984.

[Ullman 85] J. Ullman, "Implementation of Logical Query Language for Databases", Proc. of ACM-SIGMOD 85, 1985.

[Zani 84] C. Zaniolo, "Object-Oriented Programming in Prolog", Proc. Int. Logic Programming Symposium, IEEE 1984.

A PLANT INTELLIGENT SUPERVISORY CONTROL EXPERT SYSTEM

Moonis Ali and Eddie S. Washington
Knowledge Engineering Laboratory
The University of Tennessee Space Institute
Tullahoma, Tennessee

ABSTRACT

A Plant Intelligent Supervisory Control Expert System (PISCES) is being developed in the Knowledge Engineering Laboratory at the University of Tennessee Space Institute (UTSI). PISCES is a knowledge-based system whose control strategy can be applied to most types of process control plants. PISCES will be employed at the power plant research facility located at UTSI. PISCES uses a modified form of backward chaining which allows the system to dynamically generate new top level hypotheses. PISCES control strategy also performs multiple lines of reasoning during hypothesis verification. Since PISCES must operate in a real time environment, knowledge base indexing utilizing discrimination nets have been implemented to increase system efficiency. PISCES also contains an explanation facility and rule editor that increases confidence in the system diagnostics and facilitates knowledge acquistion.

INTRODUCTION

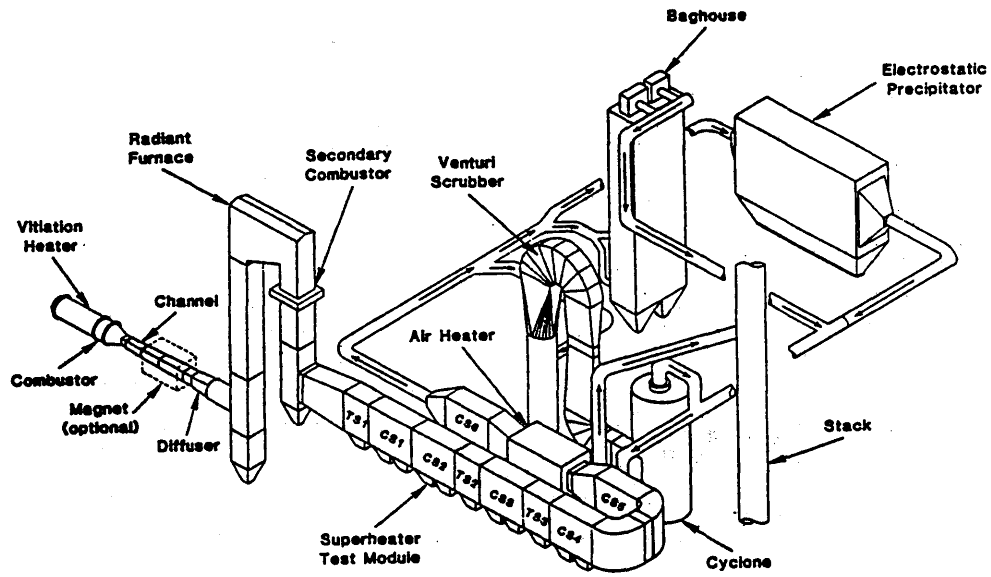
At UTSI we are developing an expert system called PISCES. PISCES is designed to provide an on-line aid in monitoring plant performance and in diagnosing plant system malfunctions in the United States Department of Energy's (DOE) Coal Fired Flow Facility (CFFF) magnetohydrodynamics (MHD) test facility located at UTSI. The Low Mass Flow (LMF) Test Train, located in the CFFF at UTSI, is an experimental magnetohydrodynamics (MHD) flow train designed for a thermal input of 28 MW. Fig. 1 is a schematic showing the current configuration of the flow train. Futher description of the facility is given in reference 3.

PISCES is a modified form of EX[1-2]. EX and PISCES are both rule based systems that use a modified form of backward chaining which allows the system to generate new top level hypotheses. EX was developed to perform fault diagnostics for setting oxidant flows during the vitiation heater ignition. The knowledge base of EX contained approximately 100 rules about the oxygen and nitrogen system flows, pressure, temperature and the oxidants delivery system control valves. PISCES will perform fault diagnostics and facility parameter monitoring not only during vitiation heater ignition, but also during ignition of the primary and secondary combustors. Once steady state operations is achieved, PISCES will monitor facility performance. Since PISCES will operate online, it is necessary that it provide real time response. To reduce system reponse time, the sequential data structure was replaced with an indexed structure which employs a discrimination net to implement the indexing. An additional improvement in response was accomplished by eliminating the need for a search in PISCES explanation facility. PISCES also provides a rule editor that allows knowledge acquistion while the system is operating.

PROBLEM DOMAIN

The architecture of PISCES includes plant monitoring and fault diagnostics in the domain of facility operation from checkout to secondary combustor ignition. The successful ignition of the CFFF

CH2345-7/86/0000/0207$01.00 © 1986 IEEE



LMF4 FLOW TRAIN

FIG. 1

secondary combustor requires that the correct sequence of procedures and events be followed. Before operation of the facility can begin, certain pre-operational and operational procedures must be performed on the facility and test train components. After these have been accomplished, the vitiation heater (oxidant pre-heater) must be ignited. This is followed by the ignition of the primary combustor on fuel oil and then coal. During each step in the sequence certain conditions and selected parameter values must be satisfied before the next sequence in the chain of events can begin.

Operation of the facility begins with the heating of the radiant furnace to operating temperature using industrial gas burners. This prevents thermal shocking of the furnace refractory. Once the furnace is sufficiently heated, ignition of the vitiation heater is accomplished by first setting nitrogen flow followed by oxygen flow. When the oxidants flows have been established the fuel oil flow to the vitiation heater burner plate is initiated. Ignition is accomplished using a spark igniter. After successful ignition of the vitiation heater, fuel oil is introduced to the primary combustor. This oil ignites in the presence of the 1500 °F oxidants. Next, coal flow to the primary combustor is started. The ignition of the secondary combustor requires the proper test conditions to be set which results in a stoichiometric value of 0.85 . It is in this domain in which PISCES will be used. During each phase of facility operation, if PISCES determines that a parameter is outside of its normal operating range or an event does not occur in the proper sequence, PISCES will display a message and attempt to determine the cause and possible solution to the problem. PISCES also allows the plant operators to request assistance in solving problems.

KNOWLEDGE BASE

A PISCES knowledge base has been developed on the basis of the knowledge acquired from three MHD engineers. One of the experts was the project leader for the design and fabrication of the CFFF. The other two experts are in charge of test operation for the CFFF. Written procedures and component operation manuals have also been used as a source of knowledge about the domain of PISCES. There are currently 200 rules in the knowledge base. They contain information about the proper sequence of events, operational procedures, the nominal ranges of parameter values and the characteristic behavior of the test train components up to and including the radiant furnace. A typical rule in the knowledge base contains IF and THEN clauses. An optional ELSE clause can also be contained in a rule. The IF clause contains the antecedents of the rule. The THEN clause contains the consequents of the rule. The ELSE clause contains the alternates to the THEN clause. If the antecedents are true, the THEN consequents are asserted to be true. However, if the antecedents fail, the ELSE clause triggers a multiples lines of reasoning to be pursued. Examples of rules with and without the optional ELSE clause are shown in Figures 2 and 3. Each of the clauses may contain one of the following:

1. A pseudo English representation of some known condition.
2. An assertion that needs to be verified before a conclusion can be drawn.
3. A LISP function to be evaluated.

With each THEN or ELSE clause a certainty factor is associated. The certainty factor represents the expert degree of confidence in the conclusion drawn if antecedents are known with absolute certainty (certainty factor equals 1.0). Positive numbers indicate belief and negative numbers indicate disbelief. When the antecedent's themselves are not known with absolute certainty, a new value of certainty factor is computed by employing an approach similar to MYCIN[4]. The examples presented in Fig. 4 and 5 from a typical session compares PISCES finding when antecedents are known with certainty and when they are not.

```
(rule 1
   (if   (LISP prog ()
   (cond  (N2-sensors-polled  (return  t))  )
   (setq  N2-sensors-polled  t)
   (write  T  '/Open  N2  isolation  valve/)
   (write  T  '/Set  N2  control  valve/)
   (write  T  '/Polling  sensors/)
   (return  t))
   (N2  flow  setting  succeeds)
   (LISP  prog  ()
   (cond  (O2-sensors-polled  (return  t))  )
   (setq  O2-sensors-polled  t)
   (write  T  '/Open  O2  isolation  valve/)
   (write  T  '/Set  O2  control  valve/)
   (write  T  '/Polling  sensors/)
   (return  t))
   (O2  flow  setting  succeeds))
   (then     (1.0  ignition  succeeds))
   )
```

Fig. 2 Typical Rule without Optional ELSE Clause

```
(rule 4
   (if
   (LISP  and  (<  N2_flow  1.61)
          (>  N2_flow  1.31)
   (not-  '(ignition  succeeds))))
   (then     (0.9  N2  flow  is  normal))
   (else     (-1.0  N2  flow  is  normal)
   (LISP  check
   (N2  flow  is  not  normal)
   (N2  isolation  valve  is  closed)
   (N2  differential  pressure  normal)
   (N2  temperature  is  normal)
   (N2  pressure  is  normal)
   (N2  control  valve  is  not  working  right)
   (N2  flow  controller  is  not  working
right)
   (N2  flow  calculation  not  correct)
(MACSYM  N2  flow  calibration  curve  is
bad)))
   )
```

Fig. 3 Typical Rule with Optional ELSE Clause

```
enter hypotheses
? (O2 differential pressure sensor
was recalibrated)
Is this true:
o2 differential pressure sensor
was changed recently? y
Rule r00092 deduces
[4]   o2 differential pressure
sensor   was recalibrated with
certainty of 0.96

enter hypotheses
```

Fig. 4 antecedents known with absolute certainty

```
? (O2 differential pressure sensor
was recalibrated)
Is this true:
o2 differential pressure sensor
was changed recently? 0.8
Rule r00092 deduces
[2]   o2 differential pressure
sensor was recalibrated with
certainty of 0.768
```

Fig. 5 antecedents known with certainty of 80%

During backward chaining, EX[1-2] performed sequential searching which resulted in slow system response. In order to minimize search time, PISCES replaced the sequential knowledge structure with an indexed knowledge structure. The rules are currently indexed on the syntatic structure of the individual members of the THEN clause. PISCES uses a discrimination net to attach properties to arbitary expressions. Encoding the information represented in the THEN clause using this scheme allows us to attach to each unique member of any THEN clause the list of rule numbers in which it is contained. Each element in the THEN clause is used as a link in the discrimination tree. When searching for a rule that contains the hypothesis "*(O2 pressure sensor failed)* " in its THEN clause, each link is followed until we either reach a terminal node that contains a list of all the rules with this hypothesis in its THEN part, or we discover that there is no node with a transition on the current element. These two possibilities are illustrated in Fig. 6. The advantage of this method over the sequential search is that the search time is significantly reduced. In a sequential structure the entire knowledge base must be searched to retrieve the rules that relate to an hypothesis. However, in PISCES knowledge structure a traversal of the discrimination net will retrieve all the rules that apply to the hypothesis. If no path through the net can be found the algorithm will terminate again eliminating the time wasted in unsuccessful searches in sequential structures. The knowledge acquistion algorithm works the same way, except that instead of returning failure, a path is created to a new terminal node for this rule. The algorithm also checks for rule duplication when new rules are added. If the rule is a duplicate it is not added to the knowledge base. The rule itself is stored on the property list of its rule number.

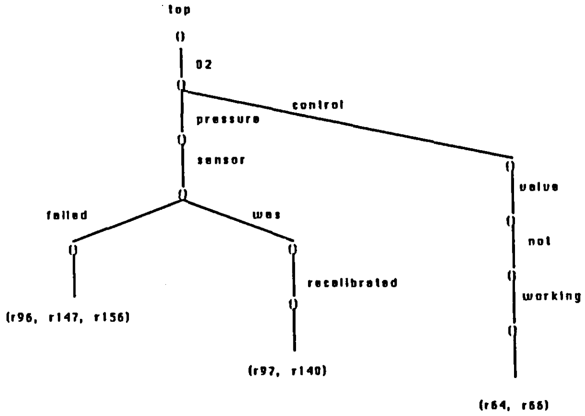


Fig. 6 Simple Discrimination Net

CONTROL STRATEGY

PISCES has a top level loop that manages the systems resources. This top level loop provides the lower level functions of synchronization and communication. The low level functions provide parameter checking, handling of keyboard interrupts, goal verification and knowledge base maintenance. If the monitor function detects an abnormal condition, the top level calls the correct function that generates a goal to determine the cause of the fault. A user request for a goal verification can also be entered from the keyboard.

PISCES uses modified backward chaining that allows multiple lines of reasoning to verify an hypothesis. In traditional backward chaining a list of one or more hypotheses is verified by using the following strategy:

1. See if the current hypothesis is already a known fact.
2. If it is known, it is verified. Therefore return SUCCESS.
3. Otherwise, find rules in your knowledge base that contain the hypothesis in their consequent clause.
4. Then use the antecedent clause of these rules to generate a new list of hypotheses.
5. If no rule can be found, ask user.
6. Now repeat the above steps starting at one.
7. If all rules have been exhausted and the current hypothesis has not been verified, return FAILURE.

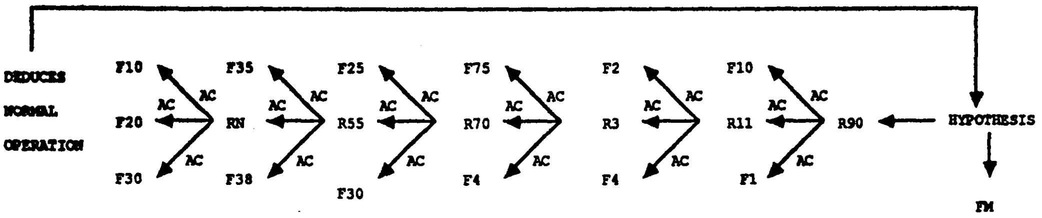
PISCES differs from the traditional backward chaining, in that if it fails to verify an hypothesis, it can, at the request of the current rule, attempt to determine the reason for the failure of the hypothesis. This is accomplished by adding the following steps:

8. Upon failure to verify the hypothesis, if an ELSE clause is part of the rule, use it to generate an alternate list of hypotheses
9. If all the rules have been exhausted and the alternate list of hypotheses have not been verified, return FAILURE.

Shown in Fig. 7 is the flow of control in a traditional backward chaining system. As can be seen from the diagram shown in Fig. 7, a system using traditional backward chaining can report success only on successful verification of the original hypothesis. For example, the hypothesis requiring verification is contained in the consequent clause of rule R90. The antecedent clause of rule R90 contains a sub-hypothesis that can be verified by rule R11. As illustrated in Fig. 7, rules R3, R70, R55 and finally rule Rn must be successfully executed before the original hypothesis is verified. If at any sub-level in the verification process a sub-hypothesis fails, the single line of reasoning approach must report failure and give up. However the diagram shown in Fig. 8 illustrates the ability of PISCES to dynamically generate new top level hypotheses. If rule R1 fails to verify, PISCES will use the knowledge contained in the rule R2's else clause to generate new top level hypotheses (multiple lines of reasoning) to determine the cause of the failure of the original hypothesis. Also shown in Fig. 8 is PISCES' ability to determine the relationship between a fault and it's effects from any of the following cases:

Single Fault -> Single Effect
Single Fault -> Multiple Effects
Multiple Faults -> Single Effect
Multiple Faults -> Multiple Effects

In PISCES multiple lines of reasoning are usually pursued due to an unsuccessful verification of an hypothesis involving the normal operating condition of a component or the normal operating range of a parameter value. The rule shown in Fig. 9 would be used to ascertain whether the nitrogen flow differential pressure value was in the correct range. Rule 9 contains one conditional clause that requests a LISP expression to be evaluated. This expression checks the range of the nitrogen differential pressure. If the expression evaluates to true, the fact "N2 differential pressures is normal" would be asserted. But, if the expression evaluates to false, then the fact "N2 differential pressures is not normal" would be asserted. This is indicated by the negative 1.0 certainty factor of the first clause in the ELSE part.



HYPOTHESIS : IGNITION SEQUENCE SUCCEEDS SUCCESSFULLY

Fi REPRESENTS EITHER A FACT IN KNOWLEDGE BASE OR SENSOR VALUE

Ri REPRESENTS THE i-th RULE

Fig. 7 Control Strategy with Single-line of Reasoning

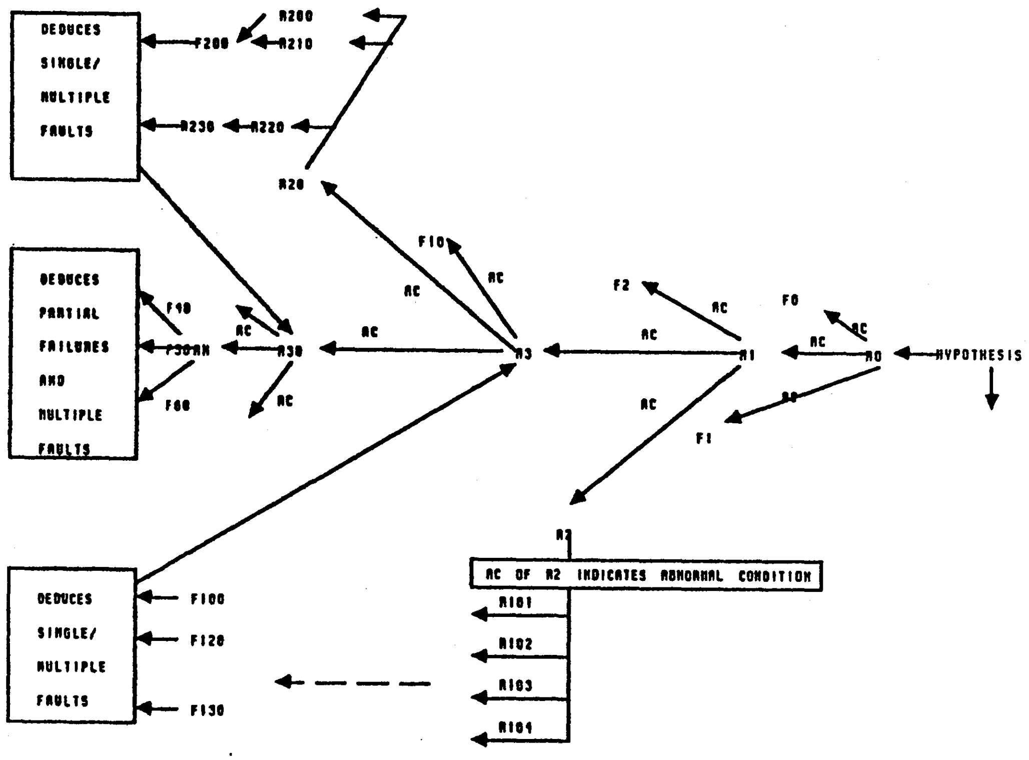


Fig. 8 Control Strategy With Multiple Lines of Reasoning

```
(rule 9  (if
   (LISP  and  (<  N2_differential_pressure
                30)
   (plusp  N2_differential_pressure)))
   (then    (0.95  N2  differential  pressure
             is  normal))
   (else    (-1.0  N2  differential  pressure  is
          normal)
   (LISP  check
   (N2  differential  pressure  is  not
      normal)
   (N2  isolation  valve  is  closed)
   (N2  tank  is  empty)
   (N2  differential  pressure  sensor  failed)
   (N2  pipe  is  leaking)))
   )
```

Fig. 9 Example of Multiple Lines of Reasoning

The terminal session presented in Fig. 10 illustrates the process PISCES uses to determine the root cause of the abnormal nitrogen differential pressure. When PISCES is requested to verify that nitrogen differential pressure is normal, it finds that the parameter value is out of range. It then asserts that the parameter value is not normal. Then the ELSE clause is executed and the LISP function activates multiple lines of reasoning to determine the cause of the fault.

```
enter hypotheses
? (N2 differential pressure is
normal)
Rule r00009 deduces
[1]    n2 differential pressure is
normal with certainty of -1.0
```

Fig. 10a:

Here, the operator query PISCES as to whether nitrogen differential pressure is normal. PISCES using rule 9 determines that· the nitrogen differential pressure is not normal.

```
Is this true:
n2 tank pressure gauge indicates very
low pressure? n
Rule r00013 deduces
[3]    n2 tank is not empty
with certainty of 1.0

Is this true:
n2 differential pressure sensor
output is greater than 5 volt? n
Is this true:
n2 differential pressure sensor
output is 0 volt? y
Rule r00020 deduces
[5]    n2 differential pressure sensor
failed with certainty of 0.85

n2 differential pressure is not
normal because n2 differential
pressure sensor failed
```

Fig. 10b:

The failure to verify the normal operating condition causes PISCES to follow the alternate line of reasoning contained in rule 9 ELSE clause. In the above dialog PISCES questions the operator about information it needs to aid in determining the root cause of the fault. Once PISCES determines the cause of the fault, it reports to the operator its findings.

EXPLANATION FACILITY

During the course of a session it is not uncommon for the operator to inquire as to the rationale behind the current line of questioning or to ask why a particular conclusion was reached. To provide the capability for dialog between the operator and PISCES, the explanation facility of EX[1-2] was incorporated into PISCES and was essentially unchanged with one exception. EX attaches each assertion it makes to a unique number. Later if the operator requests an explanation or justification of a particular assertion by entering a statement number, EX must search the knowledge base to find the rule containing the statement. PISCES avoids this search by not only attaching the assertion to the statement number, but the number of the rule used to justify it. In the example of the terminal (shown in Fig. 10) PISCES asks the operator whether the N2 tank pressure gauge indicated a very low pressure. At this point in the dialog the operator could ask PISCES why it needs to know this information. This exchange is shown in Fig. 11. For a more detailed description of the explanation facility see reference [1-2].

```
enter hypotheses

    ? (N2 differential pressure is
    normal)
    [1]    n2 differential pressure is
    normal with certainty of -1.0

    Is this true:
    n2 tank pressure gauge indicates very
    low pressure? why

    Because knowing whether n2 tank
    pressure gauge indicates very low
    pressure will help determine whether
    n2 tank is empty
```

Fig. 11a:
Here the operator wants to know the relationship between the nitrogen tank gauge reading and the assertion that nitrogen differential pressure is not normal.

```
Rule r00013 deduces
[3]    n2 tank is not empty
with certainty of 1.0

Is this true:
n2 differential pressure sensor
output is greater than 5 volt?

how 3

[i.e., how do I know that 1.0 n2 tank
is not empty?]

It is known that...
lisp not- '(n2 tank is empty)

and rule r00013 says that
if
lisp not- '(n2 tank is empty)
then
1.0 n2 tank is not empty

therefore...
1.0 n2 tank is not empty
```

Fig. 11b:
At this point PISCES has deduced that the nitrogen tank is not empty and is persuing the line of reasoning that the sensor may be faulty. However, the operator would like to know the rationale for the assertion that the tank is not empty. PISCES responds that knowledge base rule 13 says that if it cannot be proven that the nitrogen tank is empty then it not empty.

RULE EDITOR

A simple rule editor is provided with PISCES. It provides a parentheses free interface for adding rules to the knowledge base. The editor also performs syntax checking during rule input. The keywords IF, THEN, ELSE and END are used to direct the action of the editor. A sample session is presented in Fig. 12.

```
enter rule in the form of :
if
.... <if clauses> ....
then
.... <then clauses> ....
else
....<else clauses> ....
end
```

Fig. 12a Structure of PISCES rule

```
if
vh temp profile normal
flame detector normal
combustor pressure normal
then
combustor performing well
else
combustor not performing well
end

do you wish to input another rule
(y/n)

? n
```

Fig 12b:
To input a rule the operator first enters the keyword "if" followed by the rule antecedents. Next enter the keyword "then" which is followed by the consequents. The "else" and the else clause are optional. Upon completion of the rule the operator must enter end.

CONCLUSION

In this paper we have presented a description of PISCES. PISCES is a rule-based system that uses an English-like syntax to converse with the operator and represents knowledge about the domain of its application. PISCES has the ability to explain its action or to request more information in attempting to reach a solution. PISCES has been implemented in a computer simulation environment to prevent interruption of the MHD plant testing schedule. Prior to installing PISCES on the MHD plant computer the current syntatical indexing scheme will be replaced with a semantic indexing scheme. Currently, the syntactic indexing requires the operators to specify hypotheses exactly as they are stored in the Knowledge base, in order to request assistance from the system. In the next version of PISCES we will implement a natural language interface that transforms user queries into the internal representation used by the system.

REFERENCES

1. Ali, M., and Scharnhorst D. A., "An Expert System for Power Plant Monitoring and Diagnostics," The 1985 ASME International Computers in Engineering Conference. Aug. 1985.
2. Moonis Ali, and Dean Scharnhorst and Shan Chi, "EX: An Expert System for Power Plant Management," Proceedings of the Conference on Applied AI and Knowledge-Based Expert Systems, Nov. 1984.
3. Moonis Ali, and Eddie S. Washington, "PISCES: An Intelligent Computer Aid to Power Plant Operators," Wattec 86: 13th Annual Energy Conference, Feb. 11-14, 1986, Knoxville, Tennessee.
4. Shortliffe, E. H., *Computer-based Medical Consultation: MYCIN*. New York: American Elsevier, 1976.
5. Charniak, E., Riesbeck, C. K., and McDermott, D. V., *Artificial Intelligence Programming*. New Jersey: Lawrence Erlbaum Assoicates, 1980.

Knowledge-Based Layout Design System for Industrial Plants

Kenichi Yoshida * , Yasuhiro Kobayashi * , Yoshikatsu Ueda ** ,
Hideo Tanaka *** , Shouichi Muto *** and Junichi Yoshizawa ***

* Energy Research Laboratory, Hitachi Ltd.
1168 Moriyamacho, Hitachi, Ibaraki 316, Japan
** Systems Engineering Division, Hitachi Ltd.
6,Kanda-Surugadai 4 chome, Chiyoda-ku,Tokyo 101, Japan
*** Engineering Research Center, The Tokyo Electric Power Co.,Inc.
2-4-1 Nishi-Tsutsujigaoka Chofu-city,Tokyo 182, Japan

ABSTRACT

A knowledge-based method is proposed for the layout design of industrial plants and applied as a layout design system for an electric power substation. In this method, dependency directed backtracking is adopted as the means to realize efficient modifications of intermediate layout plans while maintaining consistency among the data of the modified plan. The layout design system developed for the substation automatically generates layout plans which satisfy constraints and evaluates them as to construction cost, noise level, and other performance factors.

1. INTRODUCTION

A knowledge-based method is proposed for the layout design of industrial plants and applied as a layout design system for an electric power substation. The substation is an industrial plant which converts high voltage electric power to low voltage power. In designing a substation, one of the key phases is the layout of those components at the plant site. A substation is composed of hundreds of electrical, mechanical and architectural components. Layout designers are faced with the problems of treating the various attributes of these components, the spatial relation between them, and the spatial relation between them and the plant site when determining a layout plan. This complicated design process has been partially computerized to assist designers in this area.

Conventional computer-aided design (CAD) systems for the layout of a substation support the drawing of a layout plan on graphic terminals and/or numerical calculations to evaluate the construction cost, noise level from transformers, etc. [1] Few efforts, however, have been carried out to support the generation of layout plans which is essential to layout design. Ideally this facet of design should also be supported by a CAD system to enhance productivity in the layout design for industrial plants. It is, however, necessary for a CAD system to use designers' expertise directly in order to support the layout plan generation.

Knowledge-based design systems have been proposed in the field of engineering design to realize the direct utilization of experts' knowledge. [2]-[4] Emphases are placed on the use of heuristic knowledge in declarative form and the simple inference engine in those application systems, though a numerical procedure separate from the inference mechanism is also employed in the works of Refs. [3] and [4].

Expert designers, for example, use their heuristic procedures to solve the layout problem efficiently in a conventional design method for a substation. It is important for a layout design system to have intelligent functions which simulate designers' procedures to realize efficient problem-solving process. Hardly any research efforts, however, have been directed to their embodiment.

The objectives of this study are to develop a knowledge-based method to realize the most efficient design process by the use of designers' procedures and apply it to a layout design system for a substation.

2. METHOD OF INDUSTRIAL PLANT LAYOUT

2.1 Experts' Knowledge for Industrial Plant Layout Design

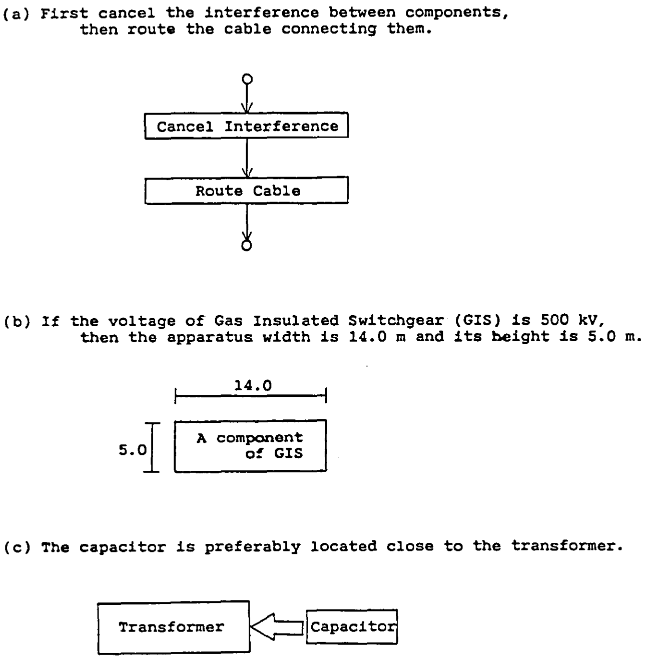
In the industrial plant layout design, designers determine the layout of the various components at the plant site. The layout design of a substation is chosen as an example to examine the characteristics of knowledge and its utilization in the design process.

Analysis of the layout process of expert designers gives the following characteristics.

(1) The use of declarative and procedural knowledge

Three typical examples of knowledge which are used in the design process are illustrated in Fig.1. Fig.1(a) shows knowledge about the sequence of two steps to realize an efficient layout process. This is an example of procedural knowledge. Fig.1(b) shows

CH2345-7/86/0000/0216$01.00 © 1986 IEEE


(a) First cancel the interference between components, then route the cable connecting them.

(b) If the voltage of Gas Insulated Switchgear (GIS) is 500 kV, then the apparatus width is 14.0 m and its height is 5.0 m.

(c) The capacitor is preferably located close to the transformer.

Figure 1. Examples of Domain Specific Knowledge

declarative knowledge about the size of GIS (Gas Insulated Switchgear). Fig.1(c) shows declarative knowledge about the relation between the location of a transformer and that of a capacitor.

(2) The efficient modification of an intermediate plan

In the design process, decisions made at an early phase are not necessarily correct later. It is necessary, then, to modify the previous decisions while maintaining the consistency among the data of the modified layout. It is, for example, difficult to predict if a component being placed now will interfere with the components placed in the future. Designers are able to modify the location of components and correct the layout plan efficiently.

(3) The preparation of design alternatives

Design criteria of various types are available for the layout design of a substation. Differences in the initial conditions result in different plans, as do different criteria. Designers often prepare several layout plans based on different sets of design criteria and compare them to select the optimal plan for overall performance.

These characteristic functions are related to designers' expertise by which expert designers can solve the layout problem efficiently. It is, therefore, necessary to embody these functions in the layout design system in order to improve productivity of the layout design of industrial plants. The individual methods to realize these functions in the knowledge-based layout design system are outlined in the subsequent sections.

2.2 Knowledge Representation

2.2.1 Procedural knowledge

Procedural knowledge is represented by a functional module, which is a set of rules together with the control information to use them. Figure 2 shows example representations of procedural knowledge. The functional module shown in Fig.2 is the representation of the knowledge shown in Fig.1(a), and is a procedure for routing the cable between components. The control information attached to this procedure is "(and rules)". This means "Rules in this module will be evaluated sequentially". When this module is invoked, first the procedure to adjust the interference between components is used, then that to plan the cable routes is used. The statements "(infer 'adjust-interference)" and "(infer 'make-cable-main)" causes activation of these functional modules.

The general design process is also controlled by the functional modules. The inference engine first invokes the modules which make an outline of the layout plan, and then invokes those which provide the details.

```
; Following statements define the procedure
;       for routing the cable.
(module make-cable

        ; Rules in this functional module
        ;       will be used sequentially.
        (and rules)

        ; if the interference between components
        ;       are not canceled yet,
        ; then cancel it.
        (if (not (eq 'adjusted layout.interference))
            (infer 'adjust-interfere))

        ; if the interference between components are canceled,
        ;    and the cables are not routed yet,
        ; then route cable.
        (if (and (eq 'adjusted layout.interference)
                 (not (eq 'made layout.cable)))
            (infer 'make-cable-main))
        )
```

Figure 2. Example of Functional Module

2.2.2 Declarative knowledge

The declarative knowledge is used to determine the layout of components at a specific layout step. This kind of knowledge is represented by two types of methods; functional modules and demons. Demons are procedures that execute whenever a particular condition about the data in the working area becomes satisfied.

The functional module represents a procedure to be executed in a specific situation of the layout process. For example, the knowledge shown in Fig.1(c) is not usable until locations of the main components, such as transformers and GIS are defined.

The demon is used to represent knowledge which is invariable in the design process, such as values of common specifications of components. A typical example is the dimensions of the transformer. The knowledge shown in Fig.1(b) is represented by a demon as shown in Fig.3. The inference engine always watches when the demon in the knowledge base is activated. If the activation condition of the demon is satisfied, the demon is invoked and a change is made in the data of the layout plan. For example, if a designer specifies the voltage of a GIS as 500 kV, the activation condition of the demon shown in Fig.3 is satisfied, and the width of the GIS is set as 14.0 m.

```
; Following statements define attribute of
;       a component of gas insulated switchgear (GIS)
(class  GIS
        ...
        (demon  ...

                ; if voltage of $ ( one of GIS ) is 500 k Volt,
                ; then width of it is 14.0 m
                ;                     and height of it is 5.0 m.
                (if (eq 500K $.volt)
                    (and (:= $.width 14.0)
                         (:= $.height 5.0)))
                ...
                )
        ...
        )
```

Figure 3. Example of a Demon

2.3 Modification of Layout

In the layout design process, it is necessary to modify a previous decision and maintain the consistency among the data in the modified layout plan. It is often difficult to predict the impact of a local modification on another part of the layout plan, for which a secondary modification is required to maintain consistency for the primary modification. Some efficient mechanism to backtrack and regenerate the layout plan should be added to an inference engine to realize efficient modification of an intermediate layout plan, while maintaining consistency among the data of the modified plan.

The dependency directed backtracking can be a mechanism to modify the layout plan and to adjust side effects of the modification. Few attempts have been done to use it as a means to keep data in an intermediate plan consistent when the layout plan is modified, though the technique of dependency directed backtracking has been experimentally studied for other purposes. [5]-[8] The modification mechanism which requires minimum data deletion is developed on the basis of the dependency between data to describe a layout plan.

Figure 4 shows a simplified example which illustrates the efficient modification function. In Fig.4(a), transmission towers are placed on the upper and lower sides of the center line of the transformer facility on the basis of knowledge on the spatial relation between transmission towers and the facility. Their locations depend directly on the location of the transformer facility. After the transformer facility is placed, a noise evaluation program written in Fortran is invoked and the result reveals that the noise level at a residential area exceeds the limitation. In this situation, the inference engine invokes a functional module which finds an alternative facility location to satisfy the constraint on the noise level. This procedure is stored in the knowledge base, and identified as procedural knowledge to adjust the spatial relation between the location of the residential area and that of the transformer facility. Fig.4(b) describes the situation after the facility location is modified. Now, the transmission tower locations are inconsistent, no longer being on the center line. This is the side effect caused by the primary modification of the layout plan.

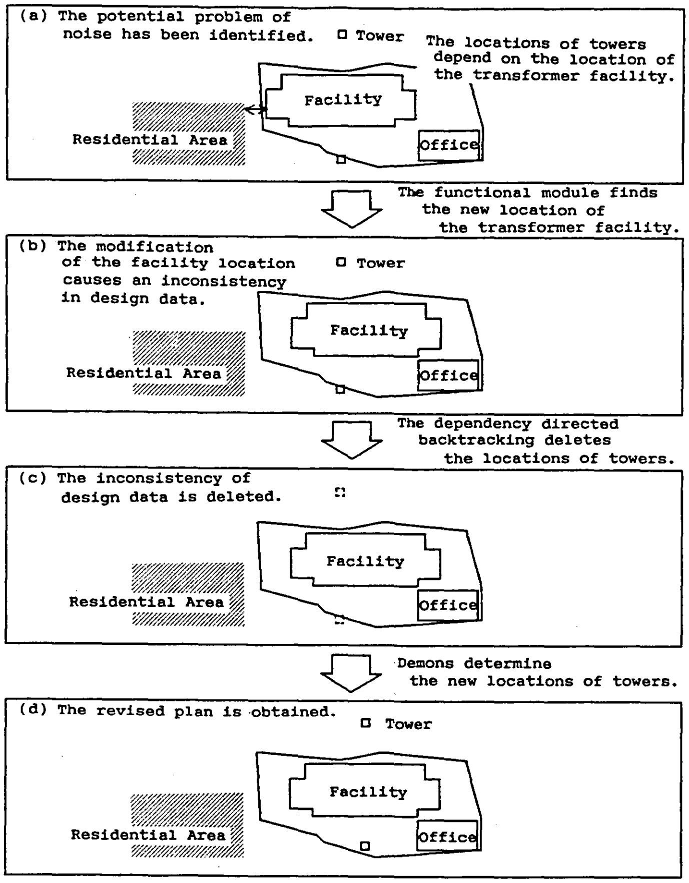

(a) The potential problem of
noise has been identified.
□ Tower
The locations of towers depend on the location of the transformer facility.
Facility
Residential Area
Office
The functional module finds the new location of the transformer facility.
(b) The modification of the facility location causes an inconsistency in design data.
□ Tower
Facility
Residential Area
Office
The dependency directed backtracking deletes the locations of towers.
(c) The inconsistency of design data is deleted.
Facility
Residential Area
Office
Demons determine the new locations of towers.
(d) The revised plan is obtained.
□ Tower
Facility
Residential Area
Office

Figure 4. Example of Layout Modification

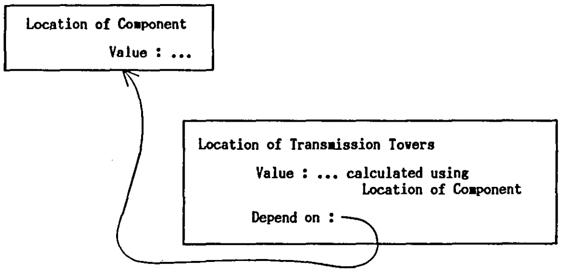
The secondary modification is done by the dependency directed backtracker and demons to adjust the undesirable side effect and to make data of the layout plan consistent. The backtracker deletes the locations of transmission towers that depend on the invalid location of the transformer facility from the working area, on the basis of the information about the dependency between the locations of components. Fig.4(c) shows this situation.

Finally, the demon is invoked to determine the locations of transmission towers. Their modified locations are shown in Fig.4(d).

The characteristics of this method to modify the layout plan are listed below.

(1) The procedure to find an alternative location of the transformer facility requires a limited search space, because it efficiently uses information about what the problem is and knowledge about how to solve it.

(2) In the working area, values of data are stored with information about the dependency between data. Figure 5 shows the dependency information of a transformer facility and its transmission towers. After the procedure modifies the layout plan, the dependency directed backtracker makes the data in the working area consistent.



Location of Component
Value : ...
Location of Transmission Towers
Value : ... calculated using Location of Component
Depend on :

Figure 5. Dependency Information

(3) If the dependency directed backtracker deletes invalid data from the working area, demons are used to reproduce the data making them compatible with the new modified data.

This mechanism, embedded into the inference engine, constitutes a functional module to find alternative locations of the transformer facility simply, because the functional module to modify the layout plan does not need to be concerned with both the existence and consistency of the data in the working area.

2.4 Alternatives to Layout Plan

It is often essential to obtain design alternatives of a layout plan in the course of design optimization. The data for different layout plans should be handled to obtain systematically several design alternatives.

Figure 6 shows the structure of a knowledge base and working area to support this function. The knowledge base includes two kinds of knowledge, and the working area includes two kinds of temporary data. The knowledge base stores knowledge to control the search process of the alternatives, and to generate layout plans from different initial conditions. The knowledge for layout plans is utilized to complete the individual layout plan, and the knowledge for control is utilized to select the next alternative layout plan to be processed.

To maintain several layout plans simultaneously, the data for layout plans are structured in the working area. The working area stores both the data to represent the layout plans and the data to describe their content. In Fig.6, three different layout plans are connected to different configurations of the main components.

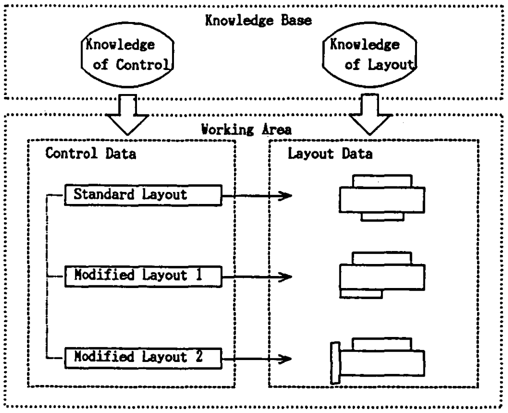


Figure 6. Alternative Layout Plans

3. SUBSTATION LAYOUT SYSTEM : XL-S

3.1 Configuration of XL-S

The demonstration system XL-S (eXpert system for Layout design for Substation) for layout design of a substation is based on the knowledge-based method described in the previous chapter.

The XL-S configuration is depicted in Fig.7. The system is composed of four elements; knowledge base, working area, inference engine and graphic interface. The knowledge base stores declarative and procedural knowledge in the form of functional modules and demons. The working area temporarily stores intermediate or/and final results of several layout

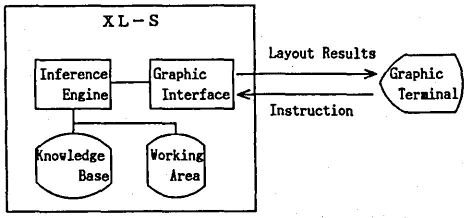


Figure 7. Configuration of Layout Design System XL-S

plans and control data. The inference engine drives the design process by forward chaining, generates hypothetical layout plans, and modifies intermediate layout plans. Through graphic interface, the substation layout is displayed on the graphic terminal and designer's instructions are sent to the inference engine.

The general flow of the layout process with XL-S is depicted in Fig.8. In the first step, the designer inputs the substation specifications such as the input voltage and capacity. In the second step, the system proposes an incremental change in the layout plan. Typical examples of these changes are the addition of a component to the layout space and the evaluation of a design parameter. The system assembles component areas and determines their placement using a standard layout process. The designer can intercept and modify the layout plan which is being incrementally made by XL-S at any layout step. If the proposed change in the layout plan satisfies the predefined constraints and is judged to be acceptable, the system continues to propose additional changes in the layout plan through the confirmation loop.

XL-S memorizes the default component size and the default relation between components in the knowledge base to minimize the amount of input data from the user. The designer thus handles fewer input data in this system than in a conventional graphic CAD system used for drafting. The system detects violations of constraints and initiates functional modules to remedy them. The designer can also directly specify modifications of the layout plan. Any modification invokes the dependency directed backtracker and demons to assure consistency among the data in the modified layout plan. With this automated function, the designer is relieved of the burdens of maintaining a consistent layout plan in the working area, and reproducing necessary data.

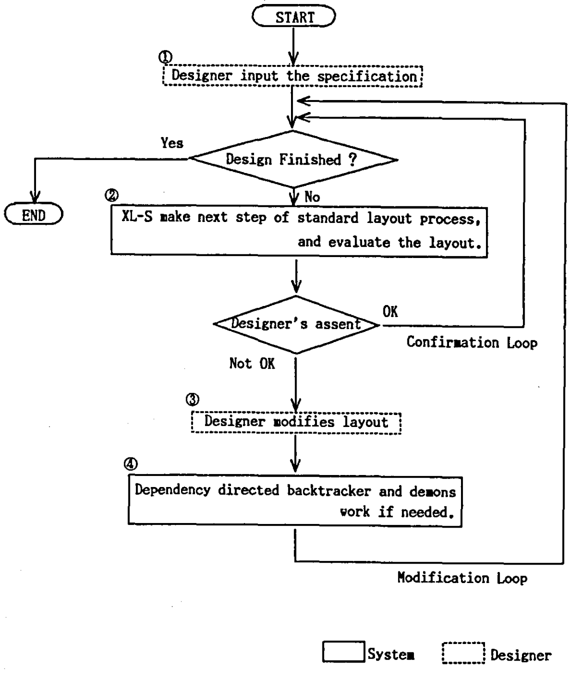
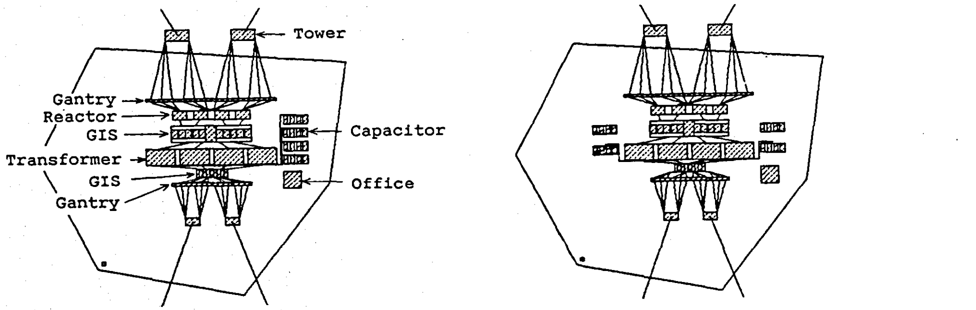

START
① Designer input the specification
Design Finished ?
Yes
END
No
② XL-S make next step of standard layout process, and evaluate the layout.
Designer's assent
OK
Confirmation Loop
Not OK
③ Designer modifies layout
④ Dependency directed backtracker and demons work if needed.
Modification Loop
System
Designer

Figure 8. Layout Process with XL-S

3.2 Results

The layout design system XL-S was applied to the layout design of a typical substation. The input and output voltages of the substation are 1000 kV and 500 kV respectively, and its total capacity is 8000 MVA. The main components are gantries, reactors, GIS's, transformers, capacitors, transmission towers, and an office building. A typical result from this system is shown in Fig.9(a).



(a) Results of a Layout Plan (b) Results of a Modification Specified by a Designer

Figure 9. Examples of Layout Plans

XL-S generated this layout plan mainly on the basis of knowledge on the spatial relation between components. The system also evaluated the layout plan as to the construction cost, noise level and other performance factors.

Fig.9(b) shows a modified layout plan based on that shown in Fig.9(a). In this case, the location of the four component areas which include a capacitor are modified directly by the designer. The total number of data that are modified is 12. This primary modification requires a secondary modification of related components. Modification of even limited data can cause a large-scale change in the layout plan. It is often inappropriate to recover consistency among data through manual operation and a simple backtracking method. With the dependency directed backtracker and demons, XL-S modifies over 200 data in this example. Figure 10 shows the main dependency in the working area of XL-S that is concerned with the modification.

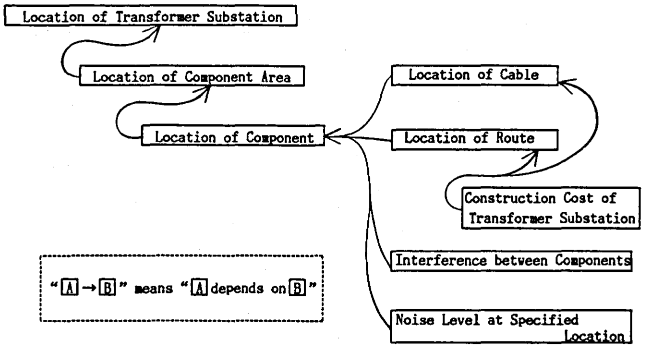

Location of Transformer Substation
Location of Component Area
Location of Component
Location of Cable
Location of Route
Construction Cost of Transformer Substation
Interference between Components
Noise Level at Specified Location
"A→B" means "A depends on B"

Figure 10. Dependency in Layout Plan

The layout design system XL-S is now undergoing functional tests conducted by expert designers. The preliminary results suggest that XL-S can reduce the period which is required to complete layout design of a substation. The reasons are listed below:

(1) The user does not need to handle the large amount of input data owing to default data and predefined procedures stored in the knowledge base.

(2) The user does not need to be concerned with both the existence and consistency of the data in the working area owing to dependency directed backtracking and demons.

4. CONCLUSION

A knowledge-based method has been developed for the layout design of industrial plants and applied to the layout design system for a substation. The proposed method is characterized not only by the direct utilization of designers' expertise, but also by intelligent functions embedded into the inference mechanism. These characteristics are summarized as follows:

(1) Designers' knowledge is utilized to generate layout plans that satisfy constraints and to evaluate the construction cost, noise level and other performance factors.

(2) The modification procedure is adopted to modify the data of an intermediate layout plan. The dependency directed backtracker deletes invalid data to make the data consistent in the layout plan, and demons reproduce new data for the modified layout plan.

(3) The knowledge base stores knowledge to control the search process for alternative plans, and to generate layout plans from different initial conditions. This allows it to support several alternative layout plans simultaneously.

The layout design system XL-S is now undergoing functional tests conducted by expert designers. The preliminary results suggest that XL-S can reduce the period which is required to complete layout design of a substation.

ACKNOWLEDGEMENTS

It is a pleasure for the authors to acknowledge the persons who have guided and encouraged them. In particular, they are grateful to Mr. R. Kadoi of the Tokyo Electric Power Co.,Inc. and Drs. S. Yamada, A. Doi, K. Inoue, H. Motoda, Mr. H. Sakai and Ms. R. Hara of Hitachi,Ltd.

REFERENCES

[1] K.G.Trickett et al., PDMS:Plant Layout and Piping Design, Computer-Aided Process Plant Design, edited by M.E.Leesley, 1123-1182, Gulf Publishing Co., Houston(1982)

[2] J.McDermott, R1:A Rule-Based Configurer of Computer Systems, Artificial Intelligence,Vol.19,No.9,39-88(1982)

[3] T.Watanabe et al., An Expert System for Computer Room Facility Layout, Proc. of 5th International Workshop, Expert Systems & Their Applications, Agence de l'Informatique, France(1985)

[4] Y.Wada et al, A Knowledge Based Approach to Automated Pipe-route Planning in Three-Dimensional Plant Layout Design, Proceedings of COMPINT 85,96-102(1985)

[5] G.J.Stallman and J.Sussman, Problem Solving About Electrical Circuits: Artificial Intelligence, An MIT Perspective, Vol.1,33-91(1979)

[6] J.Doyle, A Truth Maintenance System, Artificial Intelligence, Vol.12,No.3,231-272(1979)

[7] J.de Kleer, Choices Without Backtracking, Proceedings of AAAI84, 79-85(1984)

[8] V.Dhar, An Approach to Dependency Directed Backtracking using Domain Specific Knowledge,Proc. of IJCAI 85, 188-190(1985)

A LOGIC PROGRAMMING APPROACH TO FRAME-BASED LANGUAGE DESIGN

Hsin-Hsi Chen*, I-Peng Lin

Department of Computer Science and
Information Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.

Chien-Ping Wu

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.

Abstract

In this paper, we will propose a logic programming approach to design a frame-based language. The relationship among frame, logic and Prolog is our basic design issue. Frame is considered as a collection of slot-relations, and frame reference/inference procedure can be specified in logic form. Prolog is used to represent all of these concepts. Frame is encoded with Prolog facts and rules. This type of frame representation has the advantages of multiple access methods, ease of frame modification, slot inference rule creation and faster recognition. A frame recognition algorithm, similarity reference, is given as an example using our frame notation. With this approach, on the one hand, the semantics of logic makes it quite clear what frame and its deduction mean, and on the other hand, the simply extended Prolog will be more adequate and efficacious to represent knowledge.

1. Introduction

A good knowledge representation tool is not only adequate to the expression, but also efficacious to the reasoning [16]. Frames, which have good expressive power, were first proposed in [10]. Since then, several frame-based languages, such as FRL [6,13], KL-ONE [1], etc.[15], have been developed and used to implement systems in a variety of domains. How frames are involved in knowledge system's reasoning process was discussed in [5]. In most applications, e.g. natural language processing [1], diagnostic problems [12], and so on, frame recognition is one of major steps in the frame reference/inference process. How to select a frame to "best" match the observations, which are some attributes of an object, from large frame database, becomes an important problem. Thus, notation efficacy of frame representation affects the recognition step deeply. In this paper, we will propose a mechanism to encode frame notation with Prolog rules and facts, and make deduction from frame database with logic programming techniques.

Because "predicate logic as a programming language" is the basic idea behind Prolog [8], Prolog inherits many interesting features from logic [14]. We will emphasize the relationship between logic and frame [7], and take it into consideration of our logic programming approach. Section 2 will discuss briefly the relationship among frame, logic and Prolog. Section 3 will propose a frame representation with Prolog clauses, and its reference/inference mechanisms with logic form. A frame recognition algorithm will be given as an example using our frame notation. This experimental language, called LOGFOL (Pro<u>log</u>-based <u>F</u>rame-<u>o</u>riented <u>L</u>anguage) in [3], is implemented in CProlog [11], and operates on a DEC VAX-11/750 VMS system.

2. Frame, logic and Prolog

A frame is a data structure to represent stereotyped situations [10]. What constitutes a frame is implementation dependent, but it must be able to represent the properties of an object and the specifications of each property. For example, a classroom which is a complex object may be described by board, chalk, brush, and desk-and-chair. We can define a classroom with these properties as a lambda expression:

$$\lambda x\ (\text{board}(x,y_1) \wedge \text{chalk}(x,y_2) \wedge \text{brush}(x,y_3) \wedge \text{desk-and-chair}(x,y_4))$$

where x is some specific classroom;
y_1, y_2, y_3 and y_4 are its contents.

Properties board, chalk, brush and desk-and-chair are called slots of a classroom. Terms y_1, y_2, y_3 and y_4 are called fillers of some classroom. Relationship denoted by slot exists between a frame and its filler. Thus, on the one hand, given a frame that represents a concept, we can generate an instance of the concept by filling its slots to make these relations true. A logic formula expresses this concept as:

* The author is also a graduate student in the Institute of Electrical Engineering, National Taiwan University.

CH2345-7/86/0000/0223$01.00 © 1986 IEEE

$$\forall x\ (\mathrm{frame}(x) \rightarrow \exists\ y_1, y_2, \ldots, y_n\ (\mathrm{slot}_1(x, y_1) \wedge \mathrm{slot}_2(x,y_2) \wedge \ldots \wedge \mathrm{slot}_n(x, y_n)))$$

On the other hand, if we can find fillers for all the slots of a frame, then an appropriate instance of the concept exists. This concept can also be formalized as:

$$\forall x\ (\exists\ y_1, y_2, \ldots, y_n\ (\mathrm{slot}_1(x, y_1) \wedge \mathrm{slot}_2(x, y_2) \wedge \ldots \wedge \mathrm{slot}_n(x, y_n)) \rightarrow \mathrm{frame}(x))$$

Therefore, we can express the frame concept as logic formulae, and consider a frame as a collection of slot-relations.

Prolog is a powerful language which is used to solve problems that can be expressed in the form of objects and their relationship. It has some interesting features [9,14] helpful to design a frame-based language:

(1) Algorithm = Logic + Control [9]
The idea of Prolog is to specify the logic part only, not to care for the control part. If we regard a frame concept as a bundle of slot-relations and design it with logic programming techniques, then metaphysical and implementation interpretation of frames [7] are the same. So, frame reference/inference can be thought intuitively.

(2) multiple-purpose procedures
Not necessary to distinguish the roles of arguments in advance, some frame manipulation predicates may have more than one function. Especially, a predicate might be thought as a generator which generates result(s), or as a verifier which, given all arguments as inputs, checks whether a slot-relation is valid.

(3) high level form of iteration
Procedures may generate, through backtracking, a sequence of alternative results. Frame-based system is organized as a hierarchical network. The solution(s) in a frame reference/inference process depend on the completeness of the given information. Some search operations involved in frame reference/inference are tentative. It is likely to make trials until a solution is found. Backtracking mechanism embedded in Prolog takes care of the trial and error process very well.

(4) general record structure
Term structure makes frame representation more flexible. We can represent knowledge in a most appropriate way.

3. Language Structure

The programming structure of our extended Prolog system is shown in Fig. 1.

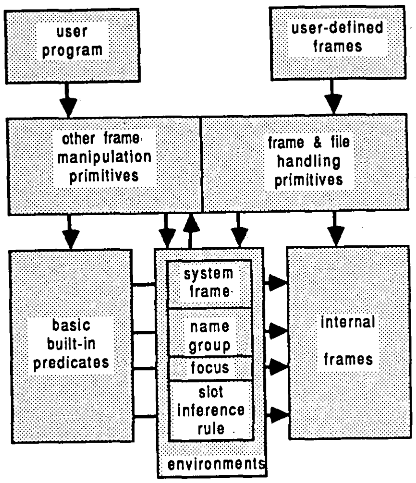


Figure 1 The programming structure of our extended Prolog system

This language includes three layers. The bottom layer is the frame definition, including two forms: the external one, with which users may represent knowledge of given problem domains intuitively, and the internal one, on which basic built-in predicates operate. The next-to-bottom layer consists of a collection of Prolog predicates used to manipulate the frame database. Frame reference/inference mechanisms constitute the top layer. All the frame manipulations are done under the current environments (system configuration, name group, focus and slot inference rule). We will discuss these issues in Section 3.1 and 3.2.

3.1 Frame Representation

From macroscopic viewpoint, a frame database is a collection of frames linked together. From microscopic viewpoint, a frame is like a tree (see Fig. 2 in FRL terms [6,13]).

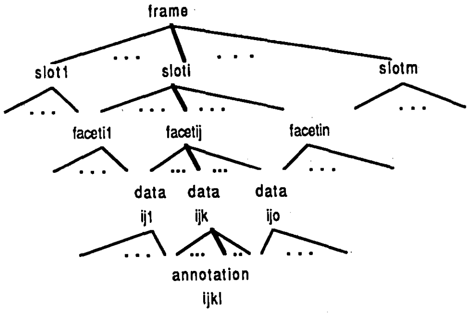
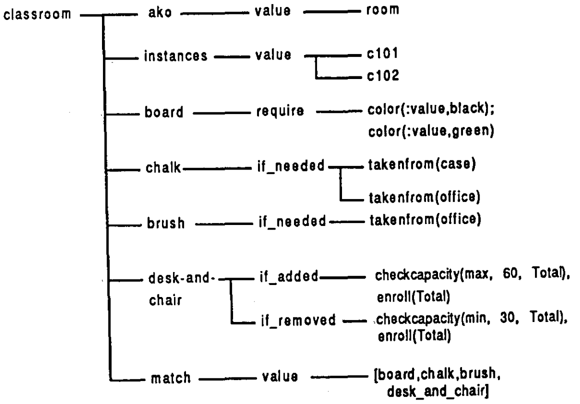


Figure 2 Frame in external form is like a tree.

Frame forms the semantics of a concept. Slot defines a property of a frame. Slot may be either system-defined or user-defined. There are system-defined slots specified in the system frame.

such as classification (generic/individual), ako and instances(for network construction) and match (for frame recognition). User-defined slot describes a specific feature for some concept. Facet specifies value, default, restrictions or interpretative hooks [1] in the corresponding slot. Useful built-in facets are: value, which specifies the slot's value; default, which denotes the default value; require, which defines the restrictions that filler must meet; if_needed, which provides a deduction rule (a sequence of goals) for the filler; and if_added and if_removed, which are used for network maintenance. Data may be either values or a goal statement depending on its facet's usage. Annotation specifies the source (perspective, ...) from which the information is collected. Such a definition constitutes a complete frame. A frame example (a generic classroom) which has been taken its syntactic sugar out is shown as follows:



Before the external frame is put into frame database, it must be preprocessed to a number of frame fragments, called internal frame. A frame fragment is retrieved along the path frame - slot - facet - data - annotation from the tree in Fig. 2 (see dark line). It is represented as a Prolog fact or rule, depending on its facet:

(1) value facet

```
slotname(framename, datum:annotation).
        or
slotname(framename, datum).
                            /* no comment */
```

This fact can be read as:

From annotation, the filler in the slot of a frame is datum (the relationship between frame and datum is slot).

(2) default facet

```
slotname(framename, default,
        datum:annotation).
        or
slotname(framename, default, datum).
                            /* no comment */
```

This fact can be read as:

From annotation, the default value in the slot of a frame is datum.

(3) other facets

```
slotname(framename, facetname, annotation)
    :- attached goals.
        or
slotname(framename, facetname,
        no_comment) :- attached goals.
```

This rule can be read as:

From annotation, the filler meets the facet's specification in the slot of a frame if attached goals succeed.

This viewpoint may be compared with Chang [2] who uses n-ary predicate to represent frame. Deliyanni and Kowalski [4] have discussed the advantages and disadvantages between binary representation and n-ary representation of logic form in the knowledge-based system.

An example (a generic classroom frame) is given below to illustrate this two-place or three-place slot-relation:

```
frame(classroom, [ako, instances, board, chalk,
                  brush, desk_and_chair, match]).
ako(classroom, room).
instances(classroom, c101).
instances(classroom, c102).
board(classroom, require, no_comment) :-
        curr_env(_frame, _slot, _facet, _value),
                /* current focus environment */
        (color(_value, black); color(_value, green)).
chalk(classroom, if_needed, no_comment) :-
        takenfrom(case); takenfrom(office).
brush(classroom, if_needed, no_comment) :-
        takenfrom(office).
desk_and_chair(classroom, if_added,
                no_comment) :-
        checkcapacity(max, 60, Total),
        enroll(Total).
desk_and_chair(classroom, if_removed,
                no_comment) :-
        checkcapacity(min, 30, Total),
        enroll(Total).
match(classroom, [board,chalk,brush,
                  desk_and_chair]).
```

Predicate frame denotes a generic classroom frame that has seven slots - ako, instances, board, chalk, brush, desk_and_chair and match.

This type of frame representation in Prolog fact or rule has the advantages of

(1) multiple access methods:

We may use "?- frame(framename, SlotList)." to find all the slots with a given frame name,

or use "?- slot(framename, Datum)." to find all the values with a given slot name of some frame,

or use "?- slot(Framename, datum)." to find all the frames with a given slot and its value.

In this retrieval process, no special hash mechanisms are required, because they are embedded in the Prolog system. This is in concordance with the idea of logic programming [9].

(2) ease of frame modification:
A good representation scheme must have capability of managing incomplete knowledge. In our paradigm, a newly invented property can merge into an existing frame easily; and on the other hand, an unsuitable property can also be retracted easily. These advantages are offered over the n-ary representation by the binary representation [4].

(3) slot inference rule creation:
Representing a slot as a relation enables slot-to-slot transference to be clear. For example,

"Y is a key of X if X has Y and Y is a key"
can be represented as:
keyof(X,Y) :- having(X,Y), iskey(Y).

Slot "keyof" is mapped into slot "having" by slot inference rule. Seeing-as relation proposed in [7] can also be specified by this type of inference rule.

(4) faster recognition:
During frame recognition, we will select a "best" frame which matches the observed features. In our paradigm, features of a frame are put in the form of slot-relations whose orders are not important. We can start out recognition with some evident features.

3.2 Environments

For each frame, there might be several names, one primary name and some secondary names, that can be used to identify it. Thus, each frame must have an associated list of allowable names which directly access it. We call each name a member of the name group of a frame. The name group is a set representing a collection of allowable names used to access some frame structure. To group two frame name X and Y together, we can use fgroup(X,Y), where X specifies an existing frame and Y is a frame name not bound to any existing frame yet. All the frame manipulations are through name group environment.

When one looks something in the world, he always concentrates on some part of it. However, he can retrieve all the relevant knowledge from this starting-point. There may be similar in the frame manipulation. During the frame manipulation, we focus only on some property (slot) of a specific frame. How to access other frame fragments within the same or different frame is an important issue. Because a frame is represented as a set of Prolog facts and rules, frame fragments are connected by the same frame name, and different frames are connected by instances, ako, or other relations. Focus transference is specified by environment change. Fact curr_env(frame, slot, facet, data) defines current focus environment. Structure :frame, :slot, :facet and :value in the clause refer to the current focus (see the generic classroom in the last section and its internal frame transformation). When a built-in predicate changes environment, the focusing frame fragment also changes at the same time.

The system configuration is defined by 'system' frame. A system frame is given as an example in the following:

```
sys_slots(system, classification).
sys_slots(system, ako).
sys_slots(system, instances).
sys_slots(system, match).
classification(system, default, generic).
primary(system, ako).
primary(system, instances).
```

Slot 'sys_slots' specifies system-defined slots. The default value of classification slot is generic. The value in the ako and instances slot must be a primary frame name. With system frame, we can maintain an appropriate system configuration for our own applications.

3.3 Example: similarity reference

The general process to manipulate a frame database of an application involves two steps:

(1) selecting a specific frame, and then
(2) applying operations on this frame.

We call step (1) frame reference. We may simply refer to a concept explicitly by its name. However, we may not know the name of the referred frame. Moreover, we cannot assure whether the frame already exists in the database. What we have known about an object is some of its features gathered from external world. So, frame reference becomes a recognition process. This type of reference, called similarity reference, is guided by the features of an object in question. We express the object's features as a collection of slot-relations, which constitute a temporary frame. If this frame does not exist in the database, an instantiation of some generic frame may be needed.

A set of requirements needs to be satisfied to ensure that a temporary frame X and a candidate frame Y are similar. In frame recognition, we express this concept in logic form as follows:

$$\forall X \ \forall Y \ (\text{relation}_1(Y,f_1(X)) \wedge \text{relation}_2(Y,f_2(X)) \wedge \ldots \wedge \text{relation}_m(Y,f_m(X)) \rightarrow \text{similarity}(X,Y))$$

In the above sentence, those conditions for X are specified by the frame Y. Relation$_i$ may be interpreted as a relationship between frame Y and the corresponding slot's value of temporary frame X. Function f_i tries to deduce the value from current knowledge base. Thus, when we specify f_i for each slot, we can define criteria to identify the frame. This sentence may be translated into Prolog-like clauses:

(c1) similarity(X,Y):- relation$_1$(Y,X), relation$_2$(Y,X), ..., relation$_m$(Y,X).

(c2) relation$_i$(Y,X):- slot$_i$(X,Z), slot$_i$(Y,Z).

(c3) relation$_i$(Y,X):- slot$_i$(X,Z), meet_requirement_of(Z, Y, slot$_i$).

We use (c2) and (c3) to simulate a deduction function f$_i$. The pragmatic requirement to consider two slots to be matched is either they have the same value (c2), or with the value conflict, but the pragmatic feature of the temporary frame meets the requirements of the corresponding slot of a candidate frame Y (c3).

The system built-in slot 'match' of a frame defines a frame recognition procedure. Given a temporary frame X, to identify a candidate frame Y as its referent, we trigger the recognition procedure of Y. Whether the identification succeeds depends on the satisfaction of this procedure call.

Match slot has two modes available: system mode and user mode. In system mode, interpreter takes the value of match slot, which is a list of slot names of a candidate frame, as input to a system-defined procedure (see next paragraph). The default for match slot is a list of all the slot name of the frame, except system built-in slots defined in the system frame. If the value of match slot is not a list of slot names, interpreter regards the value as an user-defined recognition procedure, and calls it directly. The latter usage is in user mode, which a user has all his own right to control the recognition process. The presentation of this mode increases the flexibility in the recognition process.

The system-defined procedure in system mode is clause (c1)-(c3). The i-th relation corresponds to the pragmatic requirement to match slot$_i$. The data in 'require' facet is any Prolog predicates, which constrain the allowable values of the slot, and control the slot matching. The 'require' facet specifies the requirement used by meet_requirement_of(Z, Y, slot$_i$) in the clause (c3).

A sample predicate

funify(TemporaryFrame,CandidateFrame,
UnmatchedSlotNameList)

which takes care of the recognition process mentioned above is shown below:

```
funify(FN1, FN2, UMSL) :-
    nonvarl([FN1, FN2]),                  /*FN1 and FN2 are instantiated  */
    get_fn(FN1, F1),                      /* get primary frame name from  */
    get_fn(FN2, F2),                      /* name group                   */
    ((fgeti(F2, match, value, P),         /* user-defined recognition     */
     not list(P),                         /* procedure                    */
      call(P),
      UMSL=[]);
     (not fgeti(F2, match, value, _), /*default for match slot: the      */
      fslots(F2, SNL),                    /* list of all the slot names of */
      fget(system, sys_slots, value, L), /* frame F2 minus                */
      diff(SNL, L, SL),                   /* system built-in slots        */
       funifyx(F1, SL, F2, UMSL));
     (fgeti(F2, match, value, L),         /* user-provided slot name list */
      list(L),
      funifyx(F1, L, F2, UMSL))
    ), !.

funifyx(F1, [SN|L], F2, UMSL) :-          /* (c2) SN(F1, D), SN(F2, D)    */
    fgeti(F1, SN, value, D),
    fgeti(F2, SN, value, D), !,
    funifyx(F1, L, F2, UMSL).
funifyx(F1, [SN|L], F2, UMSL) :-          /* (c3) SN(F1, D),              */
    fheritage(F2, SN, [require], PS),/* meet_requirement_of          */
    fgeti(F1, SN, value, D),              /*   (D, F2, SN)                */
    (push_env(F1,SN,require,D);          /*focus environment control    */
     pop_env, fail),                      /* focus environment control    */
    demons(PS),
    pop_env, !,                           /* focus environment control    */
    funifyx(F1, L, F2, UMSL).
funifyx(F1, [SN|L], F2, [SN|UMSL]) :-    /* SN is a potentially          */
    not fgeti(F1, SN, value, _), !,       /* matched slot                 */
    funifyx(F1, L, F2, UMSL).
funifyx(_, [], _, []).

nonvarl([H|T]) :- nonvar(H), nonvarl(T).
nonvarl([]).

get_fn(F1, F2) :-
    (frame(F1, @F2), !;
     frame(F1, _),
     F2=F1
    ).

diff(L1, [H|L2], L3) :-
    delete(H, L1, L4),
    diff(L4, L2, L3).
diff(L, [], L).

demons([P|L]) :- call(P), demons(L).
demons([]).

push_env(FN, SN, FCT, D) :-
    asserta((curr_env(FN, SN, FCT, D) :- !)).

pop_env :-
    retract((curr_env(_, _, _, _) :- !)), !.
```

Some of the built-in predicates used in the above example are summarized as follows. We use a notation to specify the role of an argument:

<i> input argument,
<o> output argument,
<*> input/output argument.

fget(FrameName<i>,SlotName<i>,
Facet-Datum-Annotation<i>, ItemList<*>)

The predicate fget gets a list of items along the path specified by FrameName, SlotName and Facet-Datum-Annotation, if ItemList is uninstantiated.

fgeti(FrameName<i>,SlotName<i>,
Facet-Datum-Annotation<i>, Item<*>)

The predicate fgeti is similar to fget, except that fgeti returns one item at a time. When backtracking, it will return another item, if exists.

fheritage(FrameName<i>,SlotName<i>,Facet<i>, ItemList<*>)

The predicate fheritage gets a list of data indicated by FrameName, SlotName and Facet, and data inherited along the slot ako, if ItemList is uninstantiated.

fslots(FrameName<i>, SlotNameList<*>)

SlotNameList<o>: The predicate returns a list of slotnames that belong to the frame specified by FrameName.

SlotNameList<i>: The predicate verifies the frame-slots relationship.

4. Conclusion

In this paper, we have described the design issues of our frame-based language, which represents and manipulates frames with logic programming techniques. On the one hand, we consider a frame to be a bundle of slot-relations, with which one may directly access a slot and its value. This design speeds up frame access operations and it has a merit of putting frames and matching assumptions into logic forms. The semantics of logic makes it quite clear what a frame structure means. On the other hand, we encode a frame mechanism in Prolog such that the simply extended Prolog will be more adequate and efficacious to represent knowledge.

References

[1] R.J. Brachman and J.G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, 9, 1985, pp.171-216.

[2] C.L. Chang, *Introduction to Artificial Intelligence Techniques*, Taipei, Sung Kang Computer Book Co., July 1985, pp. 91-97.

[3] H.H. Chen and J.Y. Fuh, "A Prolog-based Frame-oriented Language," *Proceedings of National Computer Symposium*, 1985, Republic of China, pp. 128-135.

[4] A. Deliyanni and R. Kowalski, "Logic and Semantic Networks," *Communications of the ACM*, Vol. 22, No. 3, March 1979, pp. 184-192.

[5] R. Fikes and J. Kehler, "The Role of Frame-based Representation in Reasoning," *Communications of the ACM*, Vol. 28, No. 9, Sept. 1985, pp. 904-920.

[6] I.P. Goldstein and R.B. Roberts, "Using Frames in Scheduling," in *Artificial Intelligence: An MIT Perspective, Vol. 1,* P.H. Winston and R. Brown, eds., MIT press, 1979, pp. 253-284.

[7] P. Hayes, "The Logic of Frames," in *Readings in Artificial Intelligence*, B.L. Webber and N.J. Nilson, eds., 1981, pp. 451-458.

[8] R. Kowalski, "Predicate Logic as a Programming Language," *Proc. IFIP 74*, Amsterdam, North-Holland Publishing Co., pp. 569-574.

[9] R. Kowalski, "Algorithm = Logic + Control," *Communications of the ACM*, Vol. 22, No. 7, July 1979, pp. 424-435.

[10] M. Minsky, "A Framework for Representing Knowledge," in *The Psychology of Computer Vision,* P.H. Winston, ed., New York, McGraw-Hill, 1975, pp. 211-277.

[11] F. Pereira, D.H.D. Warren, L. Byrd, and L.M. Pereira, *CProlog User's Manual*, SRI International, Melo Park, California, 1983.

[12] J.A. Reggia, D.S. Nau, and P.Y. Wang, "A New Inference Method for Frame-based Expert System," *Proceedings of the National Conference on Artificial Intelligence*, Aug. 1983, pp. 333-337.

[13] S. Rosenberg and B. Roberts, "Coreference in a Frame Database," *Proceedings of the Sixth International Joint Conference on Artificial Intelligence,* Tokyo, Vol. 2, Aug. 1979, pp. 729-734.

[14] D.H.D. Warren, *Applied Logic - Its Use and Implementation as a Programming Tool,* SRI Technical Note 290, June 1983.

[15] D.A. Waterman, *A Guide to Expert System*, Addison-Wesley, 1985, pp. 346-350.

[16] W.A. Woods, "What's Important About Knowledge Representation?" *IEEE Computer*, Oct. 1983, pp. 22-27.

INTERFACING PROLOG TO PASCAL

KENNETH MAGEL

NORTH DAKOTA STATE UNIVERSITY

ABSTRACT

This paper presents the implementation of a mechanism for linking an existing Prolog interpreter with Pascal programs so that each may call the other and so that each may share data with the other. Various approaches for linking the two languages are considered and compared. Some performance results for mixed applications are described.

INTRODUCTION

Prolog presents a convenient paradigm for implementing applications in areas which involve a set of facts and ask for inferences from those facts and rules relating facts. Prolog includes a naive automatic backtracking mechanism as well as a pattern matching-triggered application of rules. Even in these areas, however, Prolog does not provide all of the features which would make implementation as convenient as possible. In some situations where the Prolog implementation is easily constructed, the implementation will be inefficient. For example, sorting may be written rather easily in Prolog as the generation of successive permutations of a list and the testing of each to determine which is sorted. Unfortunately, such an approach is much less efficient than a more procedural one such as might be written in Pascal.

Many already written and debugged routines exist in widely used languages such as Pascal. Being able to use these routines from a Prolog program would be extremely valuable for many applications. A major advantage of newer procedural languages such as ADA and Modula-2 is their extensive support for the construction and use of libraries of routines (called packages or modules).

When Prolog is implemented on conventional vonNeumann computers, the implementation is usually inefficient and requires rather complex data structures to handle the fluid data typing, support for manipulation of not yet defined quantities, backtracking, and tentative assignments [1]. Some applications and parts of applications which could conveniently be done in Prolog can be done more efficiently in other languages because of these implementation considerations.

For these reasons, we set out to link an existing Prolog implementation [2] with programs written in a procedural language. We wanted the interference to be as open and flexible as possible. In particular, we wanted routines in each language to be able to invoke routines in the other language. Routines in each language should be able to manipulate global data and structured parameters and to return any value which a routine in the native language could return. We choose an extended Pascal which supported separate compilation for two reasons: (1) we had substantial code and expertise in Pascal; and (2) the Prolog implementation is written in a common subset of Pascal. The approach should be applicable to linking Prolog with other languages as well. We presently are starting a project to link another version of Prolog implementation in C with Pascal and Fortran programs. The Fortran-Prolog system will be particularly useful to some engineers on our campus who are developing expert systems to support engineering applications.

RELATED WORK

Three approaches have been tried to mixing languages in a single application: (1) extending an existing or proposed language to incorporate features from other languages and paradigms [3]; (2) using an intermediate file or files to communicate information between programs in each language; and (3) some sort of escape mechanism which a program in one language may employ to call upon routines in other languages. The following paragraphs discuss each of these approaches.

Incorporating multiple paradigms into one language has several advantages. The problem of passing information from one paradigm to another is handled entirely by the one language implementation. Paradigms may be mixed freely, even for rather small units of shifting paradigms. The major disadvantage from our perspective is the difficulty in using existing programs in a procedural language. Further, our experience indicates that programmers may have difficulty in understanding and using effectively a variety of paradigms which are presented in the same setting. In addition to the work described in [3], C. Mellish et al describes a POPLOG implementation which

CH2345-7/86/0000/0229$01.00 © 1986 IEEE

provides an integrated environment for writing POP-11 and Prolog programs which communicate [4]. The compilers generate a common intermediate language.

The intermediate file approach gets around the problem of communicating structured data since each language has an output facility and an input facility. As long as one language provides the capability of outputing data in a form which the other language can input, this approach works. The major problem with the intermediate file is its inefficiency. Input and output usually are the slowest operations in any language because the underlying hardware is relatively slow. When intermediate files are used, the application units implemented in each language must be quite large to make up for the time overhead of the file communication. A mechanism such as the pipes of UNIX can reduce this overhead significantly. Nevertheless, the need to translate the data from its internal form for one language to an eternal form and then to the internal form of the other language still imposes substantial overhead.

The escape mechanism approach can be implemented efficiently. Unfortunately, it is a one-way approach in the sense that communication via the escape mechanism goes from one language to the other, but not in the other direction. A further problem is the complexity of the data structure for a term in our version of Prolog. Figure 1 shows the data structure for a term in our version of Prolog. Arrows indicate references by pointers to subsidiary structures. Double vertical lines are used to indicate the additional fields which are present when the upper field has the indicated value. The routines in the other language must be written to be careful not to affect the integrity of the Prolog data structures. A final problem involves how to handle backtracking when the language escaped to does not include such a concept. There have been some previous efforts to utilize the escape mechanism approach. These efforts have been limited in either of two ways: (1) they were set up to handle specific applications with specific communications needs; or (2) they support calling programs in the other language, but only limited data transfer back and forth. Perhaps the earliest such effort was by Santane-Toth and associates in Hungary [5]. They set up a system to invoke Fortran subroutines from Prolog, but only simple non-structured data could be passed across the interface. The goal was a system which would support numerical processing in Fortran with all other aspects of the application done in Prolog. Nilsson described a small structure-sharing Prolog interpreter implemented in LISP which supported an escape to LISP evaluable predicate [6]. Several authors have considered the integration of Prolog with functional programming [7,8].

OUR IMPLEMENTATION

We choose to use the escape mechanism approach, but extended it to support two way communication. We handled backtracking by deciding that all actions done by Pascal programs on Pascal data would not be undoable. All changes made by the Pascal programs to Prolog data would follow the same rules as if those changes had been made by Prolog code at the point of the invocation of the escape mechanism. If backtracking passes back through a point of invocation of the escape4 predicate, the escape interface is called with a special first parameter to indicate that. All changes made to Pascal data by Prolog code are not undoable. This solution is not wholly satisfactory, but alternatives seem to require re-implementation of a backtracking mechanism for Pascal. We are designing a mechanism to support backtracking of Prolog changes to Pascal data. The mechanism requires a stack of previous values for each Pascal variable and may offset some of the speed advantage of using Pascal. The possibility of using precompiled Pascal routines would be eliminated. We expect to use the non-undoable version of Pascal nearly all the time.

One evaluable predicate, escape4 was added to Prolog. This predicate takes four parameters which may be any type of data in the Prolog system including anonymous variables, uninstantiated variables, unchanged goals, atoms, integers, operator expressions, complex terms, or lists. This data may be changed in arbitrary ways by the Pascal routines. In addition, some routines were added to the Prolog interpreter which may be invoked by the Pascal programs. These routines support retrieval by the Pascal programs of any global data from the Prolog side and the execution of parts of the Prolog program as subroutines of the Pascal programs. They also provide the ability for the Pascal programs to use the program modification facilities of Prolog to change the Prolog program.

The Prolog implementation of escape4 calls a Pascal external routine named EscapeRoutine. This routine may be written by the user, but we provide two versions which may be used as is or as models. The first is given as the appendix to this paper and provides some simple routines to manipulate the Prolog data passed as parameters, but leaves most of the burden of handling the complicated Prolog data structures to the user.

The second version of EscapeRoutine is much larger and provides a simple set of abstract datatypes to hide the actual data structure implementations. These allow the user to retrieve fields of any parameter data structure, and to set fields of any accessible data. New data of any type may be created as well. The abstractions are identical with those we expect to provide to Pascal and Fortran from the C implementation of Prolog. They involve arrays and single data values plus a flag to indicate undefined and another flag to indicate if the assignment should be permanent or only until backtracking goes through the statement which invoked the escape routine. As much as possible, we try to present the Pascal or Fortran programmer with facilities which are consistent with Pascal or Fortran. We try to give the Prolog programmer facilities which are consistent with Prolog.

intended for use in large applications where different programmers might be writing the procedural code from those writing the Prolog code. We use a graphical view of the shared data structure to indicate to each group and what data is available and how that data may be used by each side. A Prolog program is used to develop and maintain these data diagrams. We are implementing a program in Prolog to check Prolog and Pascal programs for adherence to the constraints described in the data diagram.

Forty-three statements of Pascal were added to the existing interpreter in all. The simpler EscapeRoutine is 120 lines of Pascal while the more complex one is about 2100 lines. We have done two applications in Prolog and in the mixed Prolog-Pascal system to determine if the mixed system was of use or not. In each case, the application was written in Prolog, and debugged. Then a trace was used to identify where in the Prolog program significant time was spent. Those sections were considered for reimplementation in Pascal. Hence the results are skewed in favor of the mixed system. The first application is a primitive calendar system which provides reminders of appointments on a daily or weekly basis and warns of overlapping commitments. The second application is a keyword-based reference retrieval system. With the first system, approximately twelve percent of the total 260 Prolog statements were rewritten in Pascal. On a typical run, the mixed system ran seventy-two percent faster than the Prolog version. With the second system, approximately seven percent of the total 174 Prolog statements were rewritten in Pascal. On a typical run, the mixed system ran twenty-one percent faster than the Prolog version. Results of twenty test runs with different data ranged from fourteen to forty-one percent faster for the mixed version compared to the Prolog version.

The approach presented here may be extended in several directions. Languages which incorporate other paradigms (e.g., object-oriented as in SMALLTALK) could be accessed from Prolog or a procedural language in the same manner. Spreadsheets, databases, painting programs, and other applications could be accessed in some cases. An interlanguage tracing and debugging tool should be developed to assist production of multiple language systems.

REFERENCES

[1] Campbell, J. A. editor, Implementations of Prolog, Ellis Horwood, New York, 1984.

[2] Spivey, J. J., "Pascal Source Code of the Interpreter", University of York Protable Prolog System, Release 1, Department of Computer Science, University of York, Great Britain, 1983.

[3] Hailpern, Brent guest editor, IEEE Software Special Issue on Multiparadigm Languages and Environments, Volume 3, no. 1, January, 1986.

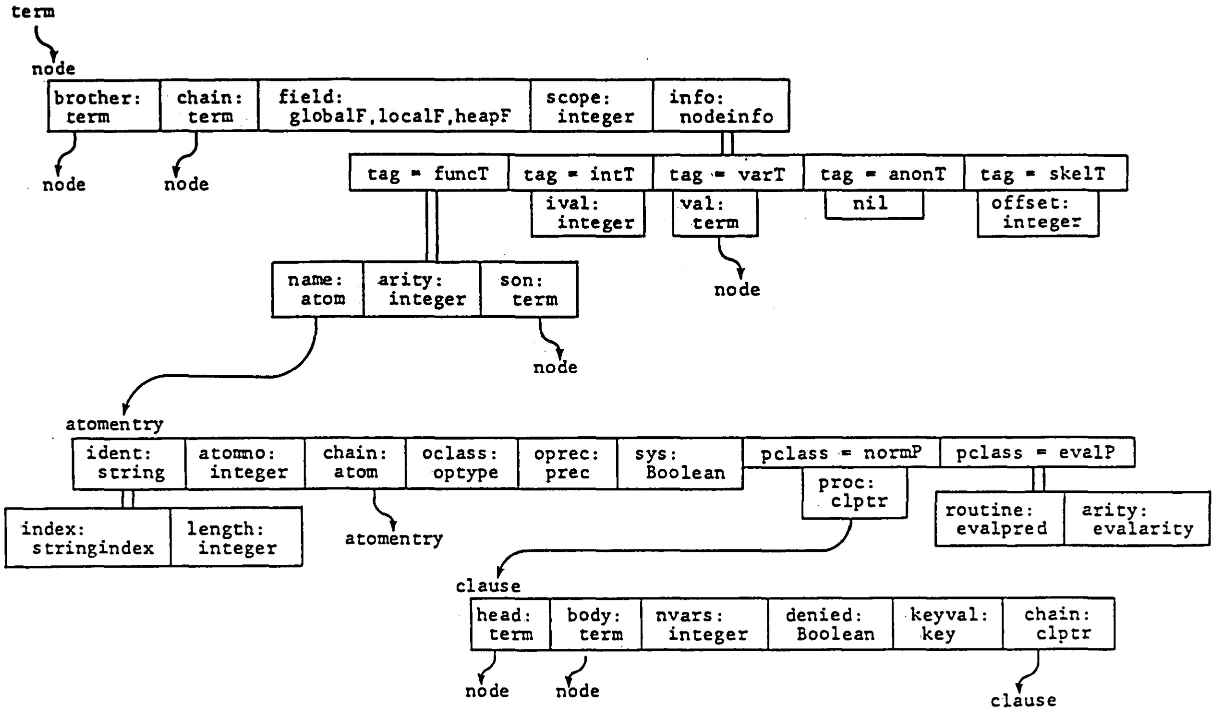
[4] Mellish, C. S., "An Alternative to Structure Sharing in the Implementation of a PROLOG INTERPRETER", in Logic Programming, edited by Clark and Tarnlund, Academic Press, 1982.

[5] Santane-Toth, E. and Szeredi, P., "Prolog Applications in Hungary", in Logic Programming, edited by Clark and Tarnlund, Academic Press, 1982.

[6] Nilsson, M., "The world's shortest PROLOG interpreter?", in Implementations of Prolog, edited by Campbell, Ellis Horwood, 1984.

[7] Robinson, J. A. and Sibert, E., "LOGLISP: Motivation, Design and Implementation", in Logic Programming, edited by Clark and Tarnlund, Academic Press, 1982.

[8] Bellia, M., et al, "A formal model for lazy implementations of a PROLOG-compatible functional language", in Implementations of Prolog, edited by Clark and Tarnlund, Academic Press, 1982.



```
stringindex = 0 .. StringSpace;                          (* StringSpace = 8800 *)
optype = (fx0,fy0,xf0,yf0,xfx0,xfy0,yfx0,non0);
prec = 0 .. MaxPrec;                                     (* MaxPrec = 1200 *)
evalpred = (callR, cutR, readR, writeR, ...., escape4R);  (* see Appendix C -Globals *)
evalarity = 0 .. MaxEvalArity;                           (* MaxEvalArity = 4 *)
key = integer;
```

A Complete Representation of the Data Structure of a Term

ESCAPE ROUTINE

```
#include 'prolog2.h'
#include 'prologEF.h'


procedure EscapeRoutine (* (a1, a2, a3, a4: term; e: env) *);
(*
  instructions in this routine are to be written by the user to do
  whatever he wishes.  BE CAREFUL!!!!! -- this allows you to
  manipulate the PROLOG database containing the 4 terms passed to this
  routine.
*)

  var atom1, atom2, atom3, atom4: atom;
      argval: array [1..MaxEvalArity] of term;
      str: array [1..1000] of charstr;

  procedure AtomToChar  (var answer: charstr; a: atom);
  (* converts an atom to a packed array of character.  *)
    var n, k: integer;
        charstring: charstr;
  begin
                            (* initialize charstring to blanks *)
    charstring := '     ';
    k := 1;
    with a↑.ident do
      for n := index + 1 to index + length do
        begin
          charstring[k] := stringbuf[n];
          k := k + 1
        end;
    answer := charstring
  end (* AtomToChar *);

  procedure WriteEscTerm;
  (*
     Will write out the data type and the individual atoms (one per
     line) of each of the 4 terms passed to the EscapeRoutine.
     WARNING:  this procedure will not work well for mixed data
     structures such as lists within complex terms, etc.  This is only
     given as an example of how to access each atom in a simple data
     type:  integer, uninstantiated variable, anonymous variable,
     unchanged goal, atom, lists, operator expressions, complex terms.
  *)
```

```
    var i: integer;
         s, y: term;
  begin
    argval[1] := a1; argval[2] := a2;
    argval[3] := a3; argval[4] := a4;
    for i := 1 to 4 do begin
       y := Deref(argval[i],e);
       case y↑.info.tag of
          funcT:
            with y↑.info do
              if arity > 2 then begin
 (* complex term *)
                 writeln('arg[',i:1,
                          '] is a complex term whose atoms are:');
                 WriteAtom(name); writeln;
                 WriteAtom(son↑.info.name); writeln;
                 s := son↑.brother;
                 while s <> nil do begin
                     WriteAtom(s↑.info.name); writeln;
                     s := s↑.brother
                 end (* while *)
              end (* if arity > 2 *)

              else
                 case arity of
                    0:
(* atom *)          begin
                         writeln('arg[',i:1,
                                  '] is an atom whose name is:');
                         WriteAtom(name); writeln
                       end;
                    1:
(* unchanged goal *)
                       if name = curlyA then
                         writeln('arg[',i:1,
                                '] ---> goal {..} to be unchanged')
                       else if
                         name↑.oclass in [fx0,fy0,xf0,yf0] then
                            begin
(* operator exp *)       writeln('arg[',i:1,
                                 '] is an operator expression of:');
                               case name↑.oclass of
                                  fx0,fy0:
                                     begin
                                         WriteAtom(name); writeln;
                                         writeln(son↑.info.ival)
                                     end;
                                  xf0,yf0:
                                     begin
                                         writeln(son↑.info.ival);
                                         WriteAtom(name); writeln
                                     end
                               end (* case oclass *)
                       end (* if name↑.oclass *)
                       else
                       begin
(* complex term *)   writeln('arg[',i:1,
                        '] is a complex term whose atoms are:');
                         WriteAtom(name); writeln;
                         WriteAtom(son↑.info.name); writeln;
                         s := son↑.brother;
                         while s <> nil do begin
                             WriteAtom(s↑.info.name); writeln;
                             s := s↑.brother
                         end (* while *)
                        end (* else *);
                   2:
(* list *)         if name = consA then
                         begin
                            writeln('arg[',i:1,
                                      '] is a list of:');
                            if son↑.info.tag = intT then
                                writeln(son↑.info.ival)
                            else begin
                                WriteAtom(son↑.info.name);
                                writeln
                            end (* else *);
                            s := Deref(son↑.brother,e);
                            while IsFunc(s,consA,2) do begin
                              if s↑.info.son↑.info.tag=intT then
                                   writeln(s↑.info.son↑.info.ival)
                              else begin
                                  WriteAtom(s↑.info.son↑.info.name);
                                  writeln
                              end (* else *);
                              s := Deref(s↑.info.son↑.brother,e)
                            end (* while *)
                       end (* if name = consA *)

                       else if
                         name↑.oclass in [xfx0,xfy0,yfx0] then
                            begin
(* operator exp *)       writeln('arg[',i:1,
                                '] is an operator expression of:');
                              writeln(son↑.info.ival);
                              WriteAtom(name); writeln;
                              writeln(son↑.brother↑.info.ival)
                            end
                       else
                        begin
(* complex term *)   writeln('arg[',i:1,
                           '] is a complex term whose atoms are:');
                          WriteAtom(name); writeln;
                          WriteAtom(son↑.info.name); writeln;
                          s := son↑.brother;
                         while s <> nil do begin
                             WriteAtom(s↑.info.name); writeln;
                             s := s↑.brother
                         end (* while *)
                        end (* else *)
                 end (* case arity *);
              intT:
                 begin
(* integer *)   writeln('arg[',i:1,
                             '] is an integer with value:');
                   writeln(argval[i]↑.info.ival)
                 end;
              varT:
(* uninstantiated or anonymous variable *)
                 writeln('arg[',i:1,
                 '] is an uninstantiated or anonymous variable');
              anonT:
(* anonymous var *)
                  writeln('arg[',i:1,
                            '] is an anonymous variable')
       end (* case tag *)
     end (* for i *)
  end (* WriteEscTerm *);

begin
  WriteEscTerm;

  atom1 := a1↑.info.name;
  atom2 := a2↑.info.name;
  atom3 := a3↑.info.name;
  atom4 := a4↑.info.name;

  writeln
     ('*** Writing out first Atom as a CharacterString ***');
  AtomToChar(str[1],atom1); writeln(str[1]);
  AtomToChar(str[2],atom2); writeln(str[2]);
  AtomToChar(str[3],atom3); writeln(str[3]);
  AtomToChar(str[4],atom4); writeln(str[4]);

  (*
      The following will write to standard output if echoing or not
      telling (and write to the tellfile if telling) each term in PROLOG
      form.  Note:  writeln's here will not be written to the tellfile.
  *)

  WriteOut(a1,e); writeln; WriteOut(a2,e); writeln;
  WriteOut(a3,e); writeln; WriteOut(a4,e); writeln
end (* EscapeRoutine *);
```

Knowledge-Based Optimization in Prolog Compiler

Naoyuki Tamura
Science Institute, IBM Japan, Ltd.
5-19 Sanban-cho, Chiyoda-ku, Tokyo 102 JAPAN

Abstract

This paper describes an optimization technique used in a Prolog compiler. The compiler generates efficient code for several machines including IBM System/370 and a RISC machine, IBM RT PC.

We stress on *knowledge-based approach* for optimization. Prolog programs are compiled into intermediate code and the code is translated into a graph for optimization. The optimization consists of following two steps; (1) tracing the graph to infer the behavior of the intermediate code by using a *semantic definition of the intermediate language*; and (2) simplifying the graph by *graph reduction rules.*

This optimization improves the performance of a list concatenation program by a factor of 1.8. And when some user's declarations are added, the improvement of the optimization becomes about 2.0.

1 Introduction

This paper describes an optimization technique used in a Prolog compiler. The compiler generates efficient code for several machines including IBM System/370 and a RISC machine, IBM RT PC [6].

Figure 1 shows the outline of the compiler [7].

In phase-1, Prolog programs are compiled into intermediate code. The intermediate language is based on David Warren's "Abstract Prolog Instruction Set" [5], but some instructions are decomposed into lower level instructions.

In phase-2, the intermediate code is translated into a graph. The optimizations are then performed on this graphical representation of the program. Then the optimized graph is translated back to intermediate code.

In phase-3, the intermediate code is translated into a high level language program — PL.8 [1] — as an object code.

Phase-4 is a PL.8 compilation phase.

2 Prolog compiler

There are some problems with the compilation of Prolog programs compared with other conventional languages, such as FORTRAN and Pascal.

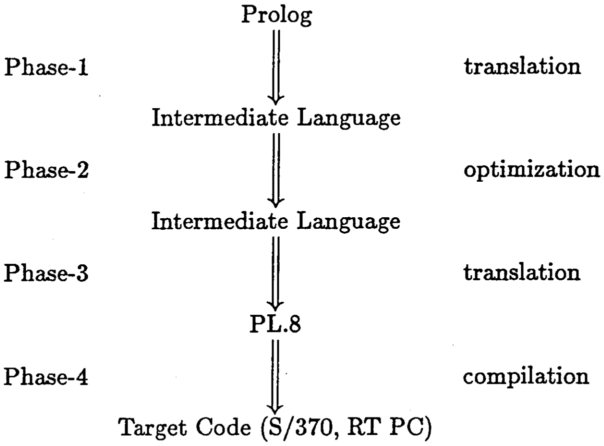


Figure 1: Outline of the Prolog compiler

- **typelessness**
  Prolog is basically a typeless language. Variables can contain any type of data and different operations are required for different data types.

- **bidirectionality**
  Any argument can work as both input and output. The same argument can be used to construct the structure or to read the elements of the structure.

In David Warren's "Abstract Prolog Instruction Set" [5], the first point is solved by using tags to distinguish data types. The second is done by using read/write modes to indicate whether the unification in process is destructive or constructive.

Tick and Warren's "Pipelined Prolog Engine" has a piece of hardware to test tags and read/write modes at execution time with little overhead [3]. But for conventional machines, testing at execution time requires several instructions and is an overhead. Therefore, for those machines, optimizing test operations is useful to improve the performance.

However, the level of Warren's instruction set is too high to perform the optimization because testing operations are implicitly embedded in most instructions. For example, the instruction, `get_list A1`, implies following operations:

1. test the tag of the register `A1`

2. if A1 is a bound variable,
   dereference A1 and go to 1

3. if A1 is an unbound variable,
   create a list cell and set to write mode

4. if A1 is a list,
   get the address of the list and set to read mode

5. otherwise,
   fail

As far as get_list is used as a primitive instruction, it is impossible to optimize it even if A1 is always a list.

Therefore, in our compiler, about 30% of instructions are decomposed into lower level instructions. Especially, in get and unify instructions, testing data types and read/write modes are explicitly coded so that the compiler can optimize the code.

3 Optimizations

The optimizer uses a graph representation of the intermediate language. The graph consists of nodes and directed edges, and represents the control flow of the intermediate code. In general, there is one node for each intermediate instruction, and there is a directed edge from a node N_1 to a node N_2 if N_2 could immediately follow N_1 during execution.

The following two steps are applied repeatedly for the graph:

- *graph tracing*
- *graph simplification*

In the "graph tracing" step, the optimizer traces the graph in a depth-first manner and infers the possible data types and read/write modes. In the "graph simplification" step, the optimizer simplifies the graph by using information generated in the tracing step.

These two steps are repeatedly executed as long as there is a node that can be deleted.

3.1 Data types and case instruction

Before the discussion of the optimizations, data types and case instruction of the intermediate language are described.

The type of a variable is expressed by the term type(V)=T where V is a variable and T is a type, such as int, atom, nil, list, struct, and ref.

The case instruction is used to test a data type or a read/write mode. The case instruction for type testing of a variable V has a form:

```
case type(V) of {
  T1  ->  S1;
  T2  ->  S2;
    ...
  Tn  ->  Sn
}
```

where
T_i is a data type,
S_i is a statement of the intermediate language.

All T_i must be exclusive like a guarded command [2]. As for the case instruction for mode testing, a form is

```
case mode of { ... }
```

and T_i is either read or write.

3.2 Graph tracing

In this step, the graph is traced to get the information of possible data types and modes. That is, the optimizer starts from an entry node with no information, and then, step by step, traces the graph updating the information.

In fact, the information is a condition of data types and read/write modes for each edge of the graph. For example, suppose an edge has the condition type(a(1))=list. This implies the register a(1) is always a list at that edge at the execution time.

While tracing, the optimizer uses a *semantic definition table* to know the effect of each intermediate instruction. The table defines the post-condition of each instruction. For example, the post-condition of the instruction put_nil(V) is type(V)=nil that means the type of variable V will be nil after the execution of the instruction. Post-conditions of some built-in predicates are also described in the semantic definition table.

Marking of traced nodes are also performed in this step to distinguish unreachable nodes.

3.3 Graph simplification

Graph reduction rules are applied to the graph by using the information obtained in the previous inference steps. Rules for the following optimizations are described as an example.

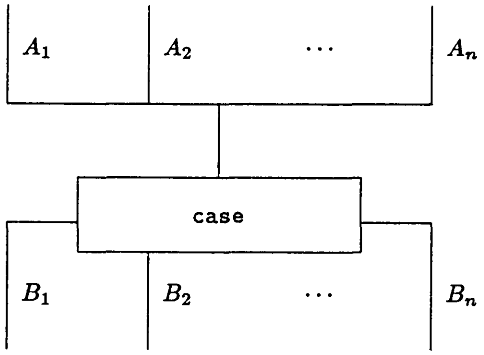
1. eliminating redundant case instructions
2. eliminating never-selected case entries
3. eliminating unreachable instructions

For the explanation of 1 and 2, suppose there is a subgraph shown in Figure 2 where A_i is a condition obtained by the inference step, and B_j is a condition to select that case entry. Optimization rules 1 and 2 are:

Rule 1 : If
$(A_1 \,|\, A_2 \,|\, \ldots \,|\, A_n) \,\&\, B_j$ is false for some j,
the entry B_j can be omitted.

Rule 2 : If

Figure 2: case node in a graph



A_i & B_j is true for some i and j
and
A_i & B_k is false for all $k \neq j$,
the edge from A_i can be redirected to B_j to skip the case instruction.

Rule 3 is very simple:

Rule 3 : If
a node is unmarked in the tracing step,
it can be deleted.

3.4 Example

The following is an example of optimization.

```
label_1 :
  case type(a(1)) of {
    ....................
    list -> goto(label_2);    <-- (a)
    ....................
  };
  ....................
label_2 :
  case type(a(1)) of {
    nil  -> goto(label_3);    <-- (b)
    list -> goto(label_4)     <-- (c)
  };
  ....................
label_3 :
  ....................
label_4 :
  ....................
```

Suppose goto statement at (a) is the only one for label_2. Then the condition at the label_2 would be type(a(1))=list. The *Rule 1* can be applied to the case instruction to delete the entry (b) because type(a(1))=nil is always false. And also, the *Rule 2* can be used because then entry (c) is the only choice. Therefore, the second case instruction can be skipped by redirecting the destination of (a) to label_4. Moreover, in the next graph simplification step, the case instruction at label_2 will be deleted by *Rule 3* because there is no goto for that label.

After these optimizations, the code will be as follows:

```
label_1 :
  case type(a(1)) of {
    ....................
    list -> goto(label_4);
    ....................
  };
  ....................
label_3 :
  ....................
label_4 :
  ....................
```

3.5 User's hints

In DEC-10 Prolog [4], input/output mode declaration is used to improve the performance. Such kind of information simplifies the control flow of the code and helps the optimizer by reducing the number of cases to be considered.

We introduced type declarations. These declarations are used to assert the data types of the arguments of a predicate.

When these declarations are added by a user, the code is considerably optimized compared to the code with no declarations.

3.6 Measurement of Optimization

Table 1 shows the improvement of the optimization for a list concatenation program. The source code is as follows:

```
concat({},L,L).
concat({X|L1},L2,{X|L3}) <- concat(L1,L2,L3).
```

In Table 1, are only considered the effects of the optimization techniques described in this paper. The results displayed in the "Opt" case have been obtained which using all the optimization techniques we have developed for our compiler (as described in [7]). In the "No opt" case, we have only omitted the optimizations obtained by the techniques described in this paper.

In case of no declarations, the performance is improved by a factor of 1.79, and when mode and type declarations are added, by a factor of 1.99.

Table 1: Improvement of the optimization for the procedure concat

concat	No declarations		Mode & type declarations	
	No opt	Opt	No opt	Opt
Number of nodes	103	95	99	51
Performance on IBM 3081K (KLIPS)	341	611	362	720

Table 2: Improvement of the optimization for other programs (number of nodes)

Number of nodes	No declarations		Mode & type declarations	
	No opt	Opt	No opt	Opt
nreverse	162	143	156	86
quick sort	260	235	254	169
N queen	327	287	317	177

Improvements for other typical benchmark programs are shown in Table 2. The number of nodes before optimization and after optimization is displayed in this table.

4 Conclusion

In this paper, optimization techniques used in a Prolog compiler are described. The optimizer is designed using a knowledge-based approach. Optimizations are performed by (1) inferring the behavior of the intermediate code and (2) simplifying the graph by using the inferred information.

This optimization improves the performance of a list concatenation program by a factor of 1.8. And when mode and type declarations are added, the improvement of the optimization is about 2.0.

Acknowledgement

I wish to thank Yasuo Asakawa and Hideaki Komatsu for their cooperation of designing and implementing the compiler. I would like to give thanks to Toshiaki Kurokawa, Tetsunosuke Fujisaki, and Peter Woon for their helpful advice.

References

[1] Auslander, M. and Hopkins, M., *An Overview of the PL.8 Compiler*, Proc. of the SIGPLAN '82 Symposium on Compiler Construction, Vol.17, No.6, June 1982.

[2] Dijkstra, E. W., *Guarded Commands, Nondeterminancy and Formal Derivation of Programs*, CACM Vol.18, No.8, pp.453-457, August 1975.

[3] Tick, E. and Warren, D. H. D., *Towards a Pipelined Prolog Processor*, Proc. of 1984 International Symposium on Logic Programming, IEEE Computer Society, 1984.

[4] Warren, D. H. D., *Implementing Prolog — compiling predicate logic program*, Research Reports 39 & 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977.

[5] Warren, D. H. D., *An Abstract Prolog Instruction Set*, SRI International Technical Note 309, October 1983.

[6] International Business Machines Corporation, *RT Personal Computer Technology*, No.SA23-1057, 1986.

[7] Kurokawa, T., Tamura, N., Asakawa, Y., and Komatsu, H., *A Very Fast Prolog Compiler on Multiple Architectures*, (to appear in Proc. of the 1986 ACM/IEEE Computer Society Fall Joint Computer Conference).

Communication with Expert Systems

Kathleen R. McKeown
Department of Computer Science
450 Computer Science
Columbia University
New York, N.Y. 10027

Abstract

The use of natural language to interact with an underlying expert system entails problems in both generation and interpretation that differ significantly from problems that have been addressed in interfaces to other types of systems. In this paper, we show how a natural language interface to an expert system can make use of the discourse environment to generate explanations that are tailored to an individual user's concerns and must derive facts in addition to a query when interpreting an input question. The techniques presented are applicable as well to more general problem solving, advising, and consulting systems.

1 Introduction

In the past, expert systems that communicated interactively with their users to gather data and convey results used relatively simple interfaces for interaction such as menus (e.g., MYCIN (Shortliffe 76)) or restricted forms of natural language (e.g., Prospector (Duda 78)). The development of a full-blown natural language interface for expert systems raises a number of difficult research questions. This paper explores how questions that must be addressed for natural language within the expert system environment differ from questions that have been addressed in the database environment, shows how the expert system provides an ideal environment to address problems in user modeling and natural language generation, and identifies requirements for natural language interpretation in the expert system environment.

The use of a natural language interface in place of a menu interface means that any single request for advice is embedded within an ongoing dialog which can provide information about a user's higher level problems and concerns. A natural language system can use this information to generate explanations that are tailored to an individual user's concerns. Furthermore, a user can provide information in a single natural language question or statement that would have to be requested via a sequence of questions in a menu system. This means that a natural language interface can avoid lengthy interactive sequences to gather information and more quickly provide the desired advice. To achieve this end, however, a natural language interface must be able to derive facts as well as a goal to prove from a single input question and must be able to use those facts in inferencing to avoid asking for irrelevant information.

These problems are currently being addressed in two ongoing projects at Columbia. The generation of explanations tailored to the system user is being addressed within the context of a student advising system which can provide advice about whether a student can or should take a particular course (McKeown, Wish, and Matthews 85). Natural language interpretation for expert systems is being addressed within the context of a tax advising system[2] which can provide assistance in filling out an income tax return (Datskovsky 85; Datskovsky and Ensor 86). Our examples come from a subsystem of the tax advisor, the dependency module, which can determine whether or not a user can claim another specified person on their federal tax return.

2 Comparisons with Natural Language Database Systems

In designing a natural language interface for an expert system one natural approach is to turn to the database environment for techniques where the development of natural language interfaces has been quite successful. In comparing the two environments, however, it immediately becomes apparent that there are significant differences between what is required for natural language in each case. The differences appear in the type of discourse (i.e., sequences of interaction) that will occur and in the results of parsing (i.e., the correlate of a natural language question in the underlying system).

2.1 Discourse

One main difference between the database and expert system environment is in the type of extended discourse possible with the system. The influence of the discourse situation on the meaning of an utterance and on the generation of a response has been noted by many researchers (e.g., Allen and Perrault 80; Appelt 81; Carberry 83; Cohen 78 Litman and Allen 84; McCoy 85). If a system can identify context and higher-level problems from the discourse that the user is attempting to solve through interaction with the system, it can supply a meaningful response.

In a natural language database system, a user is allowed to ask questions and the system will respond. An extended discourse, therefore, would consist of a sequence of questions. Whether or not questions in a sequence are related to each other depends upon whether the user is randomly asking questions or is gathering information to solve a problem. If the latter, then the sequence of questions may be related to the underlying problem.

For example, suppose the database contains information about course offerings in a university, as is the case in our student advising domain. If the user wants to determine which course to take next semester, s/he might ask the following sequence of questions to gather the information needed to make the decision.

1. What courses are offered next semester?
2. What are the pre-requisites?
3. Which of those courses are sophomore level

[1] The work described in this paper is partially supported by ONR grant N00014-82-K-0256 and by NSF grant IST-84-51438.

[2] The tax advisor, TAXPERT, is being developed primarily by AT&T Bell Laboratories (Ensor et. al. 86). Columbia's focus is on one module, the dependency module, and its natural language interface.

CH2345-7/86/0000/0241$01.00 © 1986 IEEE

courses?

4. What is the programming load in each course?

There are two main observations to be made. First, because the user is restricted to asking questions, the underlying problem being solved is never clearly stated. If the system wants to use that problem in deciding how to respond to a question, it must do quite a bit of guesswork in inferring the problem as well as criteria the user has for how the problem is to be satisfied. Second, since the user has responsibility for determining what information is needed to solve the problem and for producing the final solution, whether the questions in a discourse segment are connected depends upon the user. The user may alternate questions about several problems, making the task of inferring the underlying problem more difficult.

In contrast, in the expert system environment, the underlying expert system has responsibility for coming up with a solution to a given problem and thus, the natural language interface is aware of information needed to solve that problem. It can use that information to take responsibility for directing discourse towards the problem solution (see Matthews 85). Moreover, since the system must ask the user to enter information needed to do problem solving, the user has the opportunity to state the problem s/he is concerned with and any criteria s/he has for how the problem must be satisfied.

As an example, consider again the problem of deciding what course to take in a coming semester. If the underlying expert system can determine whether a student should take a course in the coming semester, we might expect extended discourse similar to that shown in Figure 1.

1) User: I'm a sophomore and just finished data structures. What courses should I take this spring?
2) System: Have you taken discrete math?
3) User: Yes.
4) System: Are there any courses you'd particularly like to take?
5) User: I'd like to take less programming courses.
6) System: I suggest you take fundamental algorithms and finite math which are both offered next semester. You should have finished both courses by the end of your sophomore year and only fundamental algorithms requires programming.

Figure 1: Expert System Discourse

This discourse segment is clearly concerned with a single purpose which is stated by the user at the beginning of the session[3]. This is the goal that the expert system must pursue and the ensuing discourse is directed at gathering information and defining criteria that are pertinent to this goal. Since the system must ask the user for information to solve the problem, the user is given the opportunity to provide additional relevant information. Even if this information is not strictly necessary for the problem-solving activity, it provides information about the user's plans and concerns and allows the system to select information in its justification which is aimed at those concerns. Thus, in the above example, the system can use the volunteered information that the user is a sophomore and wants to take less programming courses to tailor its justification to just those concerns, leaving out other potentially relevant information.

2.2 Parsing

Since the main purpose of a database system is to store and retrieve information, most database management systems provide formal query languages for searching and retrieving information from the database. In a natural language interface, a user asks a question in place of issuing a formal query. Parsing a question, then, requires producing as output the corresponding query. In producing the query, a parser must take into account the kind of search that is required and determine constraints on the search. Whether a parser uses a syntactic or semantic grammar, a top-down or bottom-up algorithm, its task remains the same: to produce the corresponding formal query.

The main purpose of an expert system is to apply large amounts of domain specific knowledge on a narrow real-world problem to arrive at a solution. Depending on how many problems it can solve, an expert system has one or more *goals* it can prove (e.g., one goal in MYCIN (Shortliffe 76) is to determine whether or not the patient has meningitus). We might expect a question, then, to indicate the goal the user wants proved.

For example, in our tax advising domain, one goal is **(claim ?x)** (i.e. to determine whether the user can claim whoever ?x refers to as a dependent). Possible questions for the domain would be of the sort "Can I claim my sister as a dependent?". If an expert system has a number of such goals, then parsing questions for this environment can be compared to parsing questions for the database environment, with instantiated goals replacing formal queries as output.

There are complications, however. In order to do problem solving, an expert system must gather information about the problem at hand. In the tax advising domain, for example, there are a number of tests that must be satisfied in order to allow a user to claim another as a dependent. The potential dependent must be a relative, a US citizen, if over 19 a student, etc. Existing expert systems either query the user for such information (e.g., MYCIN) or gather it from the environment (e.g., an online database as in ACE (Stolfo and Vesonder 82)). While a natural language interface may also have to ask questions of the user to gather the needed information, some facts may be provided in the initial question in addition to the system goal, thus reducing the number of queries the system must pose. In the question "Can I claim my sister as a dependent?", the user indicates that the potential dependent is a relative, in fact, a sister.

An interface for an expert system, therefore, must be able to produce a system goal to prove (analogous to a database query) and in addition, facts to add to working memory which the inference engine can use in the deduction process. This has no analogy in current natural language database systems but would be similar to allowing a user to both query and update the database with a single question (see Joshi 78 for a discussion of an approach to this problem).

2.3 Consequences

These differences have several consequences for natural language in the expert system environment. First, given that discourse segments revolving around an underlying problem the user wants

[3] Over a longer sequence of discourse, more than a single user purpose is likely to surface. We are concerned here with *discourse segments* which deal with a single or related set of purposes.

solved are very likely to occur, an expert system provides an ideal environment for addressing the problem of generating responses (in this case, explanations) that are tailored to the user's concerns. Second, interpretation requires the development of new techniques as opposed to simply adapting those from an existing environment. In particular, a system must have the facilities for deriving facts as well as the queried system goal from an input question. The following sections discuss ongoing work at Columbia that addresses these two issues. Since the approaches taken augment traditional expert system techniques, they are applicable to general problem solving systems as well.

3 User Modelling for Expert Systems

Given that natural language sessions with an expert system environment lend themselves to problem oriented dialog, one thrust of our work on natural language for expert systems involves the development of facilities to generate explanations that are tailored to the problems of the user. In order to generate tailored explanations, we have developed techniques to represent point of view in the underlying knowledge base to support different explanations, to derive a user goal underlying a discourse segment, and to relate the derived goal to different points of view to determine explanation content.

This work is being done within the context of an ongoing project to develop a dialogue facility for computer-aided problem solving. A student advising system is being developed which can provide information about courses and advice about whether a student can or should take a particular course. The system is currently structured as a question-answering system which invokes an underlying expert system on receiving "can" questions (e.g., "Can I take natural language this semester?") and "should" questions (e.g., "Should I take data structures?"). This production system uses its rule base to determine the advice provided (i.e., *yes* or *no*) and the trace of rule invocations is used to provide a supporting explanation of the advice.

3.1 Deriving the User Problem

If the system is to generate an explanation that addresses the higher level problem the user wants solved as well as constraints s/he has for solving the problem, the system must be able to identify what the higher level problem is. Since this is related to the user's goal in pursuing the dialogue, the large body of work on goal inference techniques (Allen and Perrault 80; Carberry 83; Litman and Allen 84) is applicable for deriving the user's goal. We have drawn heavily from Allen and Perrault's (80) work, making use of their plausible inference rules, representation of domain plans, and representation of speech acts as plans. While their work has been extremely useful, it falls short for our purposes in several ways. For example, their inferencing procedure derives a plausible goal for a user based on a single utterance, while we are interested in deriving a goal based on the current sequence of utterances[4].

Consider the discourse shown in Figure 2 below. Assuming that a database of domain plans common to the student advising domain is maintained, Allen and Perrault's techniques could be used to derive the domain goal shown following each question. But the explanation shown in Figure 2 (c) addresses not the derived goal of (c), nor any of the derived goals of the previous utterances, but instead addresses the higher level goal indicated by the derived goals of (a) and (b). The problem for responding to such goals in an explanation, then, is to be able to derive a higher level goal relating the goals of individual utterances.

a. S: I've read about the field of AI and I'm interested in learning more about it eventually. Is natural language offered next semester?
Plausible goal = take natural language
A: Yes.

b. S: Who is teaching artificial intelligence?
Plausible goal = take AI
A: Lebowitz this semester.

c. S: I haven't taken data structures yet. Should I take it this semester?
Plausible goal = take data structures
A: Yes, if you take data structures this semester, you can take AI next semester which is necessary for all later AI courses.

Figure 2: Goal Oriented Explanation

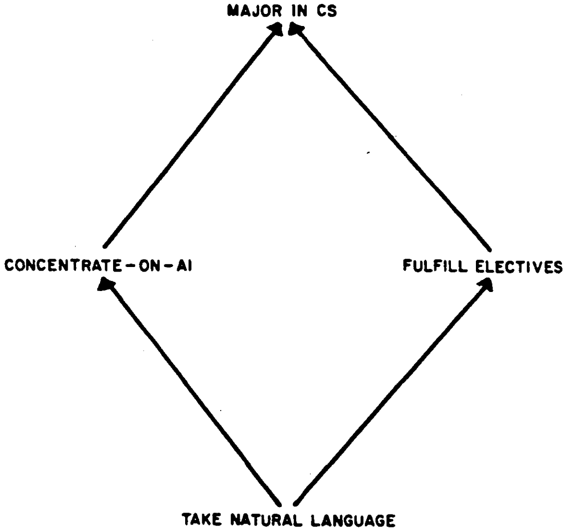
We use Allen and Perrault's rules to derive the domain goal of each individual utterance, which we term the *current goal.* We also identify a goal representing the discourse sequence which we term the *relevant goal* since it will be used to generate later explanations. Intuitively, the relevant goal is a higher level goal, if there is one, relating the goals of several utterances.

The process of determining the relevant goal involves the following steps. The current goal is first derived from the initial utterance. All higher level domain goals are then derived from the current goal using Allen and Perrault's *body-action* inference rule (i.e., if the user wants a step in the body of a plan to hold, it is plausible that s/he wants the action to hold). Any one of these is a candidate for the relevant plan. A derivation of the higher level plans for the utterance "Is natural language offered next semester?" is shown in Figure 3. Note that the action **take natural language** is a step in two separate plans, **concentrate-on-ai** and **fulfill electives**, and thus two parent paths are formed.

When the second utterance "Who is teaching artificial intelligence?" is entered, the current goal **take ai** is derived and all higher level goals derived (see Figure 4) from that using the body-action rule. The lowest level node where the two paths intersect becomes the relevant plan (**concentrate-on-ai** in this case). If the second utterance had been "When is operating systems offered?," the higher level goal **fulfill electives** would have been inferred since this is the only relation between the goals **take operating systems** and **take natural language**.

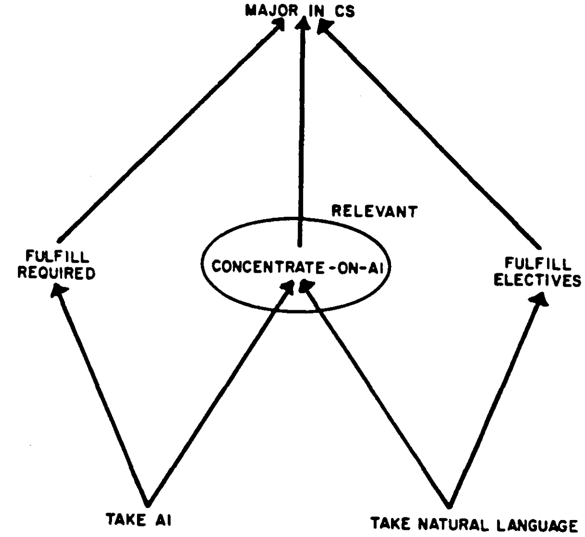
This method is essentially a search for the lowest common ancestor of the current goals of two consecutive utterances. When the third, or any subsequent utterances are encountered, the relevant goal is determined by performing the search for common ancestor using the previous relevant goal and the current goal of the new utterance.

[4] In this work, we restrict ourselves to a discourse segment that deals with a single or related set of goals. Over a longer sequence of discourse, topics may shift and the user may reveal very different goals across such boundaries. Detecting topic shifts and radical changes in goals is a difficult problem that we are not addressing.



"Is natural language offered next semester?"

Figure 3: Current and Higher Level Goals for Utterance 1



1: "Is natural language offered next semester?"
current goal = take natural language
2: "Who is teaching artificial intelligence?"
current goal = take ai
relevant goal = concentrate-on-ai

Figure 4: Relevant Goal for Utterances 1 and 2

Carberry (83) does present a method for tracking user goals over a sequence of discourse, building in the process a hierarchical model of user plans for the discourse. She uses this hierarchy and a set of focus heuristics to determine for the next incoming utterance which of several plausible plans the user could be focusing on. She does not specify which plan in the hierarchy best represents the overall discourse purpose and therefore should be addressed in succeeding explanations. Our model thus augments hers by providing this information.

3.2 How to Respond

In order to generate a response that addresses the derived relevant goal, the system must be able to identify information that is related to the goal. To do this, we are using intersecting multiple hierarchies to represent different points of view in the underlying knowledge base. The hierarchies are cross-linked by entities or processes (often courses in the student advisor domain) which can be viewed from different perspectives (and thus occur in more than one hierarchy). Hence to construct the content for explanation (c) in Figure 2 above, the system would extract information about data structures as it relates to the *AI topics* hierarchy. If the user had the goal of completing required courses as soon as possible, it would extract information from the *requirements* hierarchy. A diagram of a portion of these two hierarchies containing information for the two points of view is shown in Figure 5 below.

The partitioning of the knowledge base by intersecting hierarchies allows the generation system to distinguish between different types of information that support the same fact. From this partitioning, the system can select the portion that contains the information relevant to the current request and user goal.

After information from the appropriate hierarchy has been placed in working memory, the production system uses this information to derive the response (for explanation (c) of Figure 2, whether the user should take data structures). The trace of the reasoning is then available to provide the basis for the explanation, as is the case in traditional expert systems. Note, crucially, that information extracted from one hierarchy will allow a different set of rules to fire than will information extracted from another, thus producing different explanation content.

In constructing the explanation shown in (c) of Figure 2, the system first extracts the information shown in Figure 6 from the *AI topics* hierarchy. At this point, it also has information about what the student has already taken as well as the plan (here, *plan concentrate-on c:ai*) in working memory. After deducing that the

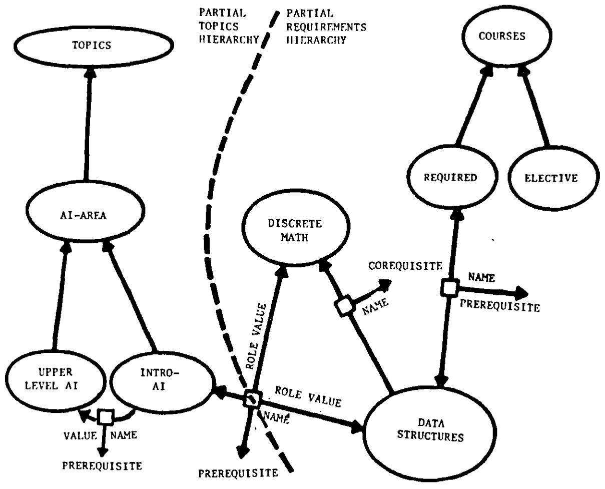


Figure 5: Representing Points of View

student *can* take these courses[5], the production system will attempt to prove that the queried action helps the user achieve his/her goals. The information shown in Figure 6, enables rule 8 to fire with **?course1** instantiated as **c:data-structures** and **?course2** instantiated as **c:intro-ai.** This rule concludes **(precursor c:data-structures c:intro-ai).** At this point rule 7 can fire, with **?future-course** instantiated as **c:intro-AI** and **?area** instantiated as **c:ai-Area,** and **?course** instantiated as **c:data-structures.** Thus, the advice is *yes* since the system can conclude **(advances-plan c:data-structures).** The instantiated rules can be used as the basis for the English explanation shown in (c) of Figure 2. Currently the inferencing component is complete, but the surface generator only partly implemented. While we can produce the content for explanation (c), we can not yet produce full English.

Note that if the user's plan had been to complete requirements, other rules indicating how the plan is advanced (such as "the course is a required course") would have fired since different information exists in working memory.

[5] Regardless of whether the user's queried action helps him/her achieve the relevant goal, if it is not permissible or will prevent the student from completing the major, the advice is always negative. Rules encoding such absolute constraints include "a course cannot be taken before its prerequisite", or "a course should not be taken if it prevents the student from completing requirements by the time s/he is a senior". Here, we assume, for convenience, that the student has already taken the prerequisites to data structures and is early enough in his/her program that s/he will be able to finish on time, and thus the absolute rules are satisfied.

Information Extracted:

```
(prerequisite c:data-structures c:intro-ai)

(superc c:intro-ai ai-area)
```

Rule 8:

```
(IF (pre-requisite ?course1 ?course2))
(THEN (precursor ?course1 ?course2)))
```

Rule 7:

```
(IF (plan concentrate-on ?area)
    (superc ?future-course ?area)
    (precursor ?future-course ?course))
(THEN (advances-plan ?course)))
```

Figure 6: Constructing Explanation Content

4 Interpretation in Expert Systems

In order to address the problem of both deriving a query (i.e., a goal to prove) as well as facts about the problem at hand from a single question, an underlying control structure for expert system inferencing that can support a natural language interface has been developed. Future work will address presuppositional analysis of noun phrases to add facts to working memory. This work is being done as part of the tax advising domain.

Since a user's question can indicate additional facts about the problem to be solved, the underlying expert system should be able to use these facts to more quickly arrive at a problem solution. If the system follows its normal problem solving chain, however, it may not make use of the given information until late in the deduction sequence in which case deduction paths that are irrelevant to the stated problem may be followed and information requested that is not needed due to the initial given information.

In order to make use of facts provided in the initial question and avoid unnecessary inferencing and system queries, an underlying control strategy is needed that can minimize queries that the system asks, maximize relevance of system queries to previous dialog, and provide information from any point in the underlying inference chain. Datskovsky developed DIRECTOR (Datskovsky and Ensor 86), embodying a control strategy for an expert system, that supports these requirements for a natural language system. She has developed three main techniques to handle these objectives: a combination of forward and backward chaining, heuristics to select which rule to fire next that minimize system queries and maximize relevance, and mechanisms to retrieve different types of information from an inference chain. DIRECTOR is currently being used as the inference engine in the dependency module of the tax advisor.

DIRECTOR's strategy for inferencing includes the following steps:

1. given all facts from the user's input, forward chain until all possible rules using existing information have fired.

2. then backward chain from the given goal using *existing* information in working memory.
3. when no further deductions can be made using existing information query the user on firing the next rule.
4. Given new input, start again at step 1.

DIRECTOR's heuristics apply at step 3. When no further rules apply using existing information, new information must be obtained from the user to continue problem solving and arrive at a solution. At this point, DIRECTOR has made maximal use of the initial given information. The rule that is fired next determines what to ask the user. As in standard expert system interfaces, the user will be queried about any facts in the left hand side of the rule not currently known by the system (i.e., not existing in working memory). Since Director wants to maximize relevance to the user's question, it chooses the rule whose left hand side contains the most known facts from the user's last input. While there is no guarantee that the solution will be found most quickly, this heuristic does guarantee maximum relevance to previous discourse and minimizes the number of queries the system will ask.

5 Conclusions

The work described here in both the tax advising domain and the student advising domain will allow for the ultimate development of full-blown natural language interfaces to expert systems. Such interfaces allow a user to receive advice more quickly, shortcutting questions the system would otherwise have had to ask, and provide the user with an explanation that addresses higher-level problems s/he is trying to solve.

References

(Allen and Perrault 80). Allen, J. F. and C. R. Perrault, "Analyzing intention in utterances," *Artificial Intelligence 15*, 3, 1980.

Appelt, D. E. "Planning Natural Language Utterances to Satisfy Multiple Goals." Ph.D. dissertation, Stanford University, Stanford, Ca., 1981.

(Carberry 83) Carberry, S., Tracking user goals in an information-seeking environment, in *Proceedings of the National Conference on Artificial Intelligence*, Washington D.C., August 1983, pp. 59-63.

(Cohen 78). Cohen, P., On Knowing What to Say: Planning Speech Acts, Technical Report No. 118, University of Toronto, Toronto, 1978.

(Datskovsky 85). Datskovsky, G., "Designing Natural Language Interfaces to Expert Systems", Columbia University Technical Report, 1985.

(Datskovsky and Ensor 86). Datskovsky, G. and J.R. Ensor, "Director -- An interpreter for rule-based programs", Columbia University Technical Report, 1986.

(Derr and McKeown 84). Derr, M.A. and K.R. McKeown, Using focus to generate complex and simple sentences, *Proceedings of COLING-84: Tenth International Conference on Computational Linguistics*, Stanford, July 1984, pp. 319-26.

(Duda 78). Duda, R.O., P.E. Hart, N.J. Nilsson, and G.L. Sutherland, Semantic Network Representations in Rule-Based Inference Systems, in D.A. Waterman and F. Hayes-Roth (eds.), *Pattern-Directed Inference Systems*, Academic Press, N.Y.,N.Y., 1978.

(Ensor et. al. 85). Ensor, J.R., J.D. Gabbe, and R.L. Blumenthal, "Taxpert - A framework for exploring interactions among experts," 11354-850130-02-TM, 1985.

(Joshi 78). Joshi, A.K., A note on partial match of descriptions: Can one simultaneously question (retrieve) and inform (update)?, in *TINLAP-2*, Illinois, July, 1978, pp. 184-6.

(Litman and Allen 84) Litman, D.J., and J.F. Allen, A plan recognition model for clarification subdialogues, *Proceedings of COLING-84: Tenth International Conference on Computational Linguistics*, Stanford, July 1984, pp. 302-11.

(Matthews 85) Matthews, K., Initiatory and reactive system roles in human computer discourse, unpublished manuscript, AT&T Bell Laboratories, 1985.

(McCoy 85). McCoy, K. F., The role of perspective in responding to property misconceptions., in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, Ca., August 1985.

(McKeown, Wish, and Matthews 85). McKeown, K.R., M. Wish, and K. Matthews, "Tailoring Explanations for the User," in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, Ca., August 1985. pp. 794-8.

(Stolfo and Vesonder 82). Stolfo, S. and G. Vesonder, "ACE: An expert system supporting analysis and management decision making," Technical Report, Department of Computer Science, Columbia University, 1982, to appear in *Bell Systems Technical Journal*.

Language Analysis in Not-so-Limited Domains

Paul S. Jacobs
Knowledge-Based Systems Branch
General Electric Corporate Research and Development
Schenectady, NY 12301 USA

Abstract

A fundamental problem in natural language analysis is the quantity of specialized knowledge necessary to understand language in a particular domain. Interfaces for specialized domains have proven successful, yet the amount of engineering required to develop these interfaces is often prohibitive. A design that takes advantage of general as well as specialized knowledge impacts both the portability of the natural language interface and the ease with which the interface can be extended within the domain.

Two aspects of language analysis systems are of particular importance with respect to this extensibility and adaptability. The first is the design of the lexicon, and the manner in which specialized lexical knowledge is handled. The second is in the method by which knowledge is combined during semantic interpretation. This combination, called *concretion*, uses specialized lexical knowledge to guide the semantic interpretation process.

1 Introduction

The term "natural language" still overstates the current ability of computers to communicate in human languages; yet the use of reasonable subsets of natural language in interfaces is a reality. With a great deal of development work, fairly robust and useful products that perform language analysis have been brought to market [5]. These early successes prove both the practicability of language analysis and the intensive effort required to achieve it. Surely the most important contribution of new natural language interface technology will be to minimize the individual effort needed for each interface application.

One area of application for natural language interfaces is as help facilities or interactive interpreters for operating systems [17][14][3]. There are dozens of such projects underway, and one finds published input and output from these systems most impressive. A major reason why these systems are absent from the marketplace is that they, like many AI systems, are fragile and fail to expand and adapt well. Thus a monstrous effort is required to achieve robustness.

This problem of fragility in natural language systems is addressed in a language analyzer called TRUMP (TRansportable Understanding Mechanism Package), a system designed to make use of general linguistic knowledge to serve as a natural language front end to a variety of applications. The operating system interface is one such application. Within this domain, the analyzer must be able to handle a range of input such as the following:

- (1a) Send the message to Jones.
- (1b) Send my job to the line printer.
- (2a) How can I delete a file?
- (2b) How can I delete a character?
- (3a) I want to get write permission on /tmp.
- (3b) Can you give me write permission on /tmp?
- (4a) What arguments does the command take?
- (4b) Which command gives the names of my files?

The effort required to build a robust analyzer is greatly increased by the fact that the relationship between verbs such as *send*, *delete*, *get*, *take* and *give* and their meanings is generally defined only in terms of each specialized usage. Because *delete a file* in an operating system has a meaning completely different from *delete a character*, for example, these expressions are often treated completely independently. And because *get write permission* and *give write permission* are specialized expressions, no knowledge about the verbs *give* and *get* is applied. The linguistic knowledge of the system is thus difficult to adapt because it contains knowledge of certain expressions but not knowledge *about language*.

The problem of applying this more general linguistic knowledge highlights two issues in the design of analyzers. The first is the structure of the lexicon; that is, how information about words and phrases is encoded to allow for a range of applications. The lexicon is the linguistic dictionary of the system. The second issue is the mechanism by which general interpretations are speciated or *concreted* to derive specific interpretations. For example, the verb *give* seems to refer to some general concept of *giving*; this concept is often concreted to more specific concepts such as *giving to charity*, *giving an order*, and *giving permission.* This issue is compounded by the fact that common verbs such as *giving* and *taking* are often used metaphorically to refer to events that seem to have little to do with possession, such as *kissing* ("giving a kiss") and *punching* ("taking a punch").

The method by which the TRUMP analyzer applies its linguistic knowledge facilitates the use of general knowledge even within specialized applications such as the operating system interface. For example, the system commands the use of the verb *send* well enough to understand the two senses in (1a) and (1b), and in fact makes use of some of its knowledge about the more general verb *give* in doing so. TRUMP can also deal elegantly with metaphorical examples such as (4a) and (4b); while the system uses some specialized knowledge about computer metaphors, it can apply the same metaphorical knowledge to understand both the *give* and *take* examples. This makes the analyzer more robust even within a specialized application without requiring extensive tailoring and anticipation of linguistic variants.

The next section describes how TRUMP's lexicon is designed to promote robustness and extensibility. Section 3 then describes how this lexical knowledge is concreted in the analysis process.

2 The Phrasal Lexicon Revisited

The approach to linguistic representation used in Berkeley's UNIX Consultant [17] stemmed from a variety of arguments, notably Becker's [1], which emphasized the role of quite specific phrasal knowledge in understanding language. The phrasal approach in UC was implemented in an analyzer called PHRAN [16] and a generator PHRED [7]. These programs shared a phrasal lexicon, which helped in the handling of a wide range of expressions, from "answer the door" to "Chinese restaurant" and "working directory". These expressions share the common aspect of being *nonproductive*; that is, their meaning seems not to be fully determined by their parts.

The problem with most such phrasal approaches is that they rely *too* heavily on the specialized phrasal knowledge. Expressions such as "send a message" and "send a job" were treated independently in UC, essentially as if they involved a different verb. The expression "give write permission" was independent of "get write permission". The problem seemed to be that the entries in the phrasal lexicon were consistently at a very specific level, yet it was difficult to incorporate them into more general linguistic knowledge. Much of the linguistic work on UC after PHRAN and PHRED [8][6] was devoted to making the "specialized" lexical knowledge flexible enough to cover the wealth of expressions considered. This work teases apart the phrasal knowledge of PHRAN/PHRED into a number of classes:

1. *word sequences*— Certain phrases, such as "by and large" and "let alone" are still treated simply as compound words, since there seems to be little benefit in trying to exploit their components.
2. *lexical relations*— Compound lexical items, such as "pick up" and "wipe out", are represented as lexical compounds of particular categories ("pick up" is of the *verb-particle* category). This allows these items to appear in a variety of surface forms, specifically, any surface form in which the *verb-particle* relation holds between the verb and the preposition.
3. *linguistic relations*— Linguistic relations link compounds which have syntactic as well as lexical constraints. For example, the *passive-verb-by-adjunct* relation specifies a verb in the passive voice used with an adjunct prepositional phrase with preposition "by".
4. *linguistic/conceptual relations*— Certain expressions, such as "giving permission", cannot easily be handled as exclusively linguistic constructs. The reason is that these expressions may be used with a variety of verbs in active or passive forms. This type of expression can be represented as *an abstract possession concept where the possessed is "permission"*, thus combining a class of concepts with a lexical category.
5. *conceptual relations*— Some "expressions" really do not seem to be lexical or syntactic entities at all, but distinguish different conceptual categories. For example the difference between "delete a file" and "delete a character" seems to be that the two different "deleting" concepts represent quite different subcategories of a general *deleting*, and it is possible that lexical knowledge plays no major role in distinguishing the two. In a similar fashion, "cut ones meat", and "cut ones hair" seem to express different *cut* categories, but these categories may best be distinguished by conceptual rather than lexical roles.

Entries in the lexicon combine three important types of information: (1) the lexical or syntactic knowledge required to identify an individual or compound lexical token, (2) a link between the token and an associated concept, and (3) knowledge placing that concept in the conceptual hierarchy. This is analogous to a typical dictionary, except that in the place of linguistic definitions are formal relationships among concepts known to the system.

The various types of lexical knowledge and their encoding in TRUMP will be considered below.

2.1 Basic lexical knowledge

Example: *Send* the message to Jones.

Desired effect: The verb *send* is interpreted as a verb, used similarly to *give*, with its meaning covering a variety of forms of transfers.

Lexical entry:

```
DEFINE-WORD send
    type: verb
    forms: past-tense sent
    concepts:
        sending
            (PARENT action
                    (ROLE-PLAY actor sender)
                    (ROLE-PLAY object sent-object))
            (PARENT physical-transfer-event
                    (ROLE-PLAY source sender))
        sending-transfer
            (PARENT sending)
            (PARENT transfer-event)
```

The above lexical entry provides the knowledge that the conjugated verb *send* refers to a concept *sending*, and possibly to a more specific concept *sending-transfer*. PARENT relationships place a concept as a subcategory of a more general, or parent concept. ROLE-PLAY relationships indicate correspondences between roles of more specific concepts, such as *sender*, and general roles, such as *actor*. These roles may have other information associated with them as well. Concepts such as *sending-transfer* have more than one parent, to indicate that their meaning combines that of at least two other concepts. In this case, the *transfer-event* concept represents the information that the destination of the *sending* was a *recipient*, rather than just a location. For a discussion of this form of conceptual hierarchy, see [6].

The entry for *send* provides sufficient lexical knowledge for handling expressions such as "send a file to the printer" and "send a note to Jones", as well as "send Jones a note". *Sending* is categorized as a physical transfer because the concept of *sending* "to" seems to describe a physical destination. Thus "send Jones a note" places "Jones" in the role of both *recipient* and *destination.* The details of how this is achieved depend on the intricacies of the conceptual representation, but this treatment seems to help with the problem of overly-rigid case structures pointed out by Langacker [9].

2.2 Word sequences

Example: The directory is *by and large* empty.

Desired effect: The phrase *by and large* is handled as an adverb corresponding in meaning to *mostly.*

Lexical entry:

```
DEFINE-COMPOUND by-and-large
        parts: word by, word and, word large
        type: adverb
        concept: most
```

This type of lexicon entry is relatively straightforward, because there is little flexibility in the way the expression is used and no real need to account for the role of each word. This knowledge is sufficient to account for the use of *by and large* as a synonym of *largely* or *mostly.*

2.3 Lexical relations

Example: I *sent* the message *out.*

Desired effect: The phrase *send out*, which may be expressed in a number of ways, is correctly understood as a special type of *sending.*

Lexical entry:

```
DEFINE-COMPOUND send-out
        parts: verb send, prep out
        type: verb-particle-relation
        concept:
            sending-out
                (PARENT sending)
```

The knowledge in 2.1 includes very general information about the verb *send* and the concept *sending*; more specific information is often contained in compound lexical entries, such as the one above for "send out". This knowledge specifies that *send-out* is a verb-particle compound that refers to the *sending-out* concept. This compound can appear in constructs such as "I sent the message out" and "Has the message been sent out to Jones?" The *sending-out* concept is categorized as *sending*; the difference is that this concept has no *transfer-event* subcategory so "I sent out Jones the message" does not work while "I sent out the message to Jones" does.

2.4 Linguistic relations

Example: The message *was sent out by Jones.*

Desired effect: The object of a prepositional phrase with preposition *by*, used with a passive verb, is understood as the *actor* of the action described.

Lexical entry:

```
DEFINE-COMPOUND passive-verb-with-by-adjunct
        parts: verb (voice passive), prep-phrase (prep by)
        type: verb-adjunct-relation
        concept:
            action
                (ROLE-PLAY prep-obj actor)
```

This type of lexical entry illustrates the flexibility required of the "dictionary" in a system like TRUMP. When a passive verb is used with a "by" prepositional phrase, the system can use this type of entry to understand the special role of the object of the preposition. This is dictionary-type information rather than knowledge about the syntax of English, because the prepositional phrase may appear anywhere an adverbial prepositional phrase ordinarily appears.

2.5 Linguistic/conceptual relations

Example: *Send* the message *back.*

Desired effect: The adverbial particle "back" is understood as reversing the direction of a transfer.

Lexical entry:

```
DEFINE-COMPOUND transfer-event-xxx-back
        parts: ref verb-adverbial-particle, adverb back
        type: transfer-event
        concept: return
```

The analysis of examples where lexical and conceptual knowledge combine may be complex. The meaning of the expression "send back" seems not tied to "send", but to the adverbial particle "back" when it modifies *any* transfer. Thus "send back", "sell back" "take back", and many other expressions are handled using the above linguistic/conceptual compound. The concept *return* encompasses the knowledge that the source and destination of the *transfer-event* being described are the reverse of some other *transfer-event.*

2.6 Conceptual relations

Example: *Send* Jones *the message.*

Desired effect: *Sending* here is understood as referring to communication of a message as well as a physical transfer.

Entry:

```
CONCEPT message-transfer
        (PARENT physical-transfer
                (ROLE-PLAY object message
                        (PARENT message)))
        (PARENT communication-transfer)
```

The expression "send a message" falls into the class of descriptions that are constrained by conceptual rather than lexical knowledge. Thus the information that "sending a message" describes a communication does not require any specific lexical compound. This knowledge is represented as above. This piece of knowledge specifies that a message transfer is both a physical transfer and a communication; thus "I got the message" may describe either a physical or communicative process; and "Send Jones the message" may describe the same command as "Tell Jones the message".

This section has outlined some of the basic lexical and conceptual knowledge necessary for a range of constructs. The verb phrases and particles used as examples here are complex in their behavior, and additional knowledge is required to account fully for their use. The lexical analysis presented here, however, is both sufficient for a technical domain and broad enough to apply across domains. For example, the concept of a UNIX message may be categorized as one type of message within this framework.

The discussion here has purposefully isolated the discussion of the lexicon from the process through which the lexical knowledge is used. The next section focuses on the application of lexical knowledge to the analysis of language.

3 Semantic Interpretation

Language analysis is frequently subdivided into two processes: PARSING and SEMANTIC INTERPRETATION. Parsing is the construction of a tree that accounts for each word and its syntactic role in a sentence; semantic interpretation is the derivation of a meaning representation. Some systems ignore one or the other; the assumption here is that both are necessary, and in fact, in the ideal case should interact (cf. [10]). This discussion concentrates on the derivation of meaning from lexical and linguistic relations because this is the troublesome aspect of constructing robust analyzers. While it is not difficult to construct a grammar that parses a tolerable subset of natural language inputs, it is extremely difficult to construct a system which makes tolerable sense of them.

The language analysis mechanism of TRUMP embodies three basic elements: (1) A pattern-matching or grammatical mechanism, which combines words into phrases and sentences, and instantiates linguistic relations such as those described in the previous section, (2) A mapping mechanism, which produces concepts and conceptual relations from linguistic structures, and (3) A mechanism that combines these bits and pieces of conceptual knowledge into an interpretation of the input. This mechanism, which I call CONCRETION, is the critical element of the next generation of language analyzers.

3.1 Concretion

Concretion is the process of taking abstract concepts and producing from them concepts that are more precise, or *concrete*. The motivation for this mechanism is strong in story understanding [11], because understanding a story seems to involve a continuous refinement of the major concepts into more specific categories. The following are two "story" examples:

1. Mary walked down the aisle. She picked up a can of tuna fish.
2. Bill went to the pool. He sat down in a chair.

In the first example, the concept *aisle* is being concreted to the concept of *supermarket-aisle*. Presumably, if new information had placed Mary in a church, a different kind of *aisle* would be derived. Intuitively, the term *aisle* does not seem ambiguous, rather it seems a more general concept than that which ordinarily results from a complete understanding. Similarly, the concept of a *chair* is a general one, which in the sentences above is probably concreted to *lounge-chair*. Certainly it does not refer to *electric-chair* or *desk-chair*.

The process of concretion is evident in understanding simple words and phrases in limited linguistic contexts as well. For example, the concept *cut* mentioned earlier invokes different meanings in "Mary *cut* the salami" and "Bill *cut* his hair". Similarly, the concept *deleting* may be concreted to *file-deleting* or *character-deleting*. The concept of *sending* may be concreted to *message-sending* and ultimately to *UNIX-error-message-sending*.

Most language analyzers do not really perform concretion. Unification-based systems [13][12][4] tend to refine semantic representations by adding semantic *features*, represented as variables with assignments. Some of the systems which use a KL-ONE knowledge representation [2][14] perform what is essentially concretion, but use specific interpretation rules to place concepts in more specific categories, rather than to attempt an algorithm for combining lexical and conceptual knowledge.

Concretion is important because it is the mechanism that allows general knowledge about language to apply at very specific levels of semantic interpretation. This is essential for interfaces, because it allows a core of linguistic and conceptual knowledge to be used for a variety of domains, and makes the addition of domain-specific linguistic knowledge easier. For example, knowledge about verbs such as *give* and *send* and their relation to *transfer-events* as described here applies to the UNIX domain as well as to other potential applications. It is hard to see how robustness could be achieved without this capability.

3.2 Metaphor and Indirect Meaning

Many of the examples given here emphasize the role of the interaction between general interpretations of words and more specific categories. This is not, however, the only way in which specialized constructs manifest themselves. Another common way is in metaphorical or indirect references. In fact, the "give permission" expression is an example of metaphor, because it does not seem that anything is being transferred, even in the abstract sense. Similarly, expressions such as "give a punch" and "give a kiss" are indirect descriptions of the events *punching* and *kissing*. As with the other types of specialized constructs described here, it is desirable to treat these expressions at a general level because of their apparent consistency. The idea is to have general knowledge about interpreting such expressions, even where specific lexical knowledge is used to trigger the general rule.

Looking back to examples (4) in the first section, we can see that there are many examples of verbs such as *give* and *take* in "computer talk" that refer to the input and output of operations. As with the "give a kiss" and "take a punch" expressions, it is problematic to treat these examples merely as special types of *giving* and *taking*. This is because "The command takes three arguments", even in the abstract sense, does not suggest that the command then *has* three arguments. In all these examples, what is at work is what I call a VIEW [15][8][6]. The execution of an operation is being VIEWed as a *giving* or *taking* in each expression, and the roles of input and output are VIEWed accordingly. Representing this metaphorical VIEW allows much of the same knowledge to be used in understanding numerous expressions of the same type.

In the systems interface domain, the concept of executing a system command may be a specific *execute-shell-command* concept. This concept in the conceptual hierarchy is a subcategory of the *execute-operation* concept, which may include individual functions and steps. We then have the following metaphorical knowledge:

```
(VIEW execute-operation causal-double-transfer
      (ROLE-PLAY input object-1)
      (ROLE-PLAY output object-2)
      (ROLE-PLAY user source-1)
      (ROLE-PLAY operation source-2))
```

The concept *causal-double-transfer*, which may seem somewhat arbitrary, is actually a very useful category. It represents any two *transfer-events* which are causally related. For example, *buying* and *selling*, *trading*, and *paying* may all be handled using this concept. Implicit here, but explicit in the knowledge base, is the information that *source-1* is both the *source* of the first transfer and the *recipient* of the second, while *source-2* is the *source* of the second and the *recipient* of the first.

This VIEW may be interpreted as follows: When any *execute-operation* concept is VIEWed as a *causal-double-transfer*, the *input*, *output*, *user*, and *operation* roles are VIEWed as roles of both transfers. Thus when a verb such as *give* or *take* is used to describe the *execute-operation* concept, the roles of *giving* or *taking* may be interpreted as the corresponding roles of *execute-operation*, as specified by the ROLE-PLAY relations above.

The VIEW above can be used to interpret constructs such as "the command takes three arguments", "the command gives you the file names", and even certain less common metaphors such as in " 'Echo' spits back a string." Notice that the use of *transfer-event-xxx-back*, as described in section 2, refers to a concept *return* that is entirely consistent with the concept of the *causal-double-transfer*. The expression "spits back", like "gives back", is interpreted as describing the second transfer event, where the *object* of both transfers is by default the same. The metaphorical VIEW can then be applied to result in the interpretation that the 'echo' command returns as output the same string that it takes as input.

The point of the VIEW relation here is to allow knowledge about transfers to be used to understand commands, even though the execution of these commands may not really be a transfer.

A VIEW is a powerful representational tool for encoding metaphorical relationships. A major problem with this tool, however, is that there are many metaphorical VIEWs associated with an abstract concept such as *transfer-event*, and the application of these VIEWs must be controlled. The solution to this problem is that VIEWs, like concretion, are performed only when specific lexical knowledge applies. Thus general VIEWs are triggered only by certain specialized knowledge. The algorithm through which this triggering is carried out is described in the next section.

3.3 The Interpretation Algorithm

As described earlier, the semantic interpretation process is driven by mappings from linguistic to conceptual relations, followed by the combination of these conceptual structures. The mapping process is initiated by the parser, which is not described here. For example, when the parser matches a grammatical structure which incorporates a relation between verb and indirect object, this structure is mapped into the *transfer-event* concept, with the indirect object playing the role of *recipient*. Thus " 'mv' takes two arguments" results, among other things, in the placement of the 'mv' command in the (metaphorical) role of *recipient*. The problem, then, is for the concretion mechanism to combine this knowledge with other knowledge in the system.

The details of parsing and semantic interpretation in TRUMP are omitted here; however, the following is a sketch of the iterative process that the system goes through as it scans its input and builds a conceptual interpretation:

1. *Mapping*. Apply mappings to produce new conceptual structures whenever there is a good chance that a new piece of linguistic information has changed the meaning interpretation; i. e., when each new important linguistic structure is instantiated. **Example:** In "Send Jones the message.", Mapping produces the concept *sending*, the concept *transfer-event (recipient Jones)*, and the concept *event (object message1)*.
2. *When to Concrete*. Perform concretion whenever new conceptual information might result in a more specific semantic interpretation. **Example:** Concretion of the event concept in the above sentence occurs after "send", "Jones" "message", and after the period.
3. *How to Concrete*. Produce the most specific conceptual interpretation of the input, with the appropriate roles filled, taking care to avoid conflicting interpretations. **Example:** Concretion combines *sending* with *transfer-event* to place Jones as the *recipient* of the message, and combines this with *event* to place the message in the *sent-obj* role.

The processing strategy of TRUMP is thus to apply its linguistic knowledge to parse the input, apply its lexical knowledge to produce conceptual structures, and use the conceptual hierarchy to build incrementally as specific an interpretation as is applicable. This approach allows the system to make maximal use of lexical knowledge of the various types described in section 2.

4 State of Implementation

The TRUMP parser and semantic interpreter are fully implemented and use the lexicon and concretion mechanism described here. The extensibility of the system is suggested by the fact that, once a set of linguistic constructs and their related conceptual knowledge are encoded, it becomes relatively easy to analyze many similar constructs. The catch to this method of handling specialized knowledge is that each specialized construct requires the proper encoding of certain general knowledge as well, and the improper treatment of this high-level knowledge may lead to the "house of cards" effect; that is, a modification at the high level negatively influences constructs that were previously handled correctly. Once the kinks are removed from the abstract knowledge, however, extension becomes easy. For example, the high-level knowledge about *transfer-events* has sufficed well in a variety of domains because it was carefully thought out beforehand.

The richness of lexical knowledge in TRUMP is behind two features of the system not described here. One is the ability to share a lexicon and grammar with a generator that produces natural language output. The development of TRUMP has followed from a knowledge representation framework called Ace [8] and a generator called KING [6], and the interface is thus specifically designed for both input and output capabilities. A second advantage of the system is the ease with which lexical knowledge is added "on the fly"; TRUMP includes a component that adds new information to the lexicon based on the dialogue with the user.

This system is being applied to a variety of domains, mostly dealing with natural language interactions about computers. In these domains, the benefit comes not only from the general knowledge about language, but also from knowledge about "computerese" that prevails regardless of the computer system.

5 Conclusion

The interaction of general and specialized knowledge is critical for language analysis, even in domains that are apparently constrained. The proper handling of general knowledge about language is essential both for portability and for the addition of new constructs within a given domain.

Two aspects of language analyzers are particularly useful in dealing with this interaction. The first is a lexicon that allows a variety of lexical collocations. The second is a semantic interpretation mechanism that facilitates concretion, the process of combining conceptual and metaphorical knowledge at various levels to produce a refined interpretation.

References

[1] J. Becker. The phrasal lexicon. In *Theoretical Issues in Natural Language Processing*, Cambridge, Massachusetts, 1975.

[2] R. Bobrow and B. Webber. Knowledge representation for syntactic/semantic processing. In *Proceedings of the National Conference on Artificial Intelligence*, Stanford, California, 1980.

[3] R. Douglass and S. Hegner. An expert consultant for the UNIX system: bridging the gap between the user and command language semantics. In *Proceedings of the Fourth National Conference of the Candadian Society for Computational Studies of Intelligence*, Saskatoon, Canada, 1982.

[4] J. M. Gawron, J. King, J. Lamping, E. Loebner, A. Paulson, G. Pullum, I. Sag, and T. Wasow. The GPSG linguistics system. In *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, Toronto, Ontario, 1982.

[5] L. Harris. Experience with INTELLECT: artificial intelligence technology transfer. *AI Magazine*, 5(2), 1984.

[6] P. Jacobs. *A knowledge-based approach to language production.* PhD thesis, University of California, Berkeley, 1985. Computer Science Division Report UCB/CSD86/254.

[7] P. Jacobs. PHRED: a generator for natural language interfaces. *Computational Linguistics*, 11(4), 1985.

[8] P. Jacobs and L. Rau. Ace: associating language with meaning. In *Proceedings of the Sixth European Conference on Artificial Intelligence*, Pisa, Italy, 1984.

[9] R. Langacker. An introduction to cognitive grammar. *Cognitive Science*, 10(1), 1986.

[10] M. Marcus. Some inadequate theories of human language processing. In T. Bever, J. Carroll, and L. Miller, editors, *Talking Minds: The Study of Language in the Cognitive Sciences*, The MIT Press, Cambridge, Massachusetts, 1984.

[11] P. Norvig. Six problems for story understanders. In *Proceedings of the National Conference on Artificial Intelligence*, Washington, D. C., 1983.

[12] F. Pereira and S. M. Shieber. The semantics of grammar formalisms seen as computer languages. In *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford, California, 1984.

[13] F. Pereira and D. H. D. Warren. Definite Clause Grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13, 1980.

[14] N. Sondheimer, R. Weischedel, and R. Bobrow. Semantic interpretation using KL-ONE. In *Proceedings of the Tenth International Conference on Computational Linguistics*, Palo Alto, 1984.

[15] R. Wilensky. KODIAK - a knowledge representation language. In *Proceedings of the Sixth Annual Conference of the Cognitive Science Society*, Boulder, Colorado, 1984.

[16] R. Wilensky and Y. Arens. *PHRAN-A Knowledge-based Approach to Natural Language Analysis.* Electronics Research Laboratory Memorandum UCB/ERL M80/34, University of California, Berkeley, 1980.

[17] R. Wilensky, Y. Arens, and D. Chin. Talking to UNIX in English: an overview of UC. *Communications of the Association for Computing Machinery*, 27(6), 1984.

Providing Expert Systems with Integrated Natural Language and Graphical Interfaces

Philip J. Hayes
Carnegie Group Inc.
650 Commerce Court at Station Square
Pittsburgh, PA 15219, USA

Abstract

Both natural language and graphical interfaces have been advanced as the best way to provide intelligent and user-friendly interfaces to knowledge-based systems. However, neither approach is sufficient on its own. This paper outlines the design[1] of a generic set of interface facilities that integrate the two modalities and allow each to contribute its relative strengths to the interface as a whole. A more detailed account is given of how the framework interprets in an integrated manner references to entities of the underlying application. The interpretation is done with respect to both the dialogue and visual contexts and for both natural language phrases and pointing events. As the detailed account of reference processing makes clear, the interface framework described in this paper depends in an essential way on the fact that the underlying application is knowledge-based.

1. Introduction

Through the use of AI techniques, intelligent knowledge-based systems are now helping people get their job done in an enormous variety of applications, including systems from manufacturing, medical, financial, and many other domains. However, the interfaces through which people interact with these expert systems often do not display a similar level of sophistication, either in intelligence or in their ability to interact (from a purely human-factors point of view). This has meant that the users have not been able to derive the maximum advantage from the expert systems, or in the worse cases, not derived any benefit at all. This is particularly true in those knowledge-based systems in which the initiative and responsibility for problem solving is divided between user and system. For such mixed initative systems, poor interfaces mean that the user cannot understand enough about what the system knows and is doing, and is also unable effectively to communicate his goals to the system. We believe that this mismatch of capability between interface and application is a major hindrance to the wider acceptance of knowledge-based systems.

The work described here is the beginning of an effort to remedy that mismatch. The work starts from the following assumptions about desirable characteristics of interfaces to mixed-initiative knowledge-based systems:

- *The interfaces should use graphical displays to convey information whenever appropriate.* Well-designed graphical output can give the user a much clearer overall picture of the system's view of the world than a scrolled dialogue (even a natural language dialogue). For instance, if the system is concerned with scheduling machining orders through various machines on a factory floor, then a graphical depiction of the machines, their layout, the current and planned location of orders, etc. would give the user a much clearer overall picture of the current state of planning than a textual description of the same information ever could.
- *The user should be able to access and modify the system's data and knowledge through direct manipulation of the graphical display whenever possible.* For instance, a user might be able to obtain and edit a table giving further information about a specific machine (its capacity, scheduled down-times, etc.) by clicking on the icon the scheduling system uses to represent it. Direct manipulation of this kind is often more efficient and usually easier to learn than any other kind of interaction.
- *The user should also be able to access and modify data and knowledge through a natural language dialogue.* Not all questions or updates can conveniently be made by direct manipulation. For instance, it might be quite hard to devise a piece of direct manipulation that allowed the user to find out if all scheduled orders could still be processed with any three (out of the current five) milling machines. Yet, as the previous sentence shows, it is straightforward to pose that question in natural language.
- *The natural language dialogue should integrate as closely as possible with the graphical display.* In particular, the resolution of anaphoric references should take account of what is on the screen. For instance, if the user says "the grinding machine" and only one is visible on the screen, then that should be interpreted as the one the user means, even if the system knows about four other milling machines related to the topic under discussion. Also the user should be able to point at the representation of an object on the screen instead of giving a natural language description of it. It may be a lot faster to point at an icon than to type "grinding machine number 4".

[1] Nothing in this document should be interpreted as a committment to nor an announcement of specific product enhancements by Carnegie Group Inc.

2. Overall Interaction Model

With these assumptions in mind, we are investigating generic support for a class of user interfaces suitable for a wide variety of knowledge-based applications. Individual interfaces built using this support would have two conceptually distinct, but highly integrated parts:

- **graphical display of the application domain**: The interface continuously displays a two dimensional view of the knowledge-based application's world. The view will not be fixed, but will be variable in the amount of detail shown and whether all or just a part of the world is shown. If specific entities in the world have internal structure, that may be displayed in overlaying windows. The main display may itself be split into windows for simultaneously displaying different parts of the world. Aids to locate a detailed view within its larger, containing scope will also be provided. The user is always free to alter the view of the world along any of the available dimensions. However, the system may also alter the display based on the dialogue in the dialogue window.
- **dialogue window**: The interface will also maintain a single thread dialogue with the user in natural language. The user's side of the dialogue may be conducted either through typed or spoken input. In either case, the complete dialogue thread will appear in a scrollable dialogue window. The user may refer to entities that are visible on the screen by pointing at them instead of speaking or typing out descriptions. Output from the application may also involve references or modifications to the display. In both cases, display references will appear explicitly in the dialogue transcript.

The overall feel of the interface is intended to be one where the user's main picture of what the application is doing and what information it is dealing with is provided by the display of the application's world. Navigation (in terms of both location and level of detail) around this world will be an important way for the user to obtain information that is not immediately visible. The dialogue window allows the user to ask for information that is inconvenient (or impossible) to obtain through navigation around the visual world, to make requests that he finds more natural in natural language, to discuss the world at a meta-level, etc.. We believe an interface of this kind will be extremely natural and productive to use. The style of interaction is analogous to that between two people poring over a diagram or having a discussion while writing on a blackboard - a very effective and natural form of communication. Moreover, unlike a paper diagram or blackboard, the application world display is active and can modify itself under control of the user or the interface/application system. This allows a very high bandwidth of communication and hence a highly productive interaction.

This kind of interface is highly appropriate to situations where spatial relationships between domain entities and movements of the entities through the space are an important aspect of the application's world. Factory job-shop scheduling is an example of such a domain; the paths from machine to machine and the distribution of orders around the shop are important features of the domain for the scheduling task. In addition, work by Negroponte [8] and the general success of the desktop metaphor in user interface design [10, 11] have shown that distributing information spatially is a useful information access aid, even if the spatial distribution does not correspond to any actual spatial relationships. Thus, there is every chance that the kind of interface envisaged here will also be useful in domains without a specific spatial orientation, such as a personnel database. This kind of information distribution is also useful for non-spatial information in worlds with a high degree of spatial information. For instance, in a factory scheduling application, it might be convenient for the user if he could find out about the scheduled down-time for a particular machine by "zooming in" on the icon that normally represents the machine. Such zooming could overlay the icon with a window in which scheduled down-time and other detailed information about that machine was displayed in a tabular fashion.

The dialogue component of the interface is intended to allow the user to access information (or issue commands) that is inconvenient to get at by navigation through the domain world display. For instance, in a factory scheduling system, the user might want to know whether one machine could take over the orders scheduled for another (perhaps because the first one had broken down). Since this question involves a dynamically constructed relationship between two machines, there is no obvious "place" to find it in the domain world. Instead, the dialogue facilities would allow the user to say (or type) something like "Can this machine *<point at one machine>* take over the orders scheduled for that one *<point at other machine>*". The answer could be displayed in text in the dialogue transcript, or as an overlay window on the graphical display, or as both. Note how intimately the displayed world and the dialogue interact to specify this information retrieval. The display is acting analogously to the physical context immediately surrounding human participants in a dialogue, i.e. something that can be referred to by abbreviated language or gesture in the knowledge that the other participant in the dialogue can see the same thing as you.

To support the construction of interfaces of this kind, we have designed a set of generic interface facilities for knowledge-based systems. The design views the facilities as extensions to the Knowledge Craft^{TM} and Language Craft ^{TM} products of Carnegie Group. Knowledge Craft is a knowledge engineering tool based on a schema-oriented knowledge representation language, called CRL^{TM}, and incorporating OPS-5 and Prolog inference engines and a powerful graphics system. Language Craft [7] is a tool system for building natural language interfaces based on case frame parsing techniques. These techniques make Language Craft a very robust language analyser, as is appropriate to deal with the user errors that inevitably arise with interactive dialogue. The specific capabilities that the design adds to these existing systems include:

- **representation and display of domain world**: This is a generic facility for mapping the world of objects, events, and states of a knowledge-based system onto a graphical display which could be browsed by the interface user at varying levels of detail. The facility is an extension to Knowledge Craft, and its existing powerful graphics system.

- **natural language analysis:** The dialogue window needs to be supported by a robust natural language analysis system of the type provided by Language Craft. This component of the design would extend Language Craft by integrating it more fully with Knowledge Craft. This will allow the natural language analysis to communicate properly with the underlying knowledge-based system and the world display system.
- **dialogue management:** The facilities in this area allow natural language input to the system to be interpreted with respect to the context built up during interaction in the dialogue window and the visual context of the world display. In addition to the interpretation of ellipsis already provided by Language Craft, the design would support a wide variety of anaphoric reference with respect to both kinds of context through an extension to Language Craft. The dialogue context can be altered by things that the user or system say and will be represented graphically in the domain world display. The display will be managed so that the complete dialogue context is always displayed.
- **integration of natural language and speech input:** While natural language is a highly expressive and natural form of input, it can be tedious to input via a keyboard. Spoken input is much more convenient. Our design integrates the natural language interpretation capability of the interface system with a speech recognition device. In other words, Language Craft will derive the meaning of the word sequences reported by the speech recognition device. Since speech recognition is inherently error-prone, Language Craft will be used to improve the overall recognition rate of the speech device. We will do this by making all the various word hypotheses considered by the speech device (with their associated certainties) available to Language Craft and let it make the final decision using its syntactic and semantic knowledge.
- **integration of natural language and pointing input:** Given the explicit graphical representation of context, and natural language input capability, particularly in its spoken form, it will be natural for the user to mix natural language and pointing input (e.g. "move that *<point>* from here *<point>* to there *<point>*"). Our design modifies Language Craft to deal with such pointing input.

Space does not permit us to go into detail on all these facilities. Instead in the remainder of this paper, we will focus on the way the design allows the user to refer to objects from the application domain. This will involve descriptions of the underlying natural language facilities, the way they handle anaphoric reference, and the way in which graphical pointing events can be substituted for ordinary linguistic anaphors. Before proceeding, we should reemphasize that the descriptions that follow refer to the design of a system, rather than one that is fully implemented.

3. Interpreting References to Application Domain Entities

When people engage in a dialogue, they share a context of past events, objects and events mentioned in the conversation, mutual assumptions about each other's goals and motivations, etc.. This shared context allows them to communicate intelligently with each other: by resolving pronouns or other abbreviated forms of reference into objects just mentioned, by completing elliptical questions by analogy to previous utterances, by recognizing that a statement is inconsistent with the speaker's assumed goals or beliefs and (internally) correcting it so it is consistent, etc.. For a machine to appear to communicate intelligently with a person, it needs to share context in a similar kind of way. Moreover, an ability to deal with anaphora and ellipsis allows the natural language input to be much more terse. It is much quicker to say or type "it" or "the machine" than always to have to say "grinding machine number 4".

The dialogue management facilities of the interface framework we are building are currently restricted to support for anaphora and reformulation ellipsis, though we hope in the future also to add more sophisticated facilities driven by a representation of the user's goals. This paper focuses on the way we deal with anaphora, i.e. the way we use context to interpret abbreviated descriptions of domain entities. Given the integration of graphical and natural language modalities in our interface model, we need to use two kinds of context to resolve anaphoric references:

- **dialogue context:** the set of entities mentioned or implied by the recent dialogue.
- **visual context:** the set of entities visible on the screen, plus perhaps other entities closely associated with them.

Moreover, the two modalities allow two kinds of anaphoric reference:

- standard natural language anaphora (pronouns, definite noun phrases, etc.)
- reference to objects by pointing at their images on the screen. Naturally, this kind of reference is resolved only against (a localized subset of) the visual context.

The following two subsections describe how each of these kinds of anaphor are handled by our design for interface facilities.

3.1. Natural language anaphora

In this section, we turn to the anaphoric interpretation capabilities we have designed as an extension to Language Craft. There is already a substantial body of work on the resolution of anaphora with respect to dialogue context (e.g. [4, 5, 6, 9]). Our approach does not represent a significant departure from this tradition. It uses techniques developed in the previous work to produce an anaphora capability suitable for restricted domain interfaces, rather than one that is completely general. In particular, we expect to make maximum use of the restricted domain semantics and not to use any techniques that would lead to noticeable (by the interface user) processing times. The novel aspects of our approach relate to the addition of visual context (though see [1, 5]).

Many current treatments of dialogue anaphora (e.g. [5, 9]) use the concept of a set of entities that are in the immediate *focus* of the context, plus others that are outside.the immediate focus, but may become focussed. The focus may change through nesting (subtopics or digressions) or by moving to related topics. Entities in the immediate focus may be referred to by pronouns. Entities that are outside the immediate focus may be referred to by abbreviated descriptions. We have adopted this kind of approach. The focus of the dialogue context is represented as a set of domain world entities. These entities may be referenced by pronouns or other abbreviated descriptions. There is a second set of entities, those related to the focussed entities by one of a (domain-specific) class of relationships, that may also be referenced by definite noun phrases, but not by pronouns. These entities form the *potential focus* [9]. Referring to an entity in the potential focus adds it the focus.

We can see how this works in the following dialogue fragment with a hypothetical factory scheduling system:

> User*: Are any grinding machines utilized less than 80%?*
> System*: Yes, grinding machine 4 has only 65% utilization.*
> User*: Has it had unscheduled downtime since Monday?*
> System*: No.*
> User*: Preventive maintenance?*
> System*: No. It is not scheduled until Friday.*
> User*: Who is the maintainer?*
> System*: Albert Smith.*
> User*: Ask him to do the preventive maintenance today.*

Here the grinding machine mentioned by the system becomes part of the focus and is referred to via the pronoun "it" by the user (and the system) in the question about utilization. The next question relies on Language Craft's current ability to handle reformulation ellipsis [3]. The system uses the context of the previous question to interpret the user's input as though he had said "Has it had preventive maintenance since Monday?". The user then goes on to refer to an entity that has not been mentioned in the dialogue so far, but is in the potential focus, viz. the maintainer of grinding machine 4, by the incomplete description "the maintainer" (the system would presumably know about lots of maintainers). There is only one maintainer in the potential focus, so the anaphoric noun phrase can be resolved correctly and unambiguously. Mentioning the maintainer entity adds it to the focus, so that the "him" in the next input can be interpreted correctly.

Integration between Language Craft and Knowledge Craft is important for the implementation of this anaphora mechanism. We represent the dialogue focus as a set of domain entities represented as Knowledge Craft schemas. The process of resolving an anaphoric referent against such a context involves integrating the constraints provided by the pronoun (e.g. "he" is a male person, "there" is a location) or noun phrase ("maintainer" is an entity in a maintaining relationship with some other entity) with the constraints provided by sentential context (e.g. in "ask him to do the preventive maintenance today", "him" must be an entity that can be asked to do maintence - an maintenance employee), and then finding items in the context that match the integrated description. Both the integration and the matching require the support of an inheritance mechanism which Knowledge Craft provides. For instance, in the above example, "him" has the constraint of being a male person inherent in the pronoun, and the constraint of being a maintenance employee from the sentential context. These two constraints are consistent since a maintenance employee ISA person and may be either gender. So the integrated description is a male maintenance employee. In the dialogue given above, this description would match (again through an inheritance process) with the focussed entity representing Albert Smith.

Knowledge Craft inheritance is also useful in specifying the relationships which define the potential focus (e.g. the relationship between a machine and its maintainer). Knowledge Craft relations are represented by user definable and modifiable schemas. This makes it convenient to attach such information to the relations.

The kind of interface we are considering here is unusual for natural language in that it has a representation of the world separate from the dialogue itself. This allows us to provide some interesting capabilities that have only been touched on in earlier work, in particular the work by Grosz [5] on task-oriented dialogues and Bolt [1] on an integrated natural language/graphical interface. First, we allow the user to refer to any entity visible in the world display by the minimum description necessary to distinguish it. For instance, if only one milling machine is shown on the display at any given time, the user would be able to refer to it by "the milling machine" rather than by typing its full name. The entities in the world display thus play a similar role to the potential focus and are treated in the same way by the anaphora resolution mechanism. In other words, the system will look for referents for definite noun phrase descriptions both among the entities in the potential focus and among the entities currently displayed on the screen.

The second useful capability opened up by the existence of the world display is an explicit representation of the dialogue focus through highlighting on the display. This allows the user to be clear at all times on what is the system's focus of attention, and helps prevent the kind of reference problems that could arise if the user thinks he has made a shift of focus, but the system fails to pick up on it. An even more intriguing possibility is to allow the user to edit the dialogue context explicitly. In this way, he could make up for any deficiencies in the system's focus tracking. He could also, as a part of browsing through the world display, explicitly set up the dialogue focus. For instance, if he set the focus of attention to include only a particular machine, then on the next natural language input, he would be able to refer to that machine by "it". The human factors of such an interface feature have never been examined and are hard to predict in advance. We, therefore, plan to determine its usefulness empirically.

In order to maintain a visual representation of the dialogue focus, the interface system will, when necessary, adjust the domain display in such a way that the dialogue focus remains a subset of the visual context. For instance, in the above dialogue example, when the user starts to ask about details of the grinding machine (utilization, unscheduled downtime, preventive maintenance, maintainer), the display of the machine (which we will assume was a named icon) would be expanded to or overlayed by a display of its attributes. The attributes that the user focusses on would be highlighted appropriately. Again, this is breaking largely unexplored ground from a human factors point of view. And we anticipate changes to the design based on experience with an implementation. The obvious danger is that the user might become confused or annoyed by changes to the display that he did not request directly. In addition, there is the potential for overly cluttered displays if this mechanism only ever displays additional entities and never removes any. The issue of when a defocussed entity can safely be removed from the display is a tricky one, involving unresolved research issues in dialogue management. Our current design does not address it.

3.2. Anaphora by pointing

One of the most interesting capabilities opened up by the kind of interface we are discussing is intermixing natural language input with pointing input. This would allow the user of a factory scheduling system to input, for instance, "Can this machine *<point at one machine>* take over the orders scheduled for that one *<point at other machine>*", where the pointing was done to the world display. We will call this kind of pointing *natural language pointing* and treat it as a kind of anaphoric reference. Natural language pointing is extremely useful in combination with speech input, allowing a very efficient combination of gesture and speech - the most natural way for people to communicate. Its effectiveness in an interface has already been demonstrated by work at MIT [1, 8]. It is somewhat less attractive for typed input because of the overhead involved in moving the hand from keyboard to pointing device and back again. However, work on the Scholar project [2] has shown that pointing can be used effectively with typed input in the context of maps in geography lessons.

Natural language pointing can be more efficient than speech alone because it is often faster to point at something than to identify it verbally (particularly when it is not in the immediate focus of the dialogue and so cannot be referred to by a pronoun). Moreover, pointing is a more direct form of identification than speech, so that its processing is likely to be faster and less error prone, considerably reducing the need for clarification dialogues, and hence enhancing communication efficiency still further. There are even some circumstances where pointing can communicate information that is very difficult to communicate through speech. For instance, pointing at a position on a map is very much easier and probably more accurate than trying to give the same information by speaking map coordinates into the system. In such circumstances, natural language pointing would also be convenient to use with typed input, particularly if all the pointing can be done after the entire sentence has been entered.

Though there are numerous advantages to natural language pointing, there are several issues which make its inclusion in an interface less than straightforward. In particular, it is necessary to:

- determine when pointing events are natural language pointing events;
- determine where the entities pointed at fit within the overall interpretation of the natural language input;
- identify which entity was actually pointed to (an issue when the visual representations of entities are nested within each other on the screen).

The difficulty of identifying pointing events as natural language pointing events stems both from ambiguity inherent in the use of the pointing device and in ambiguity in natural language as to whether a given phrase implies that a natural language pointing event will occur. In the kind of interface we are discussing, the pointing device will have other uses besides natural language pointing. In particular, it will figure prominently in the interface that allows the user to navigate around the world display. Pointing events are then potentially ambiguous between natural language pointing and these other uses of pointing. There are several potential solutions, none of which seems ideal:

- Make natural language pointing events identifiably different from others, for instance, by dedicating one mouse button (assuming there are several) to natural language pointing. This has the advantage of being clear, but the disadvantages of being fragile and difficult to learn (use of the wrong mouse button could produce highly unexpected and unintuitive results), and of reducing the options available for the world navigation interface.
- Overload some kind of neutral pointing event from the navigation interface - one that does some kind of selection without causing any specific action to happen. This has the same disadvantages in being fragile and hard to learn, and has the additional disadvantage of not being totally unambiguous, but the constraints it places on the navigation interface are different.
- Poll for the position of the pointing device when a deictic phrase is used. This avoids the disadvantages of the other alternatives, but runs into serious trouble because it is not always possible to identify deictic references in natural language just from the words involved. For instance, "there" may indicate a deictic reference or be a reference to an item in the dialogue context ("Order 17 is in the queue for ginding machine 4". "How long has it been *there*?"). It also reduces the freedom of the user in terms of the relative ordering of pointing and the deictic reference.
- Assume that all pointing events during (or close to, see below) natural language input are natural language pointing events. This has the advantages of simplicity, robustness, and lack of ambiguity. Its main disadvantage is making world navigation impossible during natural language input (even during a pause for thought during type in).

Our design currently calls for us to use the final alternative, but we regard the choice as an empirical matter and expect to experiment with several possibilities.

Once we have determined that a pointing event is a natural language pointing event, there is still the problem of matching it up with a phrase or hole in the natural language input. The co-occurrence of words and pointing events is useful information here, but does not give the whole story. All of the following examples and many other possibilities are plausible:

Can this machine <point at one machine> take over the orders
scheduled for that one <point at other machine>

Can this <point at one machine> machine take over the orders
scheduled for that <point at other machine> one

Can this machine take over the orders
scheduled for that one <point at one machine>
<point at other machine>

The last of these is most likely in a typed input situation where the user wishes to cut down on the overhead of moving between keyboard and pointing device. The only real invariant seems to be that the natural language pointing events will occur in the same order as the phrases (or holes[2]) in the natural language input to which they correspond. Moreover, the pointing events usually come during or immediately before or after the corresponding phrase. This latter fact means that information on the start and end time of words in the natural language input is important for the analysis of natural language pointing input. This information is naturally available for speech input, and our design calls for us to extend Language Craft to make it available for typed input.

There are thus two kinds of indication, neither of them conclusive, that a phrase in the natural language input corresponds to a natural language pointing event: the temporal co-occurrence of the phrase with the pointing event, and the actual form of the phrase itself. For instance, phrases with a demonstrative determiner ("this machine") are more likely to correspond to natural language pointing than phrases with a definite determiner ("the machine"). Once a candidate correspondence has been established between a phrase and a pointing event, it must be verified that the two are compatible. This process is very similar to anaphoric reference determination, except that the candidate referent is known in advance. It involves determining the constraints on the candidate phrase both instrinsically ("this machine" must be a machine) and from the sentential context (in the above examples "that one" must be a machine since orders are scheduled for it), and then checking if the entity pointed at met those constraints.

Since there is no conclusive way to determine which phrases in the natural language input correspond to pointing events, we have designed the following heuristic procedure for finding the correspondence.

1. list in left to right order all schemas representing phrases in the input that could potentially correspond to natural language pointing events;
2. list in left to right order all natural language pointing events;
3. form all lists of pairings between schemas and pointing events, such that:
    a. the left to right ordering is preserved for both schemas and pointing events;
    b. paired schemas and pointing events are compatible in the way described above;
4. if more than one list of pairings remains, choose those with the maximum number of schemas that correspond to phrases that are linguistically likely to be deictic (including all pronouns and noun phrases with demonstrative determiners);
5. if more than one list of pairings still remains, choose those with the shortest time mismatch between the pointing events and the phrases corresponding to the schemas;
6. if there is still more than one, ask the user to decide.

In the example above, little of the complexity of this algorithm is needed. There are only two pointing events and three schemas (corresponding to "this machine", "the orders", and "that one"), and hence there are three possible list of pairings, of which only the correct one satisfies all the constraints. More complex sentences could, however, require all the steps in the algorithm to find the correct pairings.

The final major issue in dealing with natural language pointing is that there can even be uncertainy as to which entity is being pointed to. In particular, if one entity has some kind of containment or subset relation with another, then the graphical representation of the contained entity may be nested within the graphical representation of the containing entity, so that pointing at the contained entity would be ambiguous between the two. For instance, an icon representing an individual machine may be graphically contained within an area which represents the factory shop of which it is a part, so that pointing at the machine icon might mean the machine or the shop. Sometimes, the corresponding natural language phrase or its sentential context can disambiguate. For instance, if the corresponding phrase was "this machine", there would be no confusion. If such disambiguation is impossible, then the system must query the user to resolve the issue. Modifying the above algorithm to use this approach means creating, for each potentially ambiguous natural language pointing event, copies of the pairing lists which differ only in the entity referred to by the ambiguous pointing event. There would be as many copies as there were alternative interpretations.

A final complication arises if the user desires to identify a group of objects or an area of the display through natural language pointing. For instance, he might want to ask about the average utilization of a group of machines. Conventional graphics applications provide ways of making group selections by incremental selection or selection by (usually rectangular) area. A similar approach could be used for natural language pointing, but it would eliminate some of the naturalness that we hope to achieve. Instead, our design allows freehand area designation by closed curves. In addition to being natural, this allows arbitrarily shaped areas to be designated as well

[2] Our current design requires some form of deictic expression and does not allow the user just to leave a gap in the input corresponding to the pointing event as in "Can <point at one machine> take over the orders scheduled for <point at other machine>". The semantic caseframe approach used by Language Craft is already quite capable of skipping over gaps and producing a partial analysis of the input. The additional ability required to match up gaps with natural language pointing would be for Language Craft explicitly to recognize that there is a gap in the input and to determine where in the input that gap is.

as being able to select groups of individual entities. As in ordinary natural language pointing, all such line drawing during or temporally close to a natural language input will be interpreted as a natural language pointing event corresponding to some deictic phrase in the input.

4. Conclusion

Although natural language is an important component of intelligent interfaces to knowledge-based systems, it is not in general adequate as an interface in and of itself. Modern AI workstations provide sophisticated graphical capabilities, which can communicate many kinds of information much better than natural language dialogue. Moreover, direct manipulation of graphical interfaces is often a convenient, natural, and efficient way for the user to access and update a system's information.

Fortunately, it is not necessary to make a either/or choice between natural language and graphical interaction. This paper has outlined a set of generic facilities for constructing combined graphical/natural language interfaces for a broad class knowledge-based systems.

We discussed in greater detail the aspects of those facilities that deal with the interpretation of references, both natural language and pointing, to entities in the application's world. In particular, we showed how standard dialogue anaphora resolution techniques can be integrated with the presence of a visual context and with pointing events into that visual context.

Unless interfaces to knowledge-based systems display a level of intelligence similar to the underlying systems themselves and make full use of the capabilities of the available I/O hardware, then the systems as a whole will fall far short of their potential impact. We believe that the way to construct interfaces that satisfy these goals lies in the direction we have outlined in this paper, and that this direction represents the future of user interfaces to knowledge-based systems.

References

1. Bolt, R. A. "'Put-That-There': Voice and Gesture at the Graphics Interface". *Computer Graphics 14*, 3 (1980), 262-270.

2. Carbonell, J. R. Mixed-Initiative Man-Computer Dialogues. 1970, Bolt, Beranek, and Newman, Inc., Cambridge, Mass., 1971.

3. Carbonell, J. G. and Hayes, P. J. "Recovery Strategies for Parsing Extragrammatical Language". *Computational Linguistics 10* (1984).

4. Charniak, E. C. Toward a Model of Children's Story Comprehension. TR-266, MIT AI Lab, Cambridge, Mass., 1972.

5. Grosz, B. J. The Representation and Use of Focus in a System for Understanding Dialogues. Proc. Fifth Int. Jt. Conf. on Artificial Intelligence, MIT, 1977, pp. 67-76.

6. Hayes, P. J. Anaphora for Limited Domain Systems. Proc. Seventh Int. Jt. Conf. on Artificial Intelligence, Vancouver, 1981, pp. 416-422.

7. Hayes, P. J., Andersen, P., Safier, S. Semantic Case Frame Parsing and Syntactic Generality. Proc. of 23rd Annual Meeting of the Assoc. for Comput. Ling., Chicago, June, 1985.

8. Negronponte, N. "Media Room". *Proceedings of the Society for Information Display 22*, 2 (1981), 109-113.

9. Sidner, C. L. Towards a Computational Theory of Definite Anaphora Comprehension in English Discourse. TR-537, MIT AI Lab, Cambridge, Mass., 1979.

10. Smith, D. C., Irby, C., Kimball, R., Verplank, W., and Harslem, E. "Designing the Star User Interface". *Byte 7*, 4 (April 1982), 242-282.

11. Williams, G. "The Lisa Computer System". *Byte 8*, 2 (February 1983), 33-50.

TEAM: An Experimental Transportable Natural-Language Interface

Paul Martin, Douglas E. Appelt,Barbara J. Grosz, Fernando Pereira

Artificial Intelligence Center,
SRI International
Menlo Park, California 94025

Abstract

This paper is a brief description of TEAM, a project whose goal was to design an experimental natural-language interface that could be transported to existing database systems by people who already possessed expertise in their use. In presenting this overview, we have concentrated on those design aspects that were most constrained by the requirements of transportability.

1 A Functional Description

A natural-language interface (NLI) to a computer database provides users with the capability of obtaining information stored in the database by querying the system in a natural language (e.g., English). The use of natural languages as a means of communication with computer systems allows users to frame a question or a statement in the way they think about the information being discussed, thereby freeing them from the need to know how the computer stores or processes the information. However, most existing NLI systems have been designed specifically to treat queries that are constrained in three ways: (1) they concern a single application domain; (2) they pertain to information in a single database; (3) they handle only a single task, namely, database query.[1] Constructing a system for a new domain or database requires a new effort almost equal to the original one in magnitude.

Transportable NLIs that can easily be adapted to new domains or databases are potentially much more useful than domain- or database-specific systems. However, because many of the techniques already developed for custom-built systems preclude automatic adaptation of the systems to new domains, the construction of transportable systems poses a number of technical and theoretical problems. In describing the transportable NLI system called TEAM (Transportable English database Access Medium), that was the focus and objective of a four-year project, this article emphasizes those choices in system design imposed by the requirement of transportability.[2] For some problems, the design decisions incorporated in TEAM are generally applicable to a wider range of natural-language processing systems; for others, we were forced to take a more limited approach.

1.1 Transportability

One of the major challenges faced in building NLIs is to provide the information needed by the system to bridge the gap between the way the user thinks about the domain of discourse and the way the computer handles the information it possesses about the domain. Existing databases employ different representational conventions, many of which favor storage efficiency over perspicuity. For example, one might encode geographic information about mountain peaks in Switzerland as part of a file of information about the mountain peaks of the world, identifying them with a "SWZ" in a COUNTRY field, or using a SWISS? feature field for which a "Y" indicates that a peak is in Switzerland and an "N" indicates it is not. Or the information might reside in a separate file on Switzerland, or one on Swiss mountain peaks. The kinds of queries a user might pose—for example "What is the highest Swiss peak?" "Are there any peaks in Switzerland higher than Mt. Whitney?" "Where is the Jungfrau?"—are equally appropriate for all the aforementioned encodings and the inputs to the NLI (an English query) remain unchanged. The output (commands to a database system), however, will be quite different. One of the main functions of the NLI is to make the necessary transformations, thus insulating the user from the particularities of the database structure.

To provide this insulation and to bridge the gap between the user's view and the system's structures requires a combination of domain-specific and general information. In particular, the system must have a model of the subject matter of the application domain. Included in this model will be information about the objects in the domain, their properties and relationships, and the words and phrases used to refer to each. Finally, the system must know the connection between entities in that model and the information in the database. A major challenge in constructing transportable systems is to provide a means for easy acquisition of domain-specific information.

TEAM is one of several recent attempts to build transportable systems (some of which are described elsewhere in this issue.) Different approaches to transportable systems reflect diverse conceptions of the kinds of skills and knowledge that might be re-

This paper has previously appeared in the quarterly newsletter of the IEEE Computer Society Technical Committee on Database Engineering, vol. 8, no. 3.

The work reported here was supported by the Defense Advanced Research Projects Agency under Contracts MDA903-83-C-0027 and DACA76-85-C-0004.

[1] This constraint is more limiting in many ways than the other two. For example, queries are typically treated largely in isolation; very few features of dialogue are handled. Since this remains a constraint in TEAM it will not be discussed further in this article.

[2] Space limitations have compelled us to omit many of the specific problems faced in this research; for a fuller treatment, please see the more extensive journal article [Gros86].

quired of those who will be doing the adaptations (in particular, whether they must have expertise in natural-language processing), and what parts of the system might change (in particular, whether the database can be restructured to fit the requirements of the NLI).

A major hypothesis underlying TEAM may be stated as follows: if an NLI is constructed in a sufficiently well-principled manner, the information needed to adapt it to a new database (and its corresponding domain) can be acquired from users who have general expertise about computer systems and the given database, but who do not have any special knowledge about natural-language processing or this NLI.

In testing this hypothesis, we also assumed (for both theoretical and practical reasons) that the database could not be restructured. Theoretically, it is the most conservative choice we could have made; it imposed general solutions upon certain issues of system design, because we could not restructure the data to alleviate problems of natural-language processing. Such restructuring can often bring about a closer match between the way information is stored and the way it is referred to in NL expressions. For instance, in the previous example, a database structure that includes the SWISS? feature field is more difficult to handle in a general manner than one that uses the COUNTRY field encoding. From a practical standpoint, the choice reflected our desire to provide techniques adequate to handle existing databases, some of which are quite large and complex, hence fairly difficult to restructure.

1.2 Using TEAM

The TEAM system is designed to interact with two kinds of users: a *database expert* (DBE) and an *end user*. The DBE engages in an acquisition dialogue with TEAM to provide the information needed to adapt the system to a new database, and, when desired, to expand its capabilities in answering questions about a database (e.g., by adding new verbs or synonyms for existing words). Once a DBE has provided TEAM with the information it needs about a database and domain, any number of end users can use the system to query the database.

The TEAM system thus has two major modes: acquisition and question-answering. The acquisition dialogue with the DBE is oriented around the database structure. It is a menu-driven interaction through which the DBE provides information about the files and fields in the database,[3] the conceptual content they encode and how they encode it, and the words and phrases used to refer to these concepts. Hence the DBE must know about the particular database structure and the subject domain its information covers, but he does not need to know how TEAM works or any special language-processing terminology.

The question-answering system consists of two major components: (1) the DIALOGIC system [Gros82] for mapping natural-language expressions onto formal logical representations of their meanings; (2) a schema translator that transforms these representations into statements of a database query language. DIALOGIC and the schema translator require both domain-specific and domain-independent information. The requisite domain-independent information is part of the core TEAM system; the domain-specific information is obtained by the acquisition component.

1.3 A Sample Database

We will use the database shown schematically in Figure 1 to help illustrate various aspects of TEAM. This database comprises four *files* (or, *relations*) of geographic data. The first file, WORLDC, has five *fields*—NAME, CONTINENT, CAPITAL, AREA and POP; respectively, they specify the continent, capital, area, and population for each country in the world. Various mountains in the world are represented in the second file, named PEAK, along with their country, height, and an indication as to whether they are volcanic. The third file, named CONT, shows the hemisphere, area, and population of the continents. The fourth file, BCITY, contains the country and population of some of the larger cities of the world. Because several files may have fields with the same names, TEAM prefixes file names to field names to form unique identifiers (e.g., WORLDC-NAME, PEAK-NAME, CONT-POP, BCITY-POP); we will do likewise in our discussion.

TEAM distinguishes among three different kinds of fields: feature, arithmetic, and symbolic. *Feature fields* contain true/false values indicating whether or not some attribute is a property of the file subject. PEAK-VOL and CONT-HEMI are feature fields. *Arithmetic fields* contain numeric values on which computations (e.g., averaging) can be performed WORLDC-AREA and PEAK-HEIGHT are examples of arithmetic fields. Let us note, however, that a field containing social security numbers would be treated more naturally as a symbolic field than as an arithmetic field, because it is unlikely that any arithmetic computations would be done on such numbers. *Symbolic fields* typically contain values that correspond to nouns or adjectives denoting the subtypes of the domain denoted by the field. WORLDC-NAME and PEAK-COUNTRY are examples.

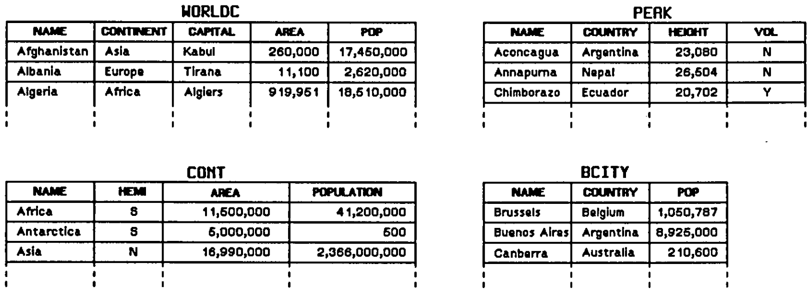
More information can be gleaned from a database than simply what the individual files contain. For instance, the continent on which a peak is located can be derived from the country in which it is located and the continent of the country. Likewise, the hemisphere in which a country is located can be determined from the continent on which the country is located and the hemisphere of that continent. TEAM allows the DBE to specify *virtual relations* that convey such additional information.

2 The TEAM System Architecture

The design of TEAM reflects several constraints imposed by the demand for transportability; our discussion will emphasize those aspects of the design. The need to decouple the representation of what a user means by a query from the procedure for obtaining that information from the database obviously affected the choice of system components. In addition, the need to separate the domain-dependent knowledge to be acquired for each new database from the domain-independent parts of the system influenced the design of the particular data structures (or "knowledge sources") selected for encoding the information used by these components.

Figure 2 illustrates the major processes of TEAM, the various sources of knowledge they use, and the flow of language-processing tasks from the analysis of an English sentence to the generation of a database query. The rectangular boxes represent

[3] TEAM currently assumes a relational database with a number of files. No difficult language-processing problems would result from conversion to other models.



WORLDC

NAME	CONTINENT	CAPITAL	AREA	POP
Afghanistan	Asia	Kabul	260,000	17,450,000
Albania	Europe	Tirana	11,100	2,620,000
Algeria	Africa	Algiers	919,951	18,510,000

PEAK

NAME	COUNTRY	HEIGHT	VOL
Aconcagua	Argentina	23,080	N
Annapurna	Nepal	26,504	N
Chimborazo	Ecuador	20,702	Y

CONT

NAME	HEMI	AREA	POPULATION
Africa	S	11,500,000	41,200,000
Antarctica	S	5,000,000	500
Asia	N	16,990,000	2,366,000,000

BCITY

NAME	COUNTRY	POP
Brussels	Belgium	1,050,787
Buenos Aires	Argentina	8,925,000
Canberra	Australia	210,600

Figure 1: Sample Database

the processes, and the ovals to their right, the various knowledge sources. The acquisition box on the right points to those knowledge sources that are augmented through interaction with the DBE. All other modules and knowledge sources are built into TEAM and remain unchanged during acquisition.

In this section we will look at the TEAM system from several angles. To begin, we will sketch the overall flow of processing during question-answering, describing the various processes involved in transforming an English query into a formal database query. Because the particular *logical form* (LF) TEAM uses to encode the meaning of a query plays a crucial role in mediating between the way queries are posed and the way information is obtained from the database, it affects the design of several components of the system. We then look in somewhat more detail at the data structures that encode domain-specific information. Finally, we discuss the overall strategy used for acquiring information about specific domains and databases.

2.1 Flow of Control

The flow of control during TEAM's translation of a natural-language query into a formal query to the database is illustrated as the path on the left side of Figure 2, from top to bottom. The transformation takes place in two major steps: first, a representation of the literal meaning of the query, or *logical form*, is constructed; second, this logical form is transformed into a database query.

The translation into logical form is performed by the DIALOGIC system, which comprises the following components, shown surrounded by the dotted box in Figure 2: the DIAMOND parser, the DIAGRAM grammar, the lexicon, semantic-interpretation functions, basic pragmatic functions, and procedures for determining the scope of quantifiers.

Since a description of DIALOGIC is provided elsewhere [Gros82], let us discuss here only those aspects of the system that were influenced by the development of TEAM. Two central data structures in DIALOGIC that are affected by TEAM's acquisition process are described: the *lexicon* and the *conceptual schema*. To understand the semantic and pragmatic components of TEAM, it is also necessary to appreciate DIALOGIC's separation of semantic interpretation operations into two main classes: *translators*, which define how the interpretations of the constituents of a phrase are combined into the phrase's interpretation; *basic semantic functions*, which are called by the translators to assemble the actual logical-form fragments that form the interpretations of phrases.

In brief, when the end user asks a query, DIALOGIC parses the sentence, producing one or more trees representing possible syntactic structures. The "best" parse tree, based on a priori syntactic criteria, is selected and annotated with semantic information [Robi82,Mart83]. Next, *pragmatic analysis* is applied to assign specific meanings that are relevant to the current domain to noun-noun combinations and to "vague" predicates like **HAVE** and **OF**.[4] Finally, the quantifier-scope determination process, after considering all possible alternatives, determines the best relative scope for the quantifiers in the query. The logical form thus constructed, using a set of predicates that are meaningful with respect to the given domain and database, constitutes an unambiguous representation of the English query.

The logical form produced by DIALOGIC is translated into a query in the SODA [Moor79] [5] database query language by the *schema translator*. In addition to the conceptual schema, the schema translator uses a database schema that furnishes information about the particular database structures. This schema, described briefly below, is also affected by the acquisition process.

Finally, the database query produced by the schema translator is given to SODA, which executes the query and displays the answer for the user. SODA was not developed as part of TEAM but was chosen for its features, which are consistent with the overall goal of transportability. SODA was designed for querying distributed databases and is capable of interfacing with several actual database management systems.

The processes TEAM executes in replying to an end user's query are similar to those that any custom-designed NLI would execute. What is different in the case of TEAM is that the

[4] We consider these predicates vague because they can be applied to many kinds of entities; they are replaced by "real" predicates during pragmatic processing.

[5] SODA is actually a query compiler that takes queries in a standard relational formalism and compiles them into optimized queries in the languages of other database management systems; both relational and codicil DBMSs have been accommodated. For our experiments, an interpreter that follows SODA commands to access a small database in primary memory was used in lieu of the actual SODA system.

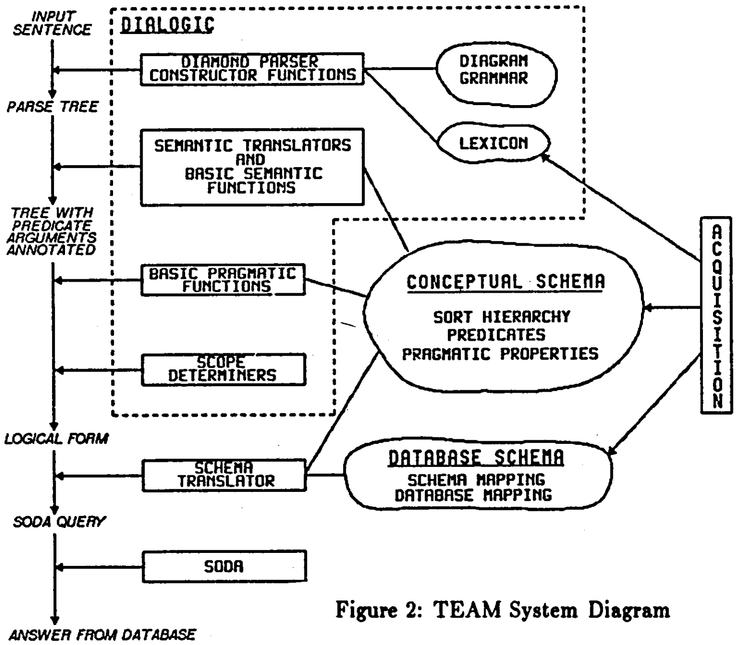


Figure 2: TEAM System Diagram

modules must be carefully designed to allow for maximal generality, which precludes many of the shortcuts that are common in custom-built NLI systems (e.g., LADDER [Hend77], PLANES [Walt75]). Two techniques that are ruled out are the using a *semantic grammar* and combining the determination of what a query means with the formulation of the DBMS query.

Semantic grammars are based on constituent categories that are chosen not for their ability to embody linguistic generalizations, but rather for the ease of parsing and interpretation that results when the grammar reflects the conceptual structure of the database domain. For example, instead of the general categories of "noun" and "verb phrase," semantic grammars may have categories such as "country" and "location specification." Such grammars are hopelessly tied to a single domain, and probably to a single database as well.

Efficiency also results from mapping a natural-language query directly into the code required for retrieving an answer from the database, but at the cost of being tied to a particular database. A number of database query systems (e.g., LADDER) construct a query directly while parsing the input with semantic grammar rules, but without building any other representation of what the query means.

Although the SODA query that results from the analysis of an English query represents, at least in some sense, the intended meaning of the latter, it does so in a way that directly reflects the structure of the database being queried. Consequently, if two databases encode the same information in different structures, the result will be two different database queries for the same English sentence. For example, if a user asks "How many Swiss mountains are there?" the database queries generated in response to his query can look very different, depending on whether the tuples representing Swiss peaks are distinguished from those representing other peaks by their membership in a different relation, or by the presence of the word "SWZ" in a COUNTRY field.

The problem this creates is not just an aesthetic one: to acquire the semantic and pragmatic rules necessary for generating a database query directly from an English query, TEAM would have to ask the DBE about far more than the structure and contents of the database. Answering the essential questions for such an acquisition would require the kind of expertise in natural-language processing that TEAM is intended to render unnecessary. Thus, the demands of transportability preclude use of the SODA language as the primary representation of the meaning of queries.[6]

2.2 Logical Form

Logical form plays a central role in TEAM: it mediates between the way an end user thinks about the information in a database, as revealed in his queries to the system, and the way information can be retrieved through queries in a formal database-query language. The predicates and terms in the logical form for a particular query are derived from information in the lexicon and conceptual schema;[7], hence, the choice of logical form indirectly affects the design of those components of the system and determines, in part, the information the DBE must supply.

The logical form employed by TEAM is first-order logic extended by certain intensional and higher-order operators and augmented with special quantifiers for definite determiners and interrogative determiners. Much research has been done to devise appropriate logical forms for many kinds of sentences [Moor81], but that investigation lies beyond the scope of this article.

[6] In addition, DIALOGIC was designed to be a general language understanding system that can be applied to tasks other than database querying. Therefore, it was undesirable to restrict its application by choosing an unsuitable semantic representation.

[7] As noted previously, the specific form depends also on general syntactic, semantic, and pragmatic rules for English that are encoded in the various components of DIALOGIC.

2.3 What Information Is Acquired

2.3.1 The Lexicon

The lexicon is a repository of the information about each word that is necessary for morphological, syntactic, and semantic analysis. There are two classes of lexical items: closed and open. Closed classes (e.g., pronouns, conjunctions, and determiners) contain only a finite, usually small number of lexical items. Typically, these words have complex and specialized grammatical functions, along with [at least some] fixed meanings that are independent of the domain. They are likely to occur with high frequency in queries to almost any database. Open classes (e.g., nouns, verbs, adjectives) are much larger and the meanings of their members tend to vary, depending on the particular database and domain. Therefore, most closed-class words are built into the initial TEAM lexicon, while open-class words are acquired for each domain separately. However, there are a number of open-class words, such as those corresponding to concepts in the initial conceptual schema (see Section 2.3.2) and words for common units of measure (e.g., "meter", "pound"), that are so broadly applicable to so many database domains that they are included in the initial lexicon as well.

Lexical entries include those for the names of file subjects (i.e., the entities about which some relation contains information—e.g., peaks for PEAK, and countries for WORLDC in the sample database illustrated in Figure 1.3), field names, and field values. In addition, the DBE can supply adjectives and verbs, as well as synonyms for words already acquired (see Section 2.4). Associated with every lexical entry is syntactic and semantic information for each of its senses. Syntactic information consists of its primary category (e.g., noun, verb, or adjective), subcategory (e.g., count, unit, or mass for nouns; object types for verbs), and morphology. Semantic information depends on the syntactic category. The entry for each noun includes the sort(s) or individual(s) in the conceptual schema (Section 2.3.2) to which that noun can refer. Entries for adjectives and verbs include the conceptual predicate to which they refer, plus information about how the various syntactic constituents of a sentence map onto arguments of the predicate. Scalar adjectives (e.g., "high") also include an indication of direction on the scale (plus or minus).

2.3.2 Conceptual Schema

The *conceptual schema* contains information about the objects, properties, and relations in the domain of the database. It includes sets of individuals, predicates, constraints on the arguments of predicates, and the information needed for certain pragmatic processing. The informational content is similar to that commonly encoded in semantic networks, but the apparatus used is more eclectic. The conceptual schema consists of a *sort hierarchy* and descriptions of various properties of nonsort predicates.

The sort hierarchy relates certain [monadic] predicates that play a primary role in categorizing individuals. These are called *sort predicates* (represented here in italics as in *PERSON*). TEAM was designed with a considerable amount of this conceptual information built in. Figure 3 illustrates a portion of this hierarchy. Each line connecting levels of the hierarchy signifies a set-subset relationship between two categories of individuals. The sorts connected by the small arcs directly below the nodes are disjoint; that is, no individual can be in two sorts joined in this manner. The sort hierarchy grows as information about a database is acquired. The DBE is required to position some of the newly acquired concepts in their appropriate places in the hierarchy.

Each field in the database is associated with the sort of objects that can appear in that field. Several additional properties are associated with the sorts derived from symbolic fields and from certain kinds of arithmetic fields.

With each sort obtained from a symbolic field, TEAM associates a predicate that encodes the relationship between that sort and the sort of the file subject. For example, for the relation WORLDC in Section 1.3, which includes information about capitals and continents, the system would link the sort *WORLDC-CAPITAL* with the predicate **WORLDC-CAPITAL-OF** (in this article, predicates are shown in boldface), which takes two arguments: the first of sort *WORLDC-CAPITAL*, the second of sort *COUNTRY*. This link is used in handling queries like "What is the capital of each country in Europe?" In particular, it is used to determine what it means for a capital to be "of" a country, or for a country to be "in" Europe. Additional properties of the sort indicate whether individual instances of it can modify or stand for instances of the sort of the file subject (e.g., "European countries," but not "Europeans" can be used to refer to the countries *c* satisfying the predication (**CONTINENT-OF** *c* EUROPE)).

Sorts that correspond to arithmetic fields containing measures (e.g., length, age) also include information about both the implicit unit of measurement (e.g., feet, years), and the kind of thing being measured (e.g., linear extent, temporal extent).

Several other kinds of information are associated with nonsort predicates. A *delineation* specifies the constraints on the sorts for each of a predicate's arguments; multiple delineations are supported but cannot be described in this brief format. Predicates corresponding to comparative-forming adjectives (e.g., "tall") have two additional properties: a link to the predicate that specifies the degree (e.g., **PEAK-HEIGHT** in our example), and an indication of polarity along the scale being measured (e.g., plus for **TALL**, minus for **SHORT**).

2.3.3 Associated Processes

Several general predicates have semantic and pragmatic specialists associated with them. The semantic specialists are Is-semantics and Degree-semantics; the pragmatic specialists are the Genitive, Noun-noun, Have, Of, General-preposition, Time, Location, Do-specialist, and Comparative.

The Is-semantics specialist is associated with the predicate **IS** and propagates sort restrictions across all the variables that are being equated by the **IS** assertion. This specialist is invoked prior to pragmatic processing (hence the "semantics" label); it attempts to reconcile any conflicts it detects and may revise some sort predications on variables in the process. For example, it is used in processing the query, "What is the area of Nepal?" to ascertain that the variable corresponding to the "what" is a *WORLDC-AREA*, not a *CONT-AREA*.

The Degree-semantics specialist replaces the general predicate **DEGREE-OF** with a more specific one. For example, by determining that predication (**DEGREE-OF** *peak1 x*) refers to the predicate **PEAK-HEIGHT**—i.e., that it is equivalent to the predication (**PEAK-HEIGHT-OF** *peak1 x*)—the specialist allows TEAM to further constrain the sort of *x* to be a *linear-measure*, thus allowing the comparative specialist invoked during pragmatic processing to make the right choice between the

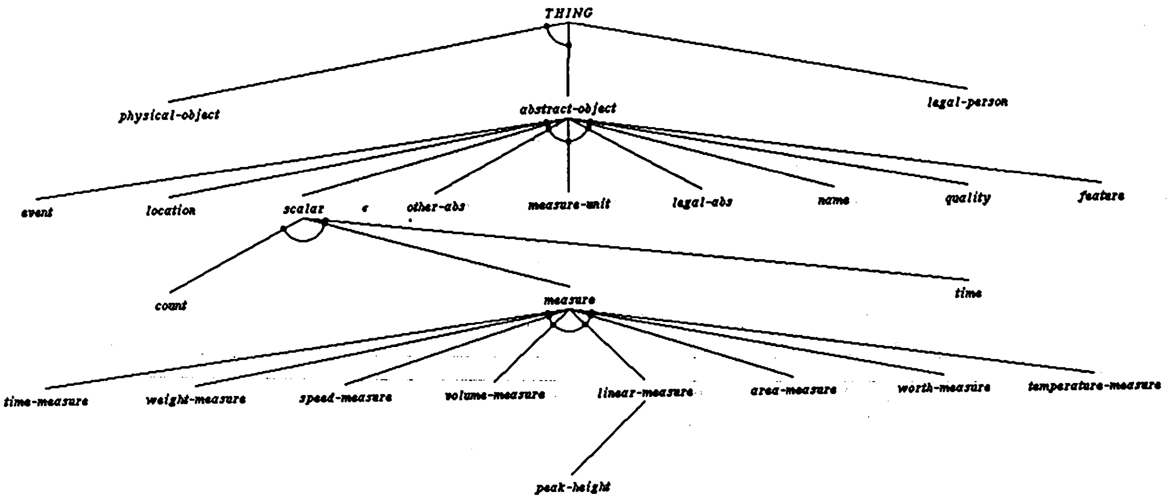


Figure 3: A Fragment of TEAM's Sort Hierarchy

alternatives of comparing the heights of two objects and comparing an object's height with a height value.

The Genitive, Noun-noun, Have, and Of specialists replace the vague predicates **GENITIVE**, **NN** (for noun-noun combinations), **HAVE**, and **OF** with more specific ones. The individual specialists differ only slightly, the differences reflecting the special restrictions associated with each construction.

The General-preposition specialist is associated with **ON**, **FROM**, **WITH**, and **IN**, converting these predicates into their appropriate domain-specific counterparts. For example, the Do-specialist determines that the phrase "countries in Asia" means those countries *c* for which the predication (**WORLDC-CONTINENT-OF** *c* **ASIA**) holds.

The Time-specialist and Location-specialist serve to map **TIME-OF** and **LOCATION-OF** into predicates that are appropriate for the database at hand. They can be invoked obliquely by the interrogative constructions "when" and "where."

The Do-specialist replaces the predicate **DO** (from the verb "do") with a more specific verb chosen from those acquired for a domain. Although "do" does not appear as the main verb very often in the database query task , the translators deduce its implied presence in some queries—for instance in such comparative questions as "What countries cover more area than Peru [does]?".

The comparative specialist examines the two arguments of a comparison to determine whether the comparison to be made is between two attribute values (e.g., Jack's height and seven feet) or between an entity and some value (e.g., Jack and seven feet). In the latter case, TEAM tries to identify the appropriate attribute of the entity (e.g., Jack's height).

2.3.4 Database Schema

The translation from logical form to SODA query requires knowing the exact structure of the target database and the manner in which the predicates appearing in the logical form are associated with the relations in the database. This information is provided by the *database schema*, which includes the following information[8]:

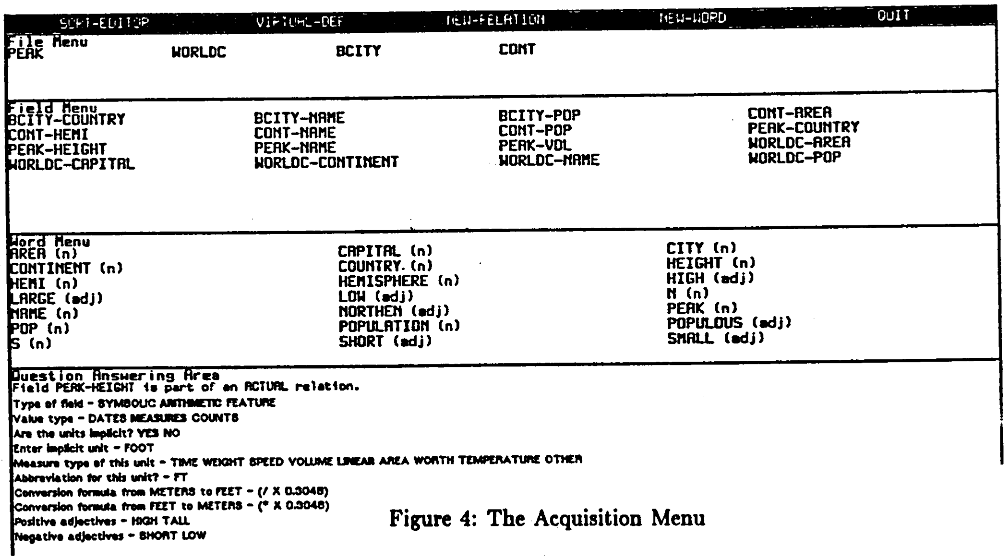
- Definition of sorts in terms of database relations (subject) or fields (and field value for sorts derived from feature fields).
- List of convenient identifying fields for each sort corresponding to a file subject or field.
- Definition of predicates in terms of actual database relations and attributes; this is done for predicates derived from both actual and virtual relations (for relation subjects and attributes).
- List of each relation's key fields.

The database schema relates all the predicates in the conceptual schema to their representation in a particular database. For each predicate, the database schema generates a logic formula defining the predicate in terms of database relations. For example, the predicate **WORLDC-CAPITAL-OF** has as its associated database schema a formula representing the fact that its first argument is taken from the WORLDC-CAPITAL field of a tuple of the WORLDC relation, and that its second argument comes from the WORLDC-NAME field of the same relation. If a predicate has multiple delineations—i.e., if it applies to different sorts of arguments (e.g., a **HEMISPHERE-OF** predicate could apply to both *COUNTRIES* and *CONTINENTS*)—the schema will include a separate definition for each set of arguments. In some cases (e.g., predicates resulting from the acquisition of some verbs and adjectives), the mapping associated with a predicate indicates that it is equivalent to another [conceptual schema] predicate with certain arguments set to fixed values.

2.4 Acquisition

The acquisition component of TEAM is crucial to its success as a transportable system. Recall that one constraint on TEAM is that the DBE not be required to have any knowledge of TEAM's internal workings, nor about the intricacies of the grammar, nor of computational linguistics in general. Yet detailed information, often necessarily linguistic in its orientation, must somehow be extracted from the DBE during an acquisition session. Furthermore, it is desirable that the acquisition component be designed to allow a DBE to change answers to questions and add information as he gains experience with TEAM and the types of questions that are asked by the end users.

[8] The schema translator also uses certain information in the conceptual schema, including taxonomic information in the sort hierarchy and delineation information associated with nonsort predicates.



Question Answering Area
Field PEAK-HEIGHT is part of an ACTUAL relation.
Type of field - SYMBOLIC ARITHMETIC FEATURE
Value type - DATES MEASURES COUNTS
Are the units implicit? YES NO
Enter implicit unit - FOOT
Measure type of this unit - TIME WEIGHT SPEED VOLUME LINEAR AREA WORTH TEMPERATURE OTHER
Abbreviation for this unit? - FT
Conversion formula from METERS to FEET - (/ X 0.3048)
Conversion formula from FEET to METERS - (* X 0.3048)
Positive adjectives - HIGH TALL
Negative adjectives - SHORT LOW

Figure 4: The Acquisition Menu

In an attempt to satisfy all these constraints, the menu-oriented system depicted in Figure 4 was developed. The acquisition system consists of a menu of general commands at the very top, three menus associated with relations, fields, and lexical items respectively, and, at the bottom, a window for questions and answers. When the DBE uses the mouse to select one of the items from the three menus, a set of questions appears in the question-answering area at the bottom of the display, to which he can then respond.

One of the general principles of acquisition is evident from this display, namely, that the acquisition is centered upon the relations and fields in the database, because this is the information most familiar to the DBE. The answers to each question can affect the lexicon, the conceptual schema, and the database schema. The DBE need not be aware of exactly why TEAM poses the questions it does—all he has to do is answer them correctly. Even the entries displayed in the word menu owe their presence to questions about the database. The DBE volunteers entries to this menu only in the case of verb acquisition, to supply an adjective corresponding to some noun already in TEAM's lexicon, or to enter a synonym for some lexicon-resident word.

The DBE is assumed not to have any knowledge of formal linguistics or of natural-language processing methods. He is assumed, however, to know some general facts about English—for example, what proper nouns, verbs, plurals, and tense are, but nothing more detailed than that. If more sophisticated linguistic information is required, as in the case of verb acquisition, TEAM proceeds by asking questions about sample sentences, allowing the DBE to rely on his intuition as a native speaker, and extracting the information it needs from his responses.

Virtual relations are specified iconically. The left side of Figure 5 shows the acquisition of a virtual relation that identifies the continent (PKCONT-CONTINENT, derived from WORLDC-CONTINENT) of a peak (PKCONT-NAME, from PEAK-NAME) by performing a database join on the PEAK-COUNTRY and WORLDC-CONTINENT fields. Similarly, the right side of Figure 5 shows the acquisition of the virtual relation that encodes the hemisphere (HEMIC-HEMI) of a country (HEMIC-NAME) by joining on the WORLDC-CONTINENT and CONT-NAME fields.

If he wishes, the DBE can change previous answers. Incremental updates are possible because most of the methods for updating the various TEAM structures (lexicon, schemata) were devised to undo the effects of previous answers before the effects of new answers could be asserted. Help information is always available to assist the DBE when he is unsure how to answer a question. Selecting the question text with the mouse produces a more elaborate description of the information TEAM is trying to elicit, usually accompanied by pertinent examples.

Finally, the acquisition component keeps track of what information remains to be supplied before TEAM has the minimum it needs to handle queries. The DBE does not have to determine himself how much information is sufficient; all he has to do is to perceive that no acquisition window indicates remaining unanswered questions. Of course, the DBE can always provide information beyond the minimum—for example, by supplying additional verbs, derived adjectives, or synonyms.

3 Conclusions

TEAM has been tested in a variety of multifile database domains by a fairly large number of people in addition to its original implementation team. While the testing has been much less rigorous than would be required for an actual product, enough has been learned to conclude that the basic ideas "work"—namely, that it is possible to build a natural-language interface that is general enough to allow its adaptation to new domains by users who are familiar with these domains, but are themselves neither experts on the system itself nor specialists in AI or linguistics.

TEAM handles a wide range of verbs, a capability that is absolutely essential for fluent natural-language communication. As it embodies no discourse model, its handling of pronoun resolution and determiner scoping is correspondingly limited. While its grammar coverage is quite extensive, the formalism used to represent it and the processes used to implement it are yielding to newer and more perspicuous designs[Shie84]. We are now investigating ways to provide transportability in natural-language systems that can interact with a variety of software services beyond database access and which more extensive discourse capabilities will be embodied.

Acknowledgments

Jerry R. Hobbs, Robert C. Moore, Jane J. Robinson, and Daniel Sagalowicz played important roles in the design of TEAM.

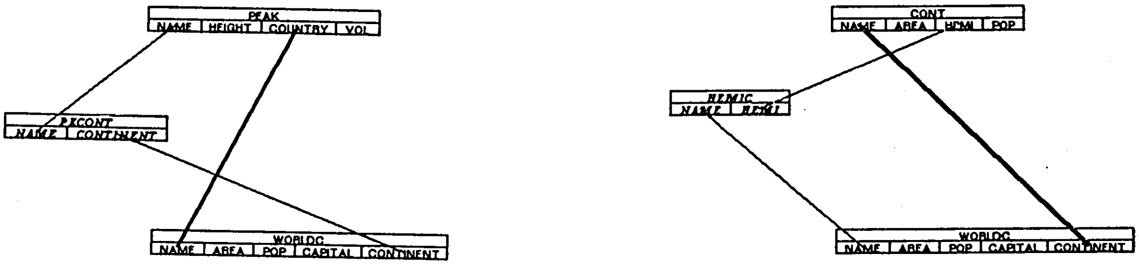


Figure 5: Acquiring the Virtual Relations PKCONT and HEMIC

Armar Archbold, Norman Haas, Gary Hendrix, Lorna Shinkle, Mark Stickel and David H. Warren also contributed to the project.[9]

References

[Gros86] Barbara Grosz, Douglas E. Appelt, Paul Martin, and Fernando Pereira. TEAM: An Experiment in the Design of Transportable Natural Language Interfaces. *Artificial Intelligence*(to appear)

[Gros82] Barbara Grosz, Norman Haas, Gary G. Hendrix, Jerry Hobbs, Paul Martin, Robert Moore, Jane Robinson, and Stan Rosenschein. *DIALOGIC: A Core Natural-Language Processing System.* Technical Note 270, Artificial Intelligence Center, SRI International, Menlo Park, California, November 1982.

[Hend77] Gary G. Hendrix. Human engineering for applied natural language processing. In *Proc. of the Fifth International Joint Conference on Artificial Intelligence*, pages 183–191, International Joint Conferences on Artificial Intelligence, Cambridge, Massachusetts, August 1977.

[Mart83] Paul Martin, Douglas Appelt, and Fernando Pereira. Transportability and generality in a natural-language interface system. In Alan Bundy, editor, *Proc. of the Eight International Joint Conference on Artificial Intelligence*, pages 573–581, International Joint Conferences on Artificial Intelligence, August 1983.

[Moor79] Robert C. Moore. *Handling Complex Queries in a Distributed Database.* Technical Note 170, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1979.

[Moor81] Robert C. Moore. Problems in logical form. In *Proc. of the 19th Annual Meeting of the Association for Computational Linguistics*, Stanford, California, 1981.

[Robi82] Jane J. Robinson. Diagram: a grammar for dialogues. *Communications of the ACM*, 25(1):27–47, 1982.

[Shie84] Stuart M. Shieber The design of a computer language for linguistic information. In *Proc. of Coling84*, pages 362–366, Association for Computational Linguistics, June 1984.

[Walt75] David Waltz. Natural-language access to a large data base: an engineering approach. In *Proc. of the Fourth International Joint Conference on Artificial Intelligence*, pages 868–872, International Conferences on Artificial Intelligence, September 1975.

[9] The development of TEAM was supported by DARPA contracts N00039-80-C-0645, N00039-83-C-0109, and N00039-80-C-0575; the National Library of Medicine NIH grant LM03611; and NSF grant IST-8209346.

SUPERCOMPUTING ARENA

Parallel Computation

TRACK CHAIR: Prof. Kai Hwang
University of Southern California

High Performance Numerical Architectures

TRACK CHAIR: Dr. Cleve Moler
INTEL Scientific Computers

Multiprocessors

TRACK CHAIR: Prof. Daniel Siewiorek
Carnegie Mellon University

Optical Computing

TRACK CHAIR: Dr. C. Lee Giles
AFORS NE

Networks

TRACK CHAIR: Dr. Michael Willett
IBM Corporation

Parallel Processing of a Knowledge-Based Vision System

D. I. Moldovan and C. I. Wu,

Department of Electrical Engineering -- Systems

University of Southern California

Los Angeles, CA 90089-0781

Abstract

A parallel processing algorithm and architecture for scene interpretation are presented in this paper. Scene interpretation and in particular, object classification, is an important aspect of high level image understanding. The algorithm for object classification presented in this paper is based on comparing image features with object models stored in a knowledge base and computing confidence values for object models. The algorithm is mapped into a multiprocessor system. Simulation results for this parallel processing technique are shown.

Index Terms: Symbolic processing, image understanding, knowledge-based parallel processing.

1. Introduction

A vision system consists of a number of numeric and symbolic algorithms aiming at understanding images. Although tremendous effort has been put in the image understanding field and many efficient algorithms have been developed, many fundamental problems still remain to be solved [Brad 82] [Wu 85]. Vision algorithms span the entire complexity range from simple purely numeric algorithms with small granularity to highly complex symbolic algorithms with large granularity for object classification [Neva 82] [Ball 82]. Historically, the focus of image understanding (IU) field has been directed more toward low level image processing [Dani 81] and consequently many parallel architectures have been proposed and built for these applications [Duff 81]. High level IU uses the results produced by image processing algorithms such as lines or regions and other features and through an inference process produces scene interpretations. This part of the vision system is more difficult and less explored than low level image processing [Wu 85]. It was believed by many that because of the symbolic nature of high level vision algorithms, parallelism was very limited in high level IU. In this paper we show that there is abundent parallelism in algorithms for object classification. We present a technique for scene analysis using the knowledge base approach and show the mapping of this algorithm into a parallel architecture.

A common technique used for knowledge processing is to compare the information on hand with a knowledge base. In the case of scene analysis the features extracted by image processing algorithms are constantly compared to known complex object models, and when the match is satisfactory and global consistency is achieved, the identity of image objects is found.

This paper is organized as follows: in section 2 an algorithm for object classification is presented followed by the description of the main processing modules, in section 3 the mapping of this algorithm into a multiprocessor system is shown, and in section 4 there are simulation results indicating the feasibility and the performance of this technique.

2. A Paradigm of Knowledge Based Vision Systems

2.1 System Organization

In figure 2-1 a flowchart of a vision system is shown [Binf 82]. The input consists of raw images provided by sensory devices. The purpose of this system is to obtain symbolic interpretations of images. The low and medium level processing such as early processing, segmentation, stereo, motion, occlusion, shading, etc. are responsible for extracting visual primitives from the original image such as regions, lines, and their attributes such as color, texture, size, shape, orientation, length, etc. Image interpretation is performed by grouping the visual primitives in various ways and attaching to them semantic labels under the constraints imposed by the knowledge base. As a result of this interpretation, object and scene descriptions are created which are constantly verified against expected object and scene models.

For our purpose the vision system from figure 2-1 may be partitioned into the following four main blocks: low level expert, short term memory, knowledge base, and high level expert. The *low level expert* incorporates all the modules required to transform raw images into intermediate symbolic representations. In early image processing, the nature of processing is iconic i.e. there is a direct relationship between physical storage location and image pixels [Rose 83]. The *short term memory* (STM) is the working memory of the vision system. It keeps all the symbolic representations made available by the low level expert and provides the processing environment for higher levels of abstractions [Mold 85a].

The *knowledge base* (KB), consisting of declarative and procedural knowledge, contains problem specific knowledge about the application domain. The declarative part is a collection of model objects and their relationships. Each model object is described by a frame and represents the classes of objects that are expected to be encountered during the processing of scenes. Model objects form an object hierarchy that plays the role of the hierarchical classifier. An unknown object is compared with each of the model objects and is classified to be the best matching class if evidence is sufficient. In this hierarchy, the scene object is the most general class and subsumes all other object classes in the image domain. The procedural part is linked to the declarative part by means of active demons in the frame structure and contains algorithms or production rules for computing attribute values, structures, and confidence scores. The *high level expert* (HLE) performs object classification through reasoning. It adaptively determines the appropriate levels of description based on the amount of information available. Once through the initial segmentation step, an input image is represented as a group of regions. Each region is represented as a frame with its own slot values for various region attributes such as area, centroid, and primary and secondary axes. Each region is initially classified as a scene object which is the most generic object class in the model object hierarchy. As the classification process proceeds, each region is further

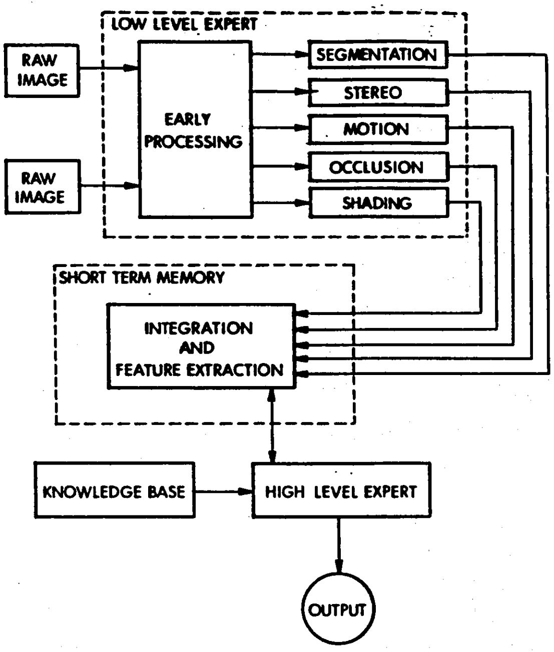


Figure 2-1: Main modules of a knowledge based vision system

hypothesized as subclasses of the established hypothesis. Then evidence is collected to confirm or deny these hypotheses. This cycle of hypothesis generation, evidence collecting, and confirmation or denial of hypothesis is repeated for each confirmed hypothesis until each region is classified as a terminal objects class of the object hierarchy, or no further subclassification is possible becaues of low confidence for the current classification. Then, as a second phase of scene analysis, the system checks for global hypothesis consistency. An object's class hypothesis is represented as an instance of the class model. Therefore, generation of a hypothesis corresponds to instantiation of the class model. The instantiation can be done in either a data driven or a model driven manner. In the data driven case, hypotheses are generated in an attempt to subclassify 'established' objects into their subclasses. In the model driven case, hypotheses are generated from the need to find new evidence for the existing hypotheses.

2.2 Classification Algorithm

The object classification algorithm can be summarized as the following six-step procedure where each step consists of one or more routines:

```
 Input: frame representation of image
        and frame representation of
        knowledge base;
Output: symbolic description of each
        image region;

Procedure:

Step 1: Hypothesis screener
        Perform a preliminary hypothesis
        check for each hypothesis to
        eliminate the apparently ridiculous
        image regions for each hypothesis;

Step 2: Evidence collection
        For each hypothesis perform specific
        image understanding algorithms to
        fill slot possibilities required to
        evaluate the confidence values;

Step 3: Establishment check
        Check if sufficient supporting evidenc
        exists for further classification usin
        a more specific hypothesis;

Step 4: Local confidence value evaluation
        Local confidence value is computed by
        optimizing the combination of slot
        possibilities;

Step 5: Subclassification
        Repeat recursively the classification
        algorithm for all the subclasses of
        this hypothesis;

Step 6: Global confidence value computation
        Compute the final confidence value for
        this hypothesis based on the local
        confidence value and the global confid
        values of its parent and children.
```

As an example, consider the classification of objects in the scene shown in figure 2-2. For processing this scene we use the knowledge base from figure 2-3. It is assumed that the image has been segmented into regions as shown in figure 2-4. The classification proceeds by applying the above procedure to each region. As a result of this process, each region is associated with a most likely frame from the knowledge base, and in the process it inherits the properties of all parent frames in the knowledge base hierarchy. For example, the path in the knowledge base most suitable to region 0 is 'scene-object', 'background', 'wood-object', and 'door'. In figure 2-5 are shown the confidence values [Shor 76] of all the frames in the model corresponding to region 0. The four numbers represent local measure of belief LMB, local measure of disbelief LMD, global measure of belief GMB, and global measure of disbelief GMD. The global confidence values (GCV) are computed as difference between the global measure of belief minus the global measure of disbelief.

$$GCV = GMB - GMD$$

The global measure of belief and disbelief are computed from local measure of belief and disbelief

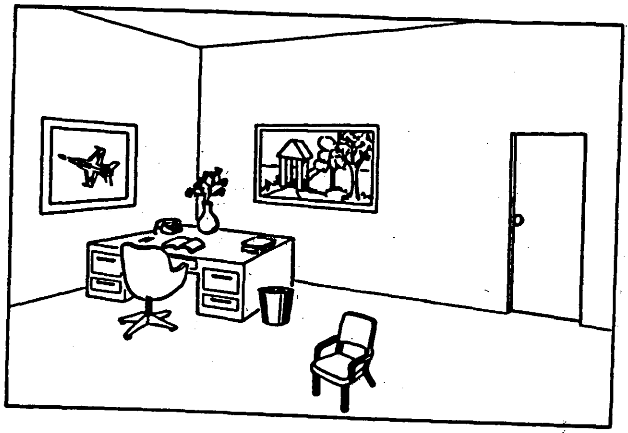


Figure 2-2: An example of scene

according to the formulas:

$$\mathrm{GMB} = \begin{cases} \mathtt{LMB,\ if\ this\ is\ a\ terminal\ node} \\ \mathtt{\Sigma LMB \oplus GMB(its\ children),\ \ otherwise} \end{cases}$$

$$\mathrm{GMD} = \begin{cases} \mathtt{LMD,\ if\ this\ is\ a\ root\ node} \\ \mathtt{\Sigma LMD \oplus GMD(its\ parent),\ \ otherwise} \end{cases}$$

$\mathtt{where}\ x \oplus y = x + y - x*y.$

The optimal path was selected by picking the model nodes with the largest global confidence value. The calculation of local confidence values is usually done in the short term memory by some vision algorithms. In this paper we do not discuss the processing performed.

Parallelism is achieved by simultaneous processing of many regions. After all the regions have been classified as described above, a global consistency check is necessary. If there are no conflicts the classification obtained holds, but if there are incompatibilities then more features are extracted from the image to help improve the decision process.

3. Mapping into hardware

Since the type of algorithms in various stages of a vision system varies considerable, no single parallel architecture can perform efficiently across the entire spectrum of vision algorithms. The architecture of a parallel processing system for the vision system described in the previous section is shown in figure 3-1. It consists of an intelligent memory [Fost 76], semantic network array processor (SNAP) [Mold 85b], a multiprocessor, and host computer. The host computer is mainly used to manage the vision tasks. The multiprocessor is a mesh-connected array of general purpose microprocessors. The SNAP consists of a square array of identical processing elements interconnected both locally and globally and having associative array capabilities. The multiprocessor acts as a controller for the SNAP array. The intelligent memory module can be accessed directly by the other three modules. The

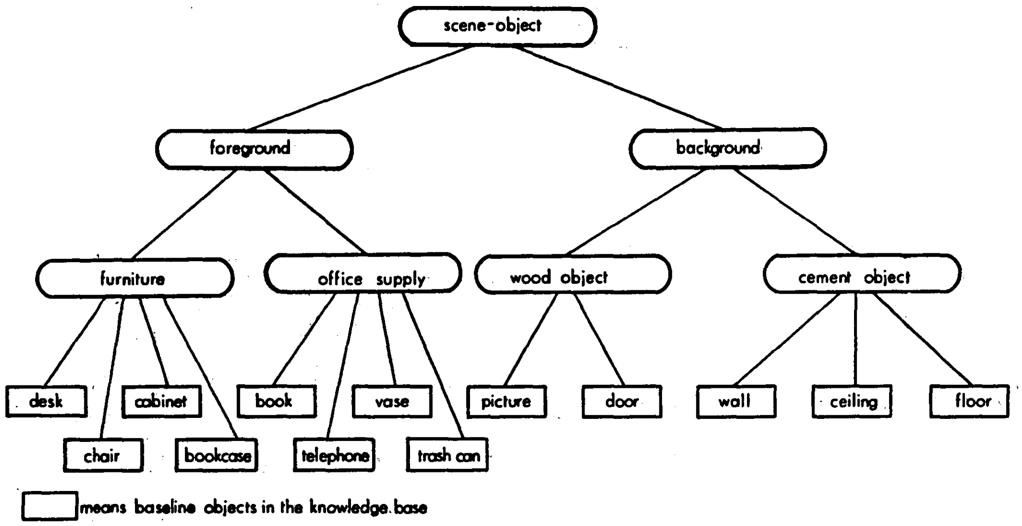


Figure 2-3: Model object hierarchy for office scene

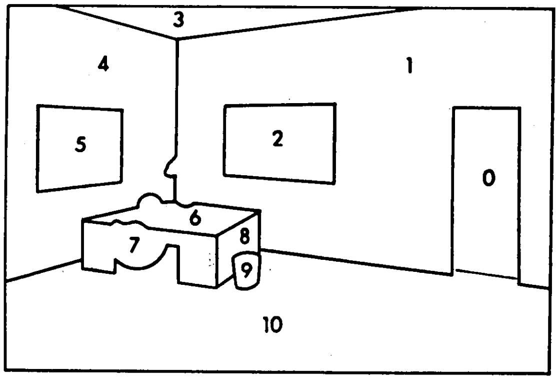


Figure 2-4: Segmented scene

intelligent memory consists of large number of processing cells with associative processing capabilities. The mapping of the vision system from figure 2-1 into this architecture is as follows: the low level expert is assigned to the intelligent memory [Rose 83]; the short term memory is assigned to the semantic network array processor [Dixi 84] [Mold 85a]; the high level expert and the knowledge base is distributed over the multiprocessor system. The SNAP and the multiprocessor perform symbolic functions while the intelligent memory performs most of the numeric functions. In this paper we are interested only in the operation of the multiprocessor system. Simulation results are described in the next section.

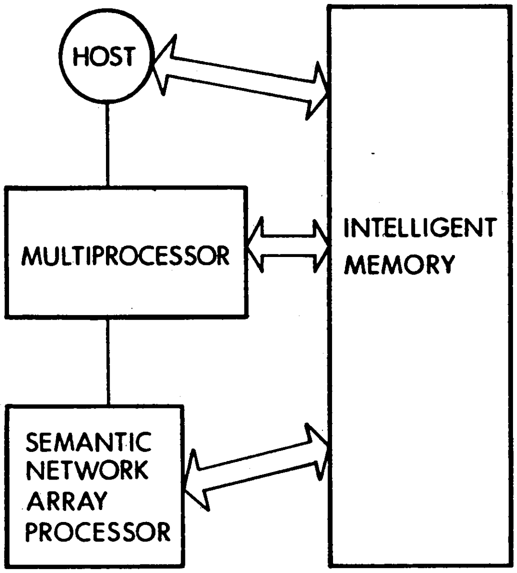


Figure 3-1: Vision Hardware System Architecture

4. Simulation Environment and Results

In this section we describe some simulation results obtained by implementing the knowledge based vision system on a multiprocessor. Through this simulation we want to demonstrate the feasibility of the

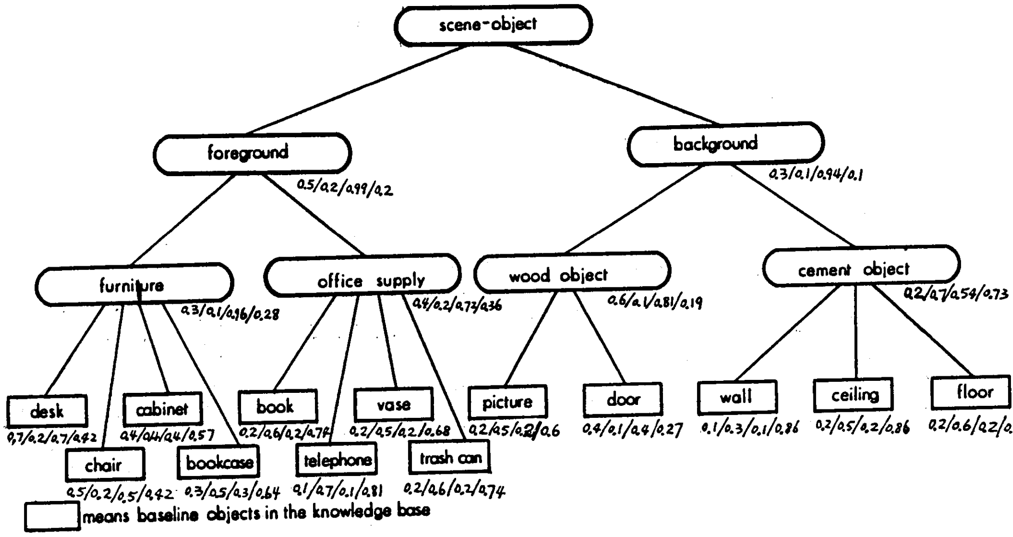


Figure 2-5: Confidence value for region 0 in figure 2-4

approach and to measure the speedup factor offered by the parallel scheme over the sequential processing. For this purpose we developed a program in Lisp [Gold 85]. This program simulates the operation of a four-node multiprocessor with mesh interconnection network. A system clock is used to coordinate all timing events. A time unit of system clock is defined as the time required to perform a simple arithmetic operation and transmit data between two adjacent processors. In each time unit it is determined if a processor is free or busy; if it is free and its queue contains a process then the processor is loaded, if busy it is determined what operation to be performed. A process consists of operations to instantiate a frame.

The inputs to the simulator are : (1) frame description of knowledge base vision system, (2) frame description of segmented scene, and (3) allocation policy of frames to processors. Three frame structures are used in our vision system: region frame, model frame and instance frame. The *region frame* is used to hold the spatial scene knowledge directly related with the image. The *model frame* contains domain knowledge and control knowledge as discussed before. The *instance frame* is the result of instantiating model frame and serves as the hypothesis of the region frame. In other words, the instance frame is the bridge between region frame and model frame. A sample structure of these three frames is shown in figure 4-1.

In its present form the simulator generates: (1) the final interpretation of the scene, (2) the parallel processing time and the sequential processing time, and (3) the utilization and load factors of each processor.

Simulation Example:

We simulated the interpretation of the scene shown in figure 2-2. For this purpose the knowledge base consisting of 20 frames was constructed as shown in figure 2-3. These frames were allocated to the four simulation processors according to three different allocation policies as indicated in table 4-1. In scheme (a) no parent and child or sibling frames are allocated to the same processor. In scheme (b) a totally random allocation was selected maintaining however a balanced load between processors. In scheme (c) an opposite policy from (a) was selected, in other words, as many related frames as possible were assigned to the same processor.

The simulator interpretes the scene by following the technique described in section 2. Each region is hypothesized on the hierarchical knowledge base and measure of belief and disbelief are computed for each instance. In step 1 of figure 4-2 is shown the sequence of instantiations for one region (region 0), and the belief and disbelief factors for each instance. The four numbers are in the following order: local measure of belief, local measure of disbelief, global measure of belief and global measure of disbelief. Based on the computed confidence value, the simulator computes the optimal path for each region, that is the path with the largest global confidence value. This is shown in step 2 of figure 4-2.

SCHEME	PROCESSOR			
	P1	P2	P3	P4
a	6, 10, 12, 19	2, 8, 9, 14, 18	3, 5, 11, 17, 20	4, 7, 13, 15, 16
b	3, 8, 13, 16	4, 5, 9, 12, 17	2, 6, 14, 15, 18	7, 10, 11, 19, 20
c	2, 4, 5, 8, 10	3, 6, 7, 17	9, 11, 12, 13, 14, 15	16, 18, 19, 20

1	--	scene object	11	--	bookcase
2	--	foreground	12	--	book
3	--	background	13	--	telephone
4	--	furniture	14	--	vase
5	--	office supply	15	--	trash can
6	--	wood object	16	--	picture
7	--	cement object	17	--	door
8	--	desk	18	--	wall
9	--	chair	19	--	ceiling
0	--	cabinet	20	--	floor

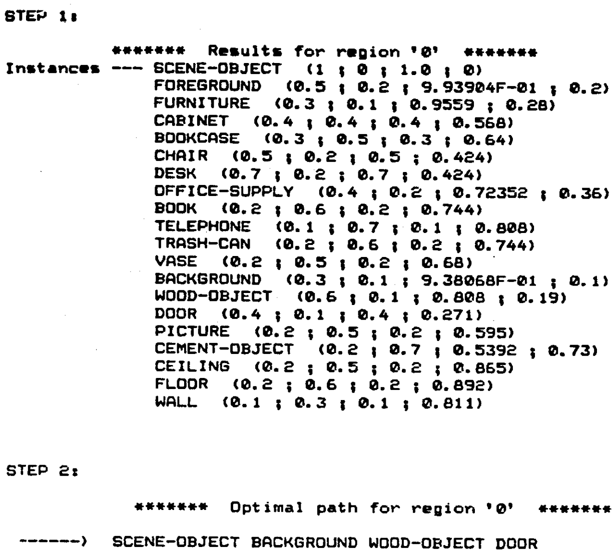
Table 4-1: Allocation scehemes of knowledge base frames to processors

```
defstruct image-region
  name          ; designation of this region
  mbr           ; minimum bounding rectangular
  bit-mask      ; binary mask
  hypothesis    ; symbolic description
  snap-pos      ; location of SNAP cell
  attributes    ; list of (attribute value) pa
  relation      ; relationship between regions
           (a) Region frame
```

```
defstruct model
  name          ; designation of object model
  location      ; address of MMS PE
  superclass    ; parent of object model
  subclasses    ; children of object model
  screener      ; hypothesis screener
  slot-frame    ; a list of routines to
                ; collect slot evidences
  established   ; routine to check establishme
  optimization; routines for slot optimizati
  action        ; final phase of control
                ; information

           (b) Model frame
```

```
defstruct instance
  name          ; designation of this instance
  location      ; address of SNAP cell
  parent
  children
  slot-frame    ; list of slot evidences
  confidence    ; confidence value of this
                ;instance

           (c) Instance frame
```

Figure 4-1: Lisp definition of three frame structures



```
STEP 1:

             *******  Results for region '0'  *******
Instances --- SCENE-OBJECT  (1 ; 0 ; 1.0 ; 0)
              FOREGROUND  (0.5 ; 0.2 ; 9.93904F-01 ; 0.2)
              FURNITURE  (0.3 ; 0.1 ; 0.9559 ; 0.28)
              CABINET  (0.4 ; 0.4 ; 0.4 ; 0.568)
              BOOKCASE  (0.3 ; 0.5 ; 0.3 ; 0.64)
              CHAIR  (0.5 ; 0.2 ; 0.5 ; 0.424)
              DESK  (0.7 ; 0.2 ; 0.7 ; 0.424)
              OFFICE-SUPPLY  (0.4 ; 0.2 ; 0.72352 ; 0.36)
              BOOK  (0.2 ; 0.6 ; 0.2 ; 0.744)
              TELEPHONE  (0.1 ; 0.7 ; 0.1 ; 0.808)
              TRASH-CAN  (0.2 ; 0.6 ; 0.2 ; 0.744)
              VASE  (0.2 ; 0.5 ; 0.2 ; 0.68)
              BACKGROUND  (0.3 ; 0.1 ; 9.38068F-01 ; 0.1)
              WOOD-OBJECT  (0.6 ; 0.1 ; 0.808 ; 0.19)
              DOOR  (0.4 ; 0.1 ; 0.4 ; 0.271)
              PICTURE  (0.2 ; 0.5 ; 0.2 ; 0.595)
              CEMENT-OBJECT  (0.2 ; 0.7 ; 0.5392 ; 0.73)
              CEILING  (0.2 ; 0.5 ; 0.2 ; 0.865)
              FLOOR  (0.2 ; 0.6 ; 0.2 ; 0.892)
              WALL  (0.1 ; 0.3 ; 0.1 ; 0.811)


STEP 2:

             *******  Optimal path for region '0'  *******

 ------>  SCENE-OBJECT BACKGROUND WOOD-OBJECT DOOR
```

Figure 4-2: Sample of simulator printout result

The parallel processing time and the speedup factor obtained are summarized in table 4-2. The parallel processing time includes the parallel computation time as well as the interprocessor communication time. Preliminary simulations for more than four processors indicate that the speedup factor increases linearly as the number of processors increases.

The processor utilization time is defined here as the ratio between the "busy time" over the parallel processing time. The utilization factors obtained for this example under the three allocation schemes considered are shown in table 4-3.

ALLOCATION SCHEME	a	b	c	sequential
PROCESSING TIME	615	588	723	1637
SPEEDUP FACTOR	2.66	2.78	2.26	1

Table 4-2: Parallel processing time for 4 processors using different allocation schemes

ALLOCATION SCHEME	UTILIZATION FACTOR			
	P1	P2	P3	P4
a	0.53	0.65	0.70	0.72
b	0.55	0.76	0.71	0.70
c	0.64	0.50	0.64	0.43

Table 4-3: Utilization factors

5. Conclusion

In this paper, a knowledge-based vision system was introduced and the emphasis was put on the parallel processing of object classification problem. The structure of the vision system proposed allows the extraction of some symbolic information from iconic domain at an early stage, and then provides the capability of performing complex symbolic processing in symbolic domain. Oftenly, because of the lack of a symbolic processor, in some IU systems many symbolic processing tasks are done in iconic domain. The proposed hierarchical architecture from section 3 is capable of performing a broad range of algorithms in image understanding. This system consists of an intelligent memory, a high level symbolic processor, a multiprocessor and a host computer. A possible mapping of a knowledge-based vision system into this architecture was proposed. We have also evaluated a portion of the architecture, the multiprocessor, using a software simulator. The results show the feasibility of parallel processing of a knowledge-based vision system. Work is in progress to simulate the operation of all the modules in the hierarchical architecture from figure 3-1 by implementing several algorithms from the vision system in figure 2-1.

References

[Ball 82] Ballard, D. H. and Brown, C. M.
Computer vision.
Prentice Hall, 1982.

[Binf 82] Binford, T.
Survey of Model-Based Image Analysis Systems.
The International Journal of Robotics Research 1(1):18-64, Spring, 1982.

[Brad 82] Brady, M.
Computational Approaches to Image Understanding.
Computing Surveys 14(1):3-71, March, 1982.

[Dani 81] Danielsson, P. and Levialdi, S.
Computer Architectures for Pictorial Information Systems.
IEEE Computer 14(11):53-67, November, 1981.

[Dixi 84] Dixit, V. and Moldovan, D. I.
Semantic Network Array Processor and Its Application to Image Understanding.
In *Proceedings of Image Understanding Workshop*, pages 65-71. DARPA, Octobor, 1984.

[Duff 81] Duff, M. J. B. and Levialdi, S.
Languages and Architectures for Image Processing.
Academic Press, London, 1981.

[Fost 76] Foster, C.
Content Addressable Parallel Processor.
Van Nostrand Reinhold, New York, 1976.

[Gold 85] Gold Hill Computers.
Golden Common Lisp
163 Harvard St., Cambridge, MA 02139, 1985.

[Mold 85a] Moldovan, D.I. et al.
Parallel Processing of Iconic to Symbolic Transformation of Images.
In *Proceedings of Computer Vision and Pattern Recognition.* IEEE Computer Society, June, 1985.

[Mold 85b] Moldovan, D.I. and Tung, Y-W.
SNAP: A VLSI Architecture for Artificial Intelligence Processing.
Journal of Parallel and Distributed Computing , May, 1985.

[Neva 82] Nevatia, R.
Machine Perception.
Prentice Hall, 1982.

[Rose 83] Rosenfeld, A.
Parallel Image Processing Using Cellular Arrays.
IEEE Computer 16(1):14-20, January, 1983.

[Shor 76] Shortliffe, E.
Computer-Based Medical Consultations: MYCIN.
American Elsevier, New York, 1976.

[Wu 85] Wu, C.I.
Integrated Hardware Structures for Knowledge-base Vision System
1985.
PhD Thesis Proposal, USC, April.

ACKNOWLEDGMENT

This research was supported by NSF Grant ECS - 8307258, JSEP Constract No. F49620-85-c0071 and DARPA Contract No. F-33615-82-k-1786.

A FAULT TOLERANT, BIT-PARALLEL, CELLULAR ARRAY PROCESSOR

Steven G. Morton

Staff Scientist, Central Research
ITT Advanced Technology Center
One Research Drive, Shelton, Connecticut 06484

ABSTRACT

We explore the question, "What are the effects of being able to build large, fault tolerant processor chips?" on a particular architecture, the SIMD, massively parallel, cellular array processor. The potential for high performance processor chips leads to a projected baseline configuration, a "personal supercomputer", that could be a coprocessor on a single plug-in board for a PC, that would provide peak performance of over 100 MFLOPS at 32-bits, or over 2,000 MIPS at 16-bits. The potential for low cost processor chips drives one to reduce costs throughout the system, including the extensive use of inexpensive DRAM, and lead to projections of price/performance with more than an order of magnitude improvement over alternative designs.

INTRODUCTION

The ITT Cellular Array Processor (CAP) architecture is characterized by its single-instruction stream, multiple-data stream (SIMD) design using bit-parallel, fault-tolerant, large area integration "array chips" for its parallel execution unit, and by its extensive use of low cost DRAM. In this paper we focus on the projected second generation, baseline design that would be a coprocessor occupying a single card slot in a PC. It would provide peak performance of over 100 MFLOPS at 32-bits or over 2,000 MIPS at 16-bits, as configured dynamically by software at runtime. A much lower performance, much larger, first generation prototype model has been demonstrated.

In this paper we emphasize the key issue of processor chip fault tolerance that we feel is the key to achieving the most compact, highest performance, lowest cost, most reliable, personal supercomputer for the image processing, scientific, and engineering marketplaces. The applications are characterized by matrix algebra, with either global operations, such as matrix multiply, for engineering and scientific processing, or many local operations, such as correlation windows, for image processing.

We particularly want to point out that we mean "personal supercomputer" in the sense of a significant fraction of a CRAY-1 or a Goodyear MPP on a desk top, as opposed to merely tens of MIPS or MFLOPS, as is getting to be the norm.

We have published several articles on various other aspects of CAP and its development, so we will only summarize that material in this paper. The instruction set, prototype model, and project evolution were covered in [1]. Details of the key prototype chip, array chip I, were covered in [2 - 5]. The logic simulation tools for chip development were covered in [6 - 7]. An application to image processing was given in [8].

This paper presents the projected second generation array chip and coprocessor design as viewed in early 1986. The technology continues to evolve in response to design and application experience. However, the underlying fault tolerant principles and their importance have not changed.

First, we compare well known cellular array processors to ours. Then we review our architecture, and explore the many effects of processor chip fault tolerance; fault tolerance that was a logical consequence of the tightly coupled, highly regular design, of a cellular array processor. Finally, we project where the next round of technology could take us.

COMPARATIVE CELLULAR ARRAY PROCESSORS

There are many parallel processors, but only a few cellular array processors. The Goodyear MPP (Massively Parallel Processor)[9], the ICL DAP (Distributed Array Processor)[10], and the Thinking Machines Connection Machine[11], are the most often mentioned machines in this class. The NCR GAPP (Geometric Arithmetic Parallel Processor) chip[12] (not a system) is unique in its diverse commercial application. Although the (ITT) CAP is similar to these machines in many ways, it also differs from them in these respects:

1. The CAP stores and processes the data stream differently. The other machines have 1-bit processor cells and are designed to operate in a bit-serial, word-parallel fashion. They store each word bit by bit through a succession of memory locations and increase word size in time. The second generation CAP, like the Illiac IV[13], has 16-bit processor cells and operates in a bit-parallel, word-parallel manner and increases word size in space, rather than in time. The first generation CAP, with 1-bit cells, is used as though it has 16-bit cells.

2. The CAP's parallel execution unit is fault tolerant at the level of a single cell. The MPP has limited fault tolerance, having four spare columns beyond is 128 minimum, but an entire column of 128 processors, not just one bad element, must be switched.

3. The CAP is a parallel reduced instruction set computer. Its assembly level programming model looks like a collection of conventional

CH2345-7/86/0000/0277$01.00 © 1986 IEEE

microprocessors working in lockstep. The MPP, DAP, and GAPP have single-bit oriented, even micro-, instruction sets.

4. The CAP interconnection architecture is an extension of other cellular array processors' nearest neighbor, except for the Connection Machine's hypercube. We use an x-y matrix of array chips II-M's, but since each array chip has 16 16-bit data processors, the x-dimension is nearest neighbor in 1 2-clock cycle, but the y-dimension requires 16 2-clock cycles to be nearest neighbor. This imbalance increases our packing density, reduces our pin count, and is coupled to our fault tolerant design, which enables defective elements to vanish from a one-dimensional array.

5. Due to both improved technology and fault-tolerant design, the second generation CAP would have 16 16-bit processors and large amounts (32+K bytes) of RAM on a single chip. The other machines integrate fewer processor bits with much smaller amounts of memory on a chip. This combination of large RAM and logic on a single chip increases the difficulty of chip design, both technically and administratively, but tremendously improves the price/performance and reduces the size of the system.

6. The CAP parallel execution unit has three levels of memory hierarchy, ranging from working registers to cache memory to external memory. The other machines have one level (GAPP and Connection Machine), or two levels (DAP and MPP), although the MPP also has a central, staging memory, as a global interconnect.

7. The CAP parallel execution unit has distributed memory addressing. The cache and external memory of each array chip have their address generated by the array chip. The other machines have only global addressing due to their single-bit architecture, making them prone to single-point failures in the address logic.

8. For a given total number of processor bits, the CAP is better suited to engineering applications than are the other machines, since the number of processors it provides is a better match to the dimensionality of many problems. Where the DAP would handle 4096 1-bit variables in bit-serial fashion, requiring enormous matrices for efficient use of the machine, a CAP with the same total number of processor bits would handle 128 32-bit variables in bit-parallel fashion. This provides efficient operation on much smaller matrices, as well as providing good processor utilization on more sizes of large matrices.

9. The CAP would have integrated, concurrent I/O so that data acquisition and computation may occur simultaneously. Multiple buffers would be provided in each processor cell so that interrupt service routines can provide direct memory access to the cache or external memory.

10. The MPP and Connection Machine processor cores occupy a volume over 1 m^3. The CAP would occupy less that 1% of that.

11. The primary application areas of the MPP, DAP, GAPP, and CAP are image processing, although the CAP targets engineering and scientific processing as well. The Connection Machine targets artificial intelligence - image understanding.

CAP ARCHITECTURE

OVERVIEW

The CAP employs a highly regular design based on an SIMD, cellular array architecture. A first generation prototype has been built from large (the largest logic chips in the U. S.), custom designed, CMOS chips that are feasible due to their fault tolerant design. A second generation design has been defined. The dominant chip, the

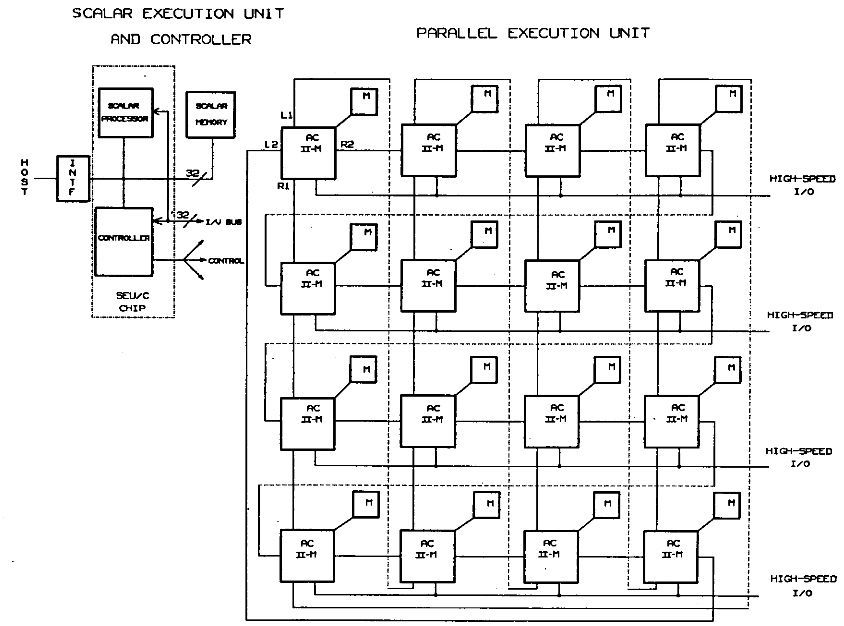


FIGURE 1 - PERSONAL SUPERCOMPUTER BLOCK DIAGRAM

array chip, has a simple, regular design that allows all processor elements, including a few spares, to be configured by software. In this way, words of varying size may be formed by cooperating processors, manufacturing defects may be overcome, and lifetime failures may be corrected.

A block diagram of the baseline, second generation personal supercomputer is shown in Figure 1. The scalar memory and scalar processor form the scalar execution unit (SEU). Scalar variables and all instructions are stored in the 32-bit wide scalar memory. A 32-bit wide scalar processor handles global address calculations, loop counts, and other scalar variables.

The parallel execution unit (PEU) contains a 4 x 4 matrix of array chip II-M's to provide 256 16-bit data processors, 128 32-bit processors, or 64 64-bit processors as configured by software at runtime. The array chips may be interconnected in many ways, but we will assume that they form 4 rows and 4 columns, with spiral connections between rows and between columns to form a modified nearest neighbor connection. The array may also be viewed one dimensionally as a single row.

Each array chip II-M in the PEU has 16 active 16-bit processors for data, 2 active 16-bit processors for address, 2 spare processors, plus on-chip RAM. The assignment of a processor, or cell, for address computation or data computation is made by configuration software at runtime. A 16-bit word is formed from a single cell. Thirty-two bit words are formed from a pair of adjacent good cells, possibly with a defective cell in the middle. Likewise, 64-bit words are formed from four good cells with intervening defective cells.

The on-chip RAM, nominally 1KW per processor, is a program - controlled cache; it is addressed by the 10 lsbs of the pair of address processors. The external DRAM, nominally 512KB per array chip, is for program-controllled page roll-in and roll-out of the cache, and is controlled by the pair of address processors, and may be viewed as a local staging memory.

The controller (C) provides instruction fetch and decode and drives the SEU and the PEU, providing a reduced instruction set for scalar and parallel processing. Operation of the SEU and PEU are mutually exclusive in order to simplify the design. Few applications seem to benefit significantly from their concurrentcy, and providing it is difficult.

We constructed a mechanical mockup of the CAP personal supercomputer to help us visualize the effect of surface mount technology. See Figure 2. The photograph was taken using a mirror so that components on both sides of the board may be seen. The two rows of eight large chips on the upper surface represent array chip II-M's. A combination scalar execution unit and controller (SEU/C) chip would also be on the upper surface. Four DRAM's external to each array chip would be on the lower surface, immediately behind the array chip.

The near-term technology projections for this second generation configuration design are shown in Table 1. A comparison of the first and second generation designs is given in Table 2.

Near-Term Technology Projections
for a Fault Tolerant CAP

Table 1

Technology Requirements
- 1.25 um double metal CMOS
- 256K DRAM VLSI layout
- 1M DRAM chips
- 5x wafer stepper
- Double sided surface mount

Architecture
- Highly Parallel Cellular Array
- SIMD - Parallel RISC
- 12.5 MHz base clock rate
- 25 MHz repeat step clock rate
- Modified nearest-neighbor interconnections

Scalar Execution Unit (SEU)
- Performance (Peak)
  - 12.5 MIPS at 32 bits
  - 6 MFLOPS at 32 bits
- Operation of PEU and SEU are mutually exclusive

Parallel Execution Unit (PEU)
- Performance (Peak)
  - 1/8 - 1/2 Cray Research Cray-1
    - 20+ MFLOPS at 64 bits
    - 100+ MFLOPS at 32 bits
  - 1/3 Goodyear MPP
    - 1,000+ MIPS at 32 bits
    - 2,000+ MIPS at 16 bits
  - 100 MB/S concurrent external I/O bandwidth
- Dynamic Configurations
  - 256 16-bit data processors
  - 128 32-bit data processors
  - 64 64-bit data processors
- Memory
  - 8KB distributed working registers
  - 512KB distributed cache memory
  - 8MB distributed external memory

Physical Size and Power
- Single board - 1" x 5" x 14"
- 50 watts

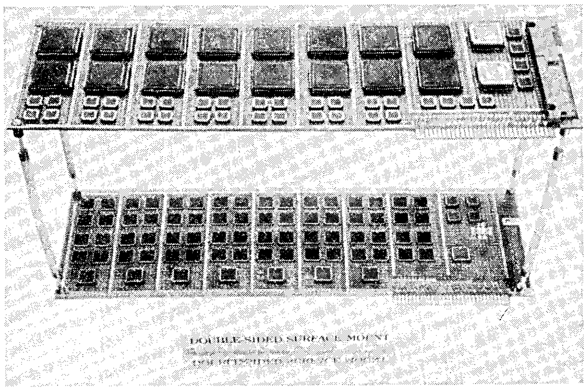


FIGURE 2

MECHANICAL MOCKUP OF PERSONAL SUPERCOMPUTER

Comparison of First and Second Generation CAPs

Table 2

	First	Second
1. Status	Demo 1985	paper design
2. Word Sizes Supported		
SEU	32-bits	32-bits
PEU	16-bits	16-, 32-, 64-bits
3. SEU/C Construction	TTL MSI	custom chip (proposed)
4. PEU Construction		
Data Processors	16 x 16-bits	256 x 16-bits
Address Processors	16 x 16-bits	16 x 32-bits
5. Peak PEU Performance		
Fixed point	24 MIPS/16-bits	2,000 MIPS/16-bits
Floating point	none	100 MFLOPS/32-bits
6. Array Chip		
Type	I	II-M
Status	operational	paper design
Processors	20 x 1-bit	20 x 16-bits
Spare processors	4	2
Registers per processor	16	16
RAM per processor	none	1KW (nominal)
7. PEU Configuration	1 column, 16 rows	64 columns, 4 rows
8. Instruction Set	Fixed point Parallel RISC	Fixed and floating point Parallel RISC
9. Clock Rate	1.5 MHz	12.5/25 MHz
10.Size	5 14" x 14" boards	1 5" x 14" board
11.Power Dissipation	250 watts	50 watts (est)

SYSTEM PERFORMANCE ESTIMATES

We calculate our architecture's maximum performance by assuming that all active parallel data processors are busy all of the time doing the quickest operation. Since this condition cannot occur indefinitely, due to the need for scalar operations, which are mutually exclusive with parallel operations, as well as iterative fixed-point operations and data-dependent masking, we derate the maximum to arrive at the peak values reported herein. We use a greater derating factor for fixed point than for floating point because the latter are inherently iterative but many fixed point are not.

For fixed-point operations:
256 16-bit processors x 12.5 MHz = 3,200 MIPS
Derating by 1/3 gives 2,000 MIPS
128 32-bit processors x 12.5 MHz = 1,600 MIPS
Derating by 1/3 gives 1,000 MIPS

For floating-point operations:
128 32-bit processors @ 1 MFLOP = 128 MFLOPS
Derating by 1/6 gives 100 MFLOPS
64 64-bit processors @ .375 MFLOP = 24 MFLOPS
Derating by 1/6 gives 20 MFLOPS

Performance estimates for 16-bit operations on a 256 by 256 data base are:

256 concurrent, 256-point complex FFTs - 7.5mS (rate of 1 per 30uS)
matrix transpose - 10mS
2-D FFT (real input) - 22mS
5 x 5 correlation - 2.5mS

ARRAY CHIP II-M

The basic idea for this chip is that one could take a DRAM, keep its common row decoder, strip off its column decoders, and directly couple a 256-bit wide bus to 16 16-bit processors, all on the same chip. Our fault tolerant design enables us to provide spare 16-bit processors and to widen the DRAM correspondingly in order to provide excellent chip yield, and thus keep the cost low. We would thus couple two fault tolerant designs, memory with spare rows and columns, and logic with spare processors.

Note, however, that there is common addressing for all of the memory on a chip. The address decoder is much too large to provide for each processor, plus it would be hard to have variable word sizes with matching memory addressing. Different chips may, however, may have different addresses. This is a compromise between the Goodyear MPP and the ICL DAP, where there is a single address for all of memory, since the memories are only one-bit wide, and loosely coupled microprocessors, where each processor can freely address its own memory. However, we do provide, at the expense of lowered efficiency, that processors on one chip may sequentially, independently, address the memory, as is valuable for matrix transpose operations.

Figure 3 shows the array chip II-M block diagram. We assume at least 256K bits of usable (320K bits total) on-chip DRAM are provided. The chip's key elements are the 20 (18 plus 2 spares) 16-bit processors, or logic cells, each of which is coupled to a 1K-word (or more) memory cell. All of the memory cells are addressed in parallel by the row decoder plus distributed group (1-of-4) decoders. The DRAM is used as a data cache and is program controlled.

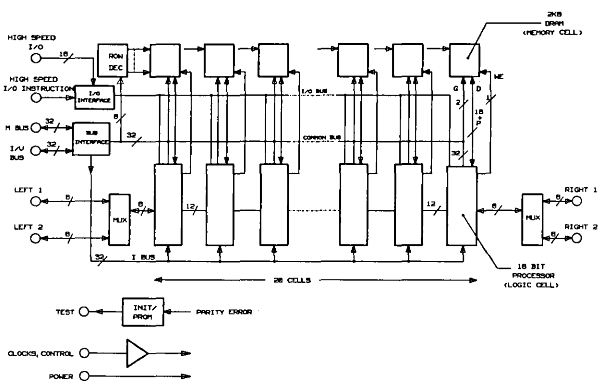


FIGURE 3

ARRAY CHIP II-M BLOCK DIAGRAM

We estimate that the array chip II-M/256K would contain 600,000 transistors, 2/3 of them in DRAM, and would have a die size of 500 x 600 mils in 1.25um double-metal CMOS. Later versions presumably would have a larger DRAM and thus substantially more (1,000,000+) transistors. While the typical process for a 256K DRAM is 1.5um, double poly, single-metal, we would use 1.25um and double-metal for the benefit of the logic cells.

The target clock rate is 12.5 MHz for a 32-bit, register-to-register ADD over 32 bits (two good cells with one bad in between). Iterative operations such as 2-bit multiply would use a 25 MHz clock. Note that we do word-serial arithmatic for multiply, divide and floating point operations, as explained later.

All memory cells simultaneously provide a word to their respective processor for a net transfer of 256 bits. In a conventional DRAM, many bits are fetched but only a few are output. This is terribly inefficient; the design of Video DRAMs recognizes this inefficiency. This combination of memory and logic on a single chip provides an enormous reduction in the number of chips and pins required to implement a system.

Manufacturing data, particularly the location of defects, would be stored in the INIT/PROM block at the time of factory testing. This data would be read by the system controller from each array chip II-M in turn when the system is turned on. Since the PEU is "defective", presumably having 1 or 2 bad cells per array chip, it must be "repaired"; the array chips do not repair themselves. Each array chip may be selected individually by an external decoder driving its chip select pin, and each cell may be selected by its physical address (0-19) in an array chip. Each cell in each array chip is turned off, to become invisible, or is set into the desired word configuration. The application may then refer to cells by their virtual address, 0-15, for data movement, independent of the physical locations of the defects. The configuration may also be changed as needed, such as following a diagnostic program that locates newly failed cells.

LOGIC CELL

The basic structure is a 16-bit slice, where in one cycle two operands are read from the multiport RAM (MPR), operated upon by the ALU and shift logic, and returned to the multiport RAM. The microinstruction set of array chip II-M (a "vector bit-slice" chip) resembles that of a collection of 64 AMD 2903 4-bit slice chips, plus memories and address logic. Register - to - register boolean operations are done in 80nS. Iterative multiply, divide, and floating point operations are done in a 40ns cycle. The logic cell block diagram is shown in Figure 4.

The path logic contains the mechanism for hiding defective cells, for connecting cells within a chip to form words longer than 16 bits, and for performing functions such as shift and rotate. A defective cell is bypassed, with data flowing over it as though it were not there. Extensive processor enable, or mask logic, controlling storage in both a logic and its memory cell, is also provided.

Words are configured by software at runtime by initializing the "configuration masks" in each logic cell. These masks are stored in the processor status word, along with the virtual cell ID, and control the flow of data within and between cells. The basic states of these masks are:

0. Inactive
1. 16-bit data
2. lsbs of 32-bit data
3. msbs of 32-bit data
4. lsbs of 32-bit address
5. msbs of 32-bit address

We have additional states to handle the assignment of a 16-bit cell to the 32-bit common bus, and to control the movement of data among processors.

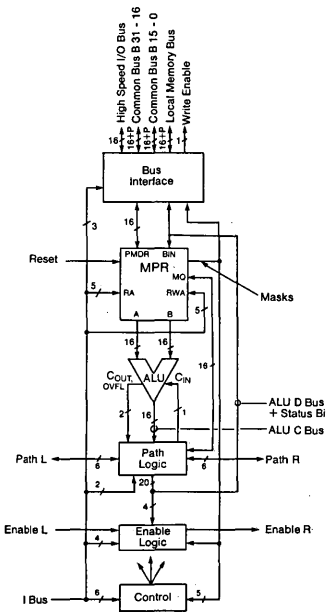


FIGURE 4

LOGIC CELL BLOCK DIAGRAM

MEMORY CELL

One memory cell, containing 2K bytes of storage, is provided for each logic cell. No provision is made for interconnection between one memory cell and other logic cells due to the complexity of interconnect that would arise. The memory cells share a common row decoder, as is found in ordinary DRAM designs. The two-bit group select from each logic cell selects one-of-four 17-bit groups (16-bits of data + 1-bit of parity) within a 256-row by 68-column memory cell. Spare rows, common to all cells, and a spare column per cell, improve the manufacturing yield. See Figure 5.

Although we assume the use of DRAM in this paper, the choice of DRAM or SRAM is based upon the availability of RAM technology to the design team, and the level of concern over soft errors. The general thought is that since no amount of memory is ever enough, the 4x (or higher for trench capacitors) density of DRAM over SRAM makes DRAM preferable.

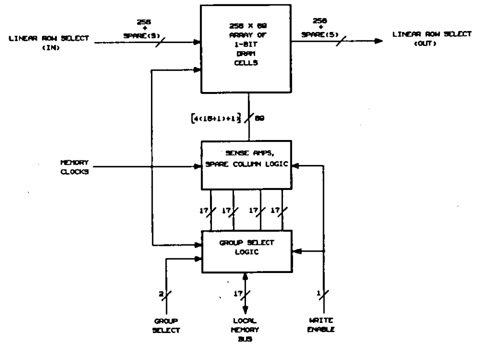


FIGURE 5

MEMORY CELL BLOCK DIAGRAM

PROGRAMMING MODEL

Unlike many array processors tnat are programmed only by microcode and are used to implement a library of Fortran-callable subroutines by a host, the CAP would run entire applications, written in assembly or high level language, often on live data. This would be possible because the CAP uses a simple, highly efficient, register-based, load/store architecture with deferred loads and jumps. The design of the instruction set is strongly influenced by the RISC[14] - reduced instruction set computer - philosophy. Our set contains only 33 basic instructions, including the load, store, arithmetic, logic, shift and program control. Instructions for configuration, scientific computation and SIMD support, such as interprocessor communication, are also provided.

From the system point of view, the programmer will begin a task on the host. The host would spawn a task that invokes a task on the CAP. The task on the CAP may be arbitrarily complex, accessing dedicated peripherals or communicating with the host operating system for file and user access.

Figure 6 shows a simplified programming model for the CAP. Each processor has sixteen registers, and each PEU processor has a local memory. All data processors have processor status words, and the PEU data processors have enabling masks (we call them vector if...else stacks - VIES). Notice the strong similarity to the programming models for ordinary, SISD, machines. This is in sharp contrast to today's supercomputers and array processors that have radically different programming models compared to SISD machines.

One easy way to visualize the CAP is as a collection of conventional SISD processors, each operating on its own data. The processors operate in lockstep and pass data to the left, right, up and down, to communicate in an organized fashion. Multiple concurrent FFTs or even recursive filters can be readily implemented. Fewer FFTs, each involving multiple processors, may also be done but with less efficiency. Implementing a perfect shuffle network on a fault tolerant grid is a problem that we have not solved.

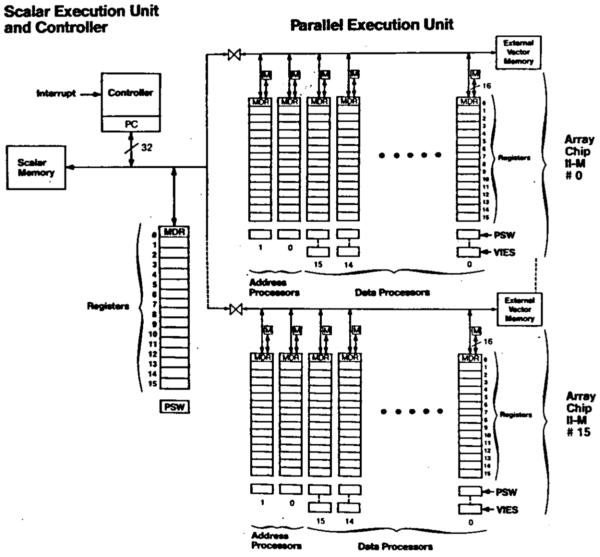


FIGURE 6

SIMPLIFIED PROGRAMMING MODEL

FAULT TOLERANCE IN THE CAP DESIGN

We recognized that the highly regular architecture of the parallel portion of a SIMD cellular array processor provides a design that is well suited to a fault tolerant design. (This realization occurred to us when our original chip design became too big to build any other way.) Just as DRAM's have employed spare elements for years, selected at the time of manufacture by a laser, our array chips provide a few spare elements selected at runtime by software. These spare elements may compensate not only for chip birth defects (from time of manufacture), but lifetime defects (wear out) as well, and are tied to our variable word size design.

Unfortunately, we have not found an efficient solution to the problem of fault tolerance in the common logic. Triple modular redundancy is too expensive for our tastes. Any failure in the scalar execution unit, controller, the power and clock distribution system, and the interconnections between chips will cause the failure of the system. We considered using error correcting codes on the microinstruction bus and data paths, but felt it was too much trouble. Furthermore, we have not found an efficient solution to detecting faults in the parallel execution unit other than running diagnostics. What we have done, however, is to use fault tolerance to reduce the cost of the largest portion of the system.

Having a few spare elements means we can increase the number of elements on a chip and maintain excellent yields. We thus need fewer chips, with fewer total interconnections to build a system. Mechanical connections, as from chip die to package to board, appear to be the least reliable portion of a system, so any attempt to reduce the number of connections should greatly improve reliability. The use of fault tolerance thus makes economic sense - it reduces the size and cost of our system.

ARRAY CHIP I

Array Chip I tested our fault tolerant technique. This device is our first-generation processor chip and is the basis for the prototype system we have built. See the chip photo in Figure 7. It uses 3um double metal CMOS technology and contains twenty 1-bit cells (in four groups of five cells for power distribution), of which only 16 are required for proper system operation, plus common logic and I/O buffers. Its block diagram is similar to the one for array chip II-M but without the DRAM, and its logic cell is similar to the one in array chip II-M, except the paths are 1-bit wide rather than 16. The locations of the bad cells are invisible to both the programmer, via cell bypass logic, and to the I/O pins, via an on-board, software - controlled, cellular switching matrix.

Array chip I has 120,000 transistors, 144 pins (many unused in our prototype system), a power consumption of about 500 mW, and a die size of 650 x 500 mils, the largest logic chip in the U.S. Any combination of word sizes, down to a single bit and upward without limit, extending across chip boundaries, may be selected by software with no external logic.

Its target clock rate for a 16-bit register-to-register ADD over 20 cells was 10 MHz, although its actual clock rate is closer to 3 MHz, due in part to miscalculated capacitive loading on clock buffers. Critical signal paths that use many pass transistors and that cross the entire chip make performance estimation particularly difficult. Our second generation, array chip II-M design, with closely contained critical paths, removes this latter difficulty.

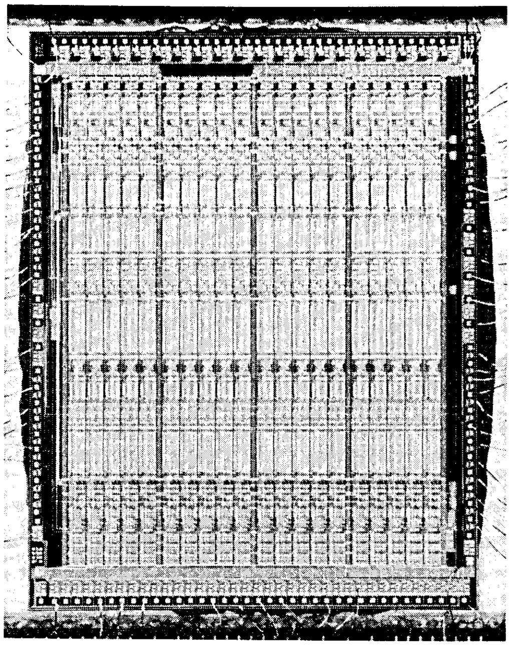


FIGURE 7

ARRAY CHIP I PHOTOGRAPH

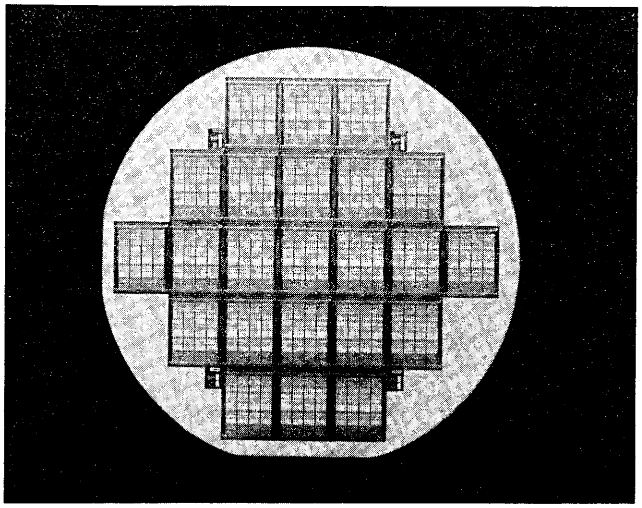


FIGURE 8

WAFER OF ARRAY CHIP I's

Array chip I may be viewed as 21 small chips rather than one large chip from a manufacturing yield point of view. Each of the 20 cells, plus one block of common logic, provide the 21 regions. Each cell area for yield analysis purposes is reduced by the small (10%) amount of critical logic area that must work, and the total amount of this critical logic area is added to the area of the non-fault tolerant common logic, which also must work. We estimated chip yields (at least 16 of 20 cells good) in the 25%-30% range, assuming 2 defects/cm^2, and we were happy to have this proven true. VLSI Technology Inc. fabricated our chips. The yield is the sum of all of the yields for the many combinations of 20 cells taken 16 at a time, times the yield of the common logic area.

A photograph of the array chip I wafer is shown in Figure 8, showing the large size of the dice. The 4" wafer contains only 23 array chip I dice (plus 4 small test dice), compared to a hundred or more for most designs.

Our first generation system prototype uses 36 array chip I's and has been in operation since late 1985. Four array chip I's form the scalar execution unit, providing 32-bit words for each of the address and data processors. Thirty-two array chips form the parallel execution unit, providing 16 16-bit processors, each with address and data units. A photograph of one of the wirewrap boards is shown in Figure 9. Twelve array chips I's, having large (and rotated) ceramic lids, form six of the parallel processors.

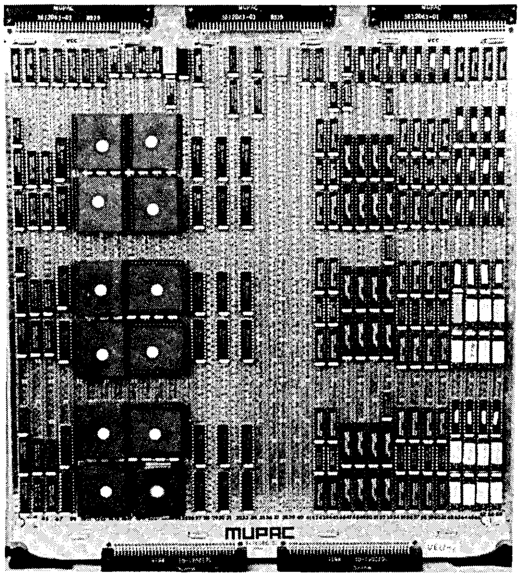


FIGURE 9

PROTOTYPE BOARD PHOTOGRAPH

EFFICIENCY OF DESIGN

We chose to use repetitive word-serial arithmetic for multiply, divide, and floating point operations. By word-serial we mean, for example, eight 2-bit ADD and SHIFT operations for a 16-bit by 16-bit multiply. This is in contrast to the parallel, often single-cycle, multipliers and floating point units that are common in many array processors. We did this for several reasons.

First, the amount of logic required and the amount of chip area required is much less for repetitive rather than parallel techniques. The yield of a processor element is inversely related exponentially to its size, so we want to keep the size small.

Second, we can support various word sizes. An add and shift structure grows linearly with the length of the word, requiring more cycles, whereas a parallel structure grows with the square of the word size. However, our throughput falls with the square of the word size for repetitive operations, since 2 cells are required for 32-bit operations, and twice as many cycles are required. Worse, we must lower the clock rate when switching from 32-bit words to 64-bit words. Our 16-bit timing is set at 32-bit cycle times because of instruction fetch limitations in the controller.

Third, the efficiency of a repetitive structure is higher than for a parallel structure. That is, there is a higher fraction of gates active for a larger portion of the time in a well-used repetitive structure than in a parallel structure. Consider a ripple-carry adder. Only one bit at a time is doing anything in the worst case carry propagation, as the carry flows from one bit to the next. There is no reasonable way to vary the clock rate depending upon when the summation is finished. Most of the adder is thus idle most of the time.

A parallel multiplier is similar. There is a wave of activity as the product formation flows through, but again, most of the gates are idle at any instant.

Thus, if one has a problem that can benefit from parallelism, the objective becomes to find a design that keeps most of the elements active and to increase the clock rate to get the most computation per unit area. This puts the speed burden on a small number of local registers, rather than on a larger, off chip memory.

This is an argument for memory hierarchy; the closer storage is to the arithmetic element, the smaller it may be and the faster it should operate. In array chip II-M we would have three levels: 16 registers, 1KW of cache, and 16KW of shared RAM, per parallel processor.

As a result of the increased number of processors that we may have per unit area of silicon because our structure is more efficient, we may keep more temporary variables in registers. This is a variation of the RISC philosophy. Thus instead of reading and writing vectors from and to RAM for each of a series of vector operations, we can be more like a systolic array, minimizing the number of memory transfers, by keeping temporaries in registers.

This enables us to use slower, denser, less expensive off-chip DRAMs rather than SRAMs. Further, the price and size disparity between SRAMs and DRAMs should be heightened as DRAMs become three - dimensional, using trench capacitors, for which there is no analogy in SRAMs.

NUMBER OF PROCESSORS PER CHIP

For a regular design that can use our cell bypass technique, one can increase the processing power per chip by increasing the number of cells so long as the I/O requirements to sustain those cells can be fulfilled. The memory on board array chip II-M, coupled to an I/O channel, is intended to help keep the cells busy.

Since the yield decreases as chip size increases for a non-fault tolerant chip, one may include some additional elements beyond the number required for a particular throughput. This further increases chip size, reducing the number of dice per wafer, but we found a broad range of choices. Better than a factor of ten improvement in yield may be observed. In addition, if one can use grade-outs, say chips with half of the cells good, one may opt for fewer spares. Thus we went from four spares to two in going from array chip I to array chip II-M, although decreased defect densities over the last few years increase our conviction for doing so.

A limit on chip size is set by the capability of the photolithographic equipment that exposes the wafers. In 1984, for 3um features as used by array chip I, one had 10x or 1x reticles; that is, the mask is ten times or one time the size of the die. A 10x wafer stepper can only build up to a 400 x 400 mil chip. A 1x stepper can build a chip the size of the wafer. For 1.25 um features, as would be used by array chip II-M, one generally uses a 5x stepper, limiting the size to 800 x 800 mils. Building a single device bigger than that size would require tiling the device with multiple sub-chips that are minimally interconnected due to alignment problems between them.

We might have built array chip I with few enough processors to achieve a chip size that was considered an upper limit, 400 x 400 mils. Instead, driven by an interest in wafer scale integration, we addressed the issue of what constitutes a limit on size. We were fortunate in being able to translate our interest in a highly regular, cellular array structure with bit-parallel, variable size words, into bypassable, fault tolerant cells.

NUMBER OF BITS PER PROCESSOR

The MPP, DAP, GAPP, and Connection Machine all have one bit per processor. Our array chip II-M would have 16 bits, and we use array chip I as though there is a single 16-bit processor.

We found how inefficient it is to build wide words in space from 1-bit cells. Holding fabrication technology constant, one can build a 16-bit cell in about twice the area of our 1-bit cell. The inefficiency in our one-bit cell results from the very high overhead that we paid to string one-bit cells side-by-side to build words of arbitrary size.

If, however, one is content to build long words in time, by using multiple cycles rather than multiple cells, which is better? The answer seems to lie in the amount of parallelism that one can use effectively, and the degree of freedom that one would like to have in addressing memory.

One reason that we went to 16-bit words was so we could afford some flexibility in addressing memory. We can accept having all parallel processors reference the same register, but having all the processors reference the same cache or bulk memory location severely limits the efficiency of many operations, such as matrix transpose, which needs a diagonal of addresses. No processor using 1-bit elements can afford to independently address each bit of memory. We can not afford in array chip II-M to have all processors independently address each 16-bit word, but at least each array chip may form a different address. We can then do time-division multiplexing between processors within an array chip, with 1/16 efficiency, to independently address the cache or external DRAM.

We also considered 8-bit and 32-bit cells. We had no applications for 8-bits, and the yield improvement of the smaller cell was insignificant. We liked 32-bits for floating point, but would have lost half our capability at 16-bits which we wanted for image processing, and the cell would have been bigger than we liked. So we chose 16 bits, with the provision that a pair of cells may work together to form 32-bit words, and the provision that a defective cell may be between them.

ONE-DIMENSIONAL FAULT TOLERANCE

The key to fault tolerance in array chip I and array chip II-M is that bad cells are invisible. Data flows across them from the left to right and vice-versa as though the cell were not there. Array chip I has five so-called horizontal paths, and array chip II-M would have twelve. These paths are local, from the left of cell N to the right of cell N+1, and are easily implemented. The implementation of a 2-dimensional bypass network, however, is extremely cumbersome as we found on the I/O pin reconfiguration logic on array chip I which connects good cells in one chip to good cells in another chip regardless of the locations of defects.

DIRECTIONS FOR FUTURE WORK

One of the most difficult problems is how to interconnect processors. We are currently limited to two-dimensional structures on the surface of a chip or board. The application of optical communication through stacks of wafers, unknown today, would give us a third dimension and should pave the way for far more powerful and more compact structures. Enormously parallel processors with capabilities in the near teraFLOP region and with multi-gigabit I/O rates, dissipating only 10KW, and contained in a liquid-cooled, six inch diameter cylinder six inches long are conceivable[15].

CONCLUSION

We have demonstrated a first generation, and defined a second generation, cellular array processor using large area integration chips for the parallel execution unit - the bulk of the system. These chips are feasible only because of their fault tolerant design. We believe that the high performance, low cost, and small size that result from the use of these or like chips will dramatically improve the price, performance, and reliability of cellular array processors.

ACKNOWLEDGEMENTS

Many people contributed to the project from its inception in 1981 until its prototype

demonstration in 1985. Figure 10 shows the team that was closely involved at the time of the demo. Particular recognition goes to the project founder, Jack Cotton (not shown), and the division director, Santanu Das (not shown) for their belief in the project.

FIGURE 10

CAP TEAM PHOTOGRAPH

Back row, from left: Steve Demianczyk, Dave Jenkins, Craig Barrila, Dave Evans (project manager), Ravi Masand, Steven Kulick, Enrique Abreu; middle row: Fred Tse, Sarma Jayanthi, Nick Carter, Mark Hervin; on floor: Steven Morton (author).

BIBLIOGRAPHY

[1] S. Morton, E. Abreu, and F. Tse, "ITT CAP - Toward the Personal Supercomputer", IEEE Micro, December 1985, pp. 37-49.
[2] S. Morton and E. Abreu, "A Dynamically Reconfigurable Array Chip", IEEE Journal of Solid State Circuits, (in preparation).
[3] S. Morton, U. S. Patent #4,536,855, "Impedance Restoration for Fast Carry Propagation", Aug 20, 1985.
[4] S. Morton, U. S. Patent #4,546,428, "Associative Array Processor With Transversal Horizontal Multiplexer", Oct 8, 1985.
[5] S. Morton, U. S. Patent #4,580,215, "Associative Processor with Five Arithmetic Paths", Apr 1, 1986.
[6] S. Morton, "Use of Zycad Logic Evaluator", Electronics Business, Oct 15, 1985, pp 76-77.
[7] D. Jenkins and S. Morton, "Transistor Level Logic Simulation on the Zycad Logic Evaluator", Proceedings of the ADEE-East Conference, Oct 1985.
[8] S. Morton and S. Jayanthi, "Image Compression on a Parallel RISC Machine", Mini-Micro Northeast Professional Program, May 1986, pp 15/1 - 15/11.
[9] K. E. Batcher, "Design of a Massively Parallel Processor", IEEE Transactions on Computers, 1980, pp 836-845.
[10] J. Modi, "DAP-Fortran and Programming Techniques on the Distributed Array Processor (DAP)", Cambridge Univ. Eng Dept Tech Rep CWED/F-CAMS TR 250, Feb 1985.
[11] W. D. Hillis, The Connection Machine, MIT Press, 1985.
[12] R. Davis and D. Thomas, "Systolic Array Chip Matches the Pace of High Speed Processing", Electronic Design, Oct 31, 1984, pp 207-214.
[13] Barnes et al, "The ILLIAC-IV Computer", IEEE Transactions on Computers, Aug 1968, pp 746-757.
[14] D. A. Patterson, "Reduced Instruction Set Computers", Communications of the ACM, Vol. 28, No. 1, Jan 1985, pp 8-21
[15] S. Morton, "A Fault Tolerant, Enormously Parallel Processor Architecture Using Optical Interconnections for near TeraFLOP Performance", in preparation.

BIOGRAPHY

Steven G. Morton was born in Yokosuka, Japan, in 1949. He is a member of the IEEE and is married, has two children, and lives in Oxford, Connecticut. He is the architect of the CAP on which he has filed for 15 patents. He joined ITT in 1979, and his first responsibility was the construction of a digital signal processor for the ITT System 12 Digital Switch. Prior to ITT, he was with the MIT Lincoln Laboratory where he worked on attitude control and ground - based telemetry systems for the LES-8 and LES-9 Air Force communication satellites. He received his BSEE and MSEE from MIT in 1972, where he specialized in computer architecture. He is now in Central Research at ITT studying the limits of optoelectronics.

IMPLEMENTATION OF PARALLEL PROLOG ON TREE MACHINES

Hajime MIURA* Masaharu IMAI**
Masafumi YAMASHITA*** Toshihide IBARAKI****

* Fujitsu Limited, Kawasaki, Kanagawa, 211 Japan
** Toyohashi Univ. of Technology, Dept. of Information
and Computer Sciences, Toyohashi, 440 Japan
*** Hiroshima University, Dept. of Electrical Engineering,
Higashi Hiroshima, 724 Japan
**** Kyoto University, Dept. of Applied Mathematics and Physics,
Kyoto, 606 Japan

Abstract

In this paper, parallel algorithms to control the execution of Prolog programs on tree machines are proposed, and their efficiencies are compared through simulation experiments. The model of the tree machine used in this experiment consists of a single-tree engine, a multiplexer, and a system controller. The single-tree engine is a common part of tree machines and has generality. From the experimental results through simulation, it appears that if the given problem has enough parallelism, then high performance can be obtained by the algorithms proposed in this paper. While a better utilization of processing elements and a better load balancing are future research problems, tree machines are found to be suitable for parallel implementation of Prolog.

Key Terms: Prolog, knowledge-based problems, tree machines, logic programming, multiprocessor systems, parallel algorithms.

I. Introduction

Logic programming is one of the best methods to formalize and solve knowledge-based problems [Nils80]. Prolog is a logic programming language based on first order logic [Kowa79], where knowledge is represented by a set of facts and a set of inference rules. The ICOT (Institute for New Generation Computer Technology) started its research project for the Fifth Generation Computing Systems (FGCS) in 1982 under the auspices of MITI (Ministry of International Trade and Industry) of Japan [Moto82], [Feig83]. Prolog is supposed to be the kernel language of the new computer systems in its project.

One of the main difficulties in solving knowledge-based problems is that many of them are NP-hard, that is, the computation time needed to solve these problems on a conventional sequential computer will increase exponentially as the problem size increases. One of the best solutions to overcome this difficulty is to implement special purpose multiprocessor systems for logic programming. If the problem has enough parallelism, the computation time to solve these problems on such systems will be effectively reduced as the number of processing elements (PE's) is increased.

Multiprocessor systems can be classified in two categories. The first includes **tightly coupled systems** where PE's are connected through shared common memories; and the second includes **loosely coupled systems**, where PE's are connected through message links.

One of the advantages of tightly coupled systems is that the communication between PE's can be quickly executed if the number of PE's is small. However, if the number of PE's is large, the performance of communication among PE's tends to be reduced due to memory access contention. Another disadvantage of tightly coupled systems is that their system software tends to be harder to design and implement compared to the loosely coupled systems. As a consequence, the construction of efficient tightly coupled systems that have a large number of PE's is more difficult.

Loosely coupled systems are designed to avoid these problems. Though the message transfer rate between PE's of these systems is slower than that of tightly coupled systems with a small number of PE's, loosely coupled systems have some advantages. One is that access contention for message transfer can be reduced; and another is that system software is relatively easy to design and implement due to its modularity. Therefore, it is easier to construct loosely coupled systems that have a large number of PE's.

The choice of interconnection topology is one of the critical problems in constructing efficient loosely coupled systems. In regard to the interconnection topologies for loosely coupled systems, various possibilities have been proposed. They include: ring, mesh, hypercube, and tree structures. Interconnection topologies based on tree structure seem to be efficient for solving knowledge-based problems because the computation process for solving these problems can be naturally represented in the form of a tree.

Another important key to constructing efficient loosely coupled systems is the design of parallel algorithms that fit the interconnection topology of the system. In this paper, several parallel algorithms to control the execution of Prolog programs on tree machines are proposed, and their efficiencies are compared through simulation experiments.

The model of the tree machine used in this experiment consists of a single-tree engine, a multiplexer, and a system controller. The single-tree engine is a common part of tree machines and has generality. From the experimental results through simulation, it appears that if the given problem has enough parallelism, then high performance can be obtained by the algorithms proposed in this paper. While a better utilization of Processing Elements and a better load balancing are future research problems, tree machines are found to be suitable for parallel implementation of Prolog.

II. Parallel Processing Schemes for Prolog

2.1 Prolog Programs

A Prolog program consists of a database and a question. The database consists of a set of facts, which represent properties of objects and/or relations among objects, and a set of inference rules, which are used to obtain an answer to the question. As an example, consider the following database.

Example 1

```
1:   offspring (John, Matt).
2:   offspring (Matt, Don).
3:   descendant (*X, *Y) :-
          offspring (*X, *Y).
4:   descendant (*X, *Y) :-
          offspring (*X, *Z),
          descendant (*Z, *Y). []
```

In this example, constant objects (John, Matt, and Don) are assumed to begin with alphabets, while variables (*X, *Y, and *Z) are assumed to begin with an asterisk.

The database in Example 1 describes the facts (lines 1 and 2):

1: "Matt" is an offspring of "John";
2: "Don" is an offspring of "Matt";

and the inference rules (lines 3 and 4):

3: If *Y is an offspring of *X,
then *Y is a descendant of *X;
4: If, for some *Z, *Z is an offspring of *X
and *Y is a descendant of *Z,
then *Y is a descendant of *X.

A question to the database is of the form like:

?- descendant (John, *A),

which means "Who are descendants of John?" The answer to this question will be:

*A=Matt;
*A=Don;

which means "Matt and Don are." And if a question:

?- descendant (John, Don),

which means "Is Don a descendant of John?" is given, the Prolog system will reply:

yes.

A question to the Prolog system is also called a **goal clause.**

2.2 Parallel Processing Schemes

An inference process in Prolog can be represented in the form of a state space search tree. As an example, the state space search tree that corresponds to the question "?- des(John, *A)" to the database of Example 1 is shown in Figure 1. In this figure, each node (box) represents a sub-goal of the given goal (or in other word, a sub-question of the given question). Each edge represents the super/sub relation between two goals; and the number on the right hand side of each edge represents the applied rule number. Here, "off" and "des" represent the relations "offspring" and "descendant" respectively.

Several parallel schemes for processing Prolog programs have been studied. They include: **OR-Parallel Scheme, AND-Parallel Scheme, Stream-Parallel Scheme,** and **Search-Parallel Scheme** [Cone81]. The basic idea of the **OR-Parallel Scheme** can be explained in the following way. The sub-goals in the state space search tree can be solved independently if there is no side effect; that is, each fact and inference rule can be unified at the same time. For example, clauses "off(John,*A)" and "off(John,*Z), des(*Z,*A)" in Figure 1 can be processed simultaneously.

In the **AND-Parallel Scheme,** all literals in the goal clause are unified to the corresponding literals in the right hand side of a inference rule. For example, two literals "off(John,*Z)" and des(*Z,*A)" in the clause

"off(John,*Z), des(*Z,*A)"

of Figure 1 can be unified at the same time. While this scheme introduces more parallelism then the OR-Parallel Scheme, this scheme could be inefficient because all possible combinations of values should be assigned to the variables included in the literals and investigated whether or not the assignment causes a contradiction.

The **Stream-Parallel Scheme** is one of the methods to realize the AND-Parallel Scheme. In this scheme, every literal is treated as an independent process, and each common variable is treated as a message channel. For example, "off(John,*Z)" and "des(*Z,*A)" in the "off(John,*Z), des(*Z,*A)" are considered as two independent processes, and "*Z" is considered as a message channel. However, efficient implementation of this scheme awaits future research.

In the **Search-Parallel Scheme**, the database is decomposed into subsets and they are placed in different PE's. The goal clause is sent to each PE and then examined simultaneously.

Because the OR-Parallel Scheme is the easiest to implement, many parallel Prolog systems are based on this scheme [Aida83], [Yasu83], [Maru84]. However, parallel Prolog systems based on the Stream-Parallel Scheme are also studied [Ito_84], [Onou85]. These systems are supposed to execute the programs written in Concurrent Prolog [Shap83].

```
           +------------+
           |des(John,*A)|
           +------------+
             |      |
     +-------+3     +-----------+4
     |                          |
     v                          v
  +-----------+       +-------------------------+
  |off(John,*A)|      |off(John,*Z), des(*Z,*A)|
  +-----------+       +-------------------------+
   |        |            |            |
   |1       |2           |1           |2
   |        |            |            |
   v        v            v            v
+-------+ +-----+  +-------------+  +-----+
|*A=Matt| |false|  |des(Matt,*A)|  |false|
+-------+ +-----+  +-------------+  +-----+
                     |   |
     +---------------+3  |4
     |                   |
     v                   v
 +-----------+   +-------------------------+
 |off(Matt,*A)|  |off(Matt,*Z), des(*Z,*A)|
 +-----------+   +-------------------------+
   |       |         |         |
   |1      |2        |1        |2
   |       |         |         |
   v       v         v         v
+-----+ +------+ +-----+ +------------+
|false| |*A=Don| |false| |des(Don,*A)|
+-----+ +------+ +-----+ +------------+
                           |    |
           +---------------+3   |4
           |                    |
           v                    v
       +----------+    +------------------------+
       |off(Don,*A)|   |off(Don,*Z), des(*Z,*A)|
       +----------+    +------------------------+
        |       |          |          |
        |1      |2         |1         |2
        |       |          |          |
        v       v          v          v
     +-----+ +-----+    +-----+    +-----+
     |false| |false|    |false|    |false|
     +-----+ +-----+    +-----+    +-----+
```

Figure 1 - Computation Process of a Prolog Program

III. Tree Machine Model

3.1 Conventional Tree Machines

Tree machines are loosely coupled systems where PE's are connected to form a tree structure. In these machines, each PE has its own local memory unit and works independently. Messages are transferred between PE's through message links that connect them. One of the advantages of tree machines in problem solving is that the structure of these machines fit the state space search trees of problems.

Though tree machines fit state space trees, there are several drawbacks to tree machines. One of them is that the load tends to concentrate at PE's near the root of the tree. Load balancing is one of the critical problems in designing efficient algorithms for tree machines.

Typical structures of tree machines are shown in Figure 2. In this figure, circles and rectangles denote PE's, and arrows denote the message links. Messages are transferred along the direction of these arrows. System (a) is the simplest model called a single-tree machine [Brow80], [Taka81], [Shaw81], and [Stol82]. DADO [Stol82] and NON-VON [Shaw81] are designed for production systems and expert systems. System (b) and (c) are called double-tree machines, which include Bentley's machine (b) for database operations such as sorting and searching [Bent79], and the DON System (c) for combinational optimization problems [Imai84].

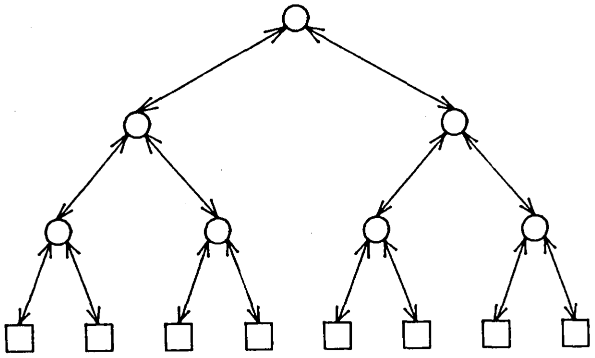
The double-tree machines have several advantages over the single-tree machines. In the single-tree machines, messages are transferred in both directions of the message links, while they are transferred in only one direction of the message links in the double-tree machines. As a result, effective bandwidth of a message link in the double-tree machines is twice as wide as that in the single-tree machines. And it is easier to implement algorithms that have pipeline features on double tree machines than single-tree machines. Therefore, double-tree machines will have better performance than single-tree machines. Other advantages of double-tree machines are discussed in the literature [Imai84].

3.2 The Tree Machine Model

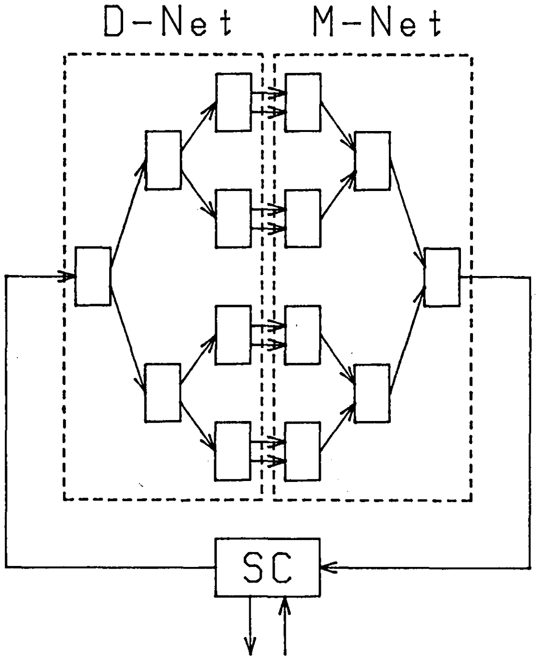
The model of the tree machine used in the simulation experiment is shown in Figure 3, which consists of a single-tree engine, a multiplexer, and a system controller. The reason behind the selection of this model is that the single-tree engine is a common part to all tree machines and has generality. The system controller is involved in this model in order to improve the efficiency of the tree machine.

In this model, it is assumed that messages can be transferred through the multiplexer quickly enough not to affect the efficiency of the computation in the single-tree engine and the system controller. Note that this model is more efficient than other tree machines under this assumption.

Especially, in the single-tree machine, messages must be transferred in both direction of the message links, which causes conflicts at PE's. Therefore, single-tree machines will be less efficient than this tree machine model. And double-tree machines will be about as efficient as this tree machine model.

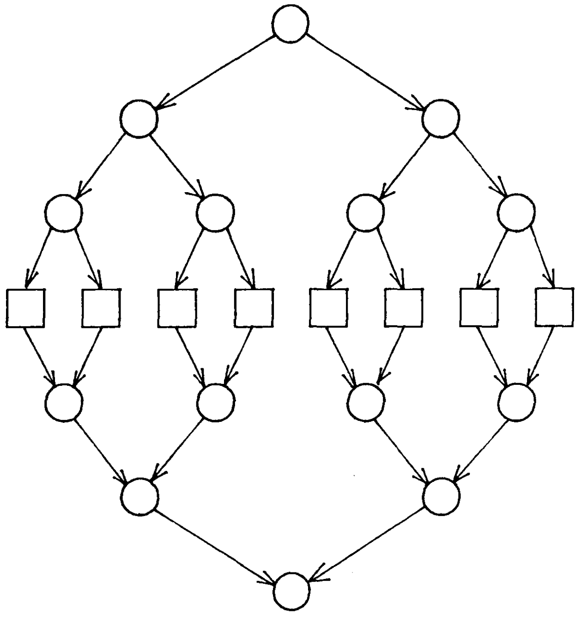


(a) Single-Tree Machine

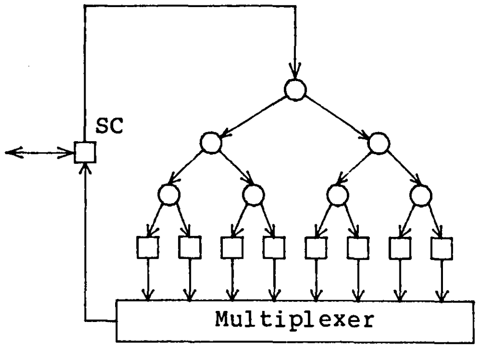


(c) The DON System

Figure 2 - Typical Tree Machines



(b) Double-Tree Machine



SC: System Controller

Figure 3 - The Tree Machine Model

IV. Prolog Implementation on Tree Machines

4.1 Observation

The process in which a Prolog program is executed can be represented in the form of a state space search tree like Figure 1. Comparing the structure of this tree with the interconnection topology of the tree machines, the following observations can be made. First, while the number of successors of a node in the state space search tree is indefinite, internal PE's of the tree machines have up to two successors or up to two predecessors (in the case of double tree machines).

Secondly, the depth of a leaf node of a search tree is also indefinite. As a consequence, it would not be effective to map a search tree directly to a tree machine, because some of the nodes of the search tree could not be assigned to PE's and the load of PE's would not be balanced. Therefore, it is important to design efficient algorithms that distribute the load to the PE's as evenly as possible.

4.2 Assumptions

For the rest of this paper, the OR-Parallel Scheme is implemented because the OR-Parallel Scheme is the easiest to implement on multiprocessor systems. The purpose of the simulation experiment is to investigate the efficiency of tree machines in the application to parallel Prolog implementation. Evaluation of other parallel schemes awaits future work.

The implementation of Prolog in this paper is an extension of the scheme for the DON System as proposed by Sano, et al. [Sano84]. In this scheme, programs are written in "pure Prolog," where no side effect is allowed and no "cut operator" is used. The outline of the OR-Parallel Scheme used in this experiment is as follows. Further details of this Scheme are shown in the literature [Miur85].

(1) The unification operation is executed simultaneously in each PE and the database of the program is stored in each PE prior to the execution.

(2) Each PE has an input queue to store the messages. A message sent from the predecessor PE is placed at the tail of the input queue, and the PE gets a message from the head of the queue in the order of arrival.

(3) If the depth of the state space search tree is deeper than that of the processor tree, the unsolved sub-goals are resent to the root PE and processed in the processor tree in the pipeline manner.

(4) The Prolog interpreter uses the "copy method" instead of the "structure sharing method." The source program is converted into an intermediate form. The goal clause is also converted into the intermediate form.

4.3 Decomposition of the Set of Unifiable Clauses

When a goal clause is examined in the tree machine, the set of unifiable clauses is decomposed into two or three subsets in order to make the load of each PE balanced. There are two decomposition methods used in the experiments.

Method D-2 (Decomposition into two):

Decompose the set of unifiable clauses into two subsets that have as equal a number of clauses as possible; then, send them to successors. []

Method D-3 (Decomposition into three):

Keep one of the clauses for the PE, and decompose the rest of the clauses into two subsets that have as equal a number of clauses as possible; then send the subsets of clauses to successors. []

The difference between Methods D-3 and D-2 is that one of the unifiable clauses is kept for the PE in Method D-3. As an example, suppose that a goal clause can be unified with clauses #1 through #5. In Method D-2 (decomposition into two), clauses #1 through #3 will be sent to the left successor and clauses #4 and #5 will be sent to the right successor. On the other hand, in Method D-3 (decomposition into three), clause #1 is kept for the PE, clauses #2 and #3 are sent to the left successor, and clauses #4 and #5 are sent to the right successor.

4.4 Control Procedures

Though the use of Method D-3 is expected to reduce the message transfer overhead, it is possible that the PE's near the root of the tree become a bottleneck. To avoid such a situation, the control procedures switch the use of these decomposition methods according to the following information:

(1) Depth of the PE, and
(2) Number of goal clauses stored in message queues.

However, in the leaf PE's, Method D-3 is always applied in order to make the leaf PE's as busy as possible. Because the number of leaf PE's is about the half of that of all PE's in the system, this strategy will yield a better utilization of PE's.

Definition

Let d(i) be the depth of PE#i, which is defined as the path length from the root PE to PE#i. And let q(i) be the number of goal clauses stored in the message queue of PE#i. []

Three control procedures D, Q1, and Q2 for the non-leaf PE's are given in the following way.

(1) Control Procedure D

The basic idea of procedure D is to divide the set of PE's into two subsets. Then, Method D-2 is used in the PE's which are near to the root PE, while Method D-3 is used in the PE's which are far from the root PE. More formally, this procedure is described as follows.

Control Procedure D:

```
Let $D_t$ be a positive parameter.
If $d(i) \geq D_t$ then use Method D-3;
otherwise, use Method D-2. []
```

(2) Control Procedure Q1

In procedures Q1 and Q2, the number of messages in PE's are used to switch the decomposition procedures. In procedure Q1, if the number of messages in the PE does not exceed the given threshold, then Method D-3 is used; otherwise Method D-2 is used. Procedure Q1 is described as follows.

Control Procedure Q1:

```
Let $Q_t$ be a positive parameter.
If $q(i) \leq Q_t$ then use Method D-3;
otherwise, use Method D-2. []
```

(3) Control Procedure Q2

Control procedure Q2 takes advantage of the number of messages in the successor PE's. Let PE#j and PE#k be the successor PE's of PE#i. The number of messages in the message queues of the PE#i, PE#j, and PE#k are used to switch the decomposition method. After the decomposition, sub-goals are sent to the successors so that the number of messages in the message queues of PE's #j and #k becomes as equal as possible. There are three variations of this method.

Procedure Q2-a:

```
If $q(i) \leq q(j)$ or $q(i) \leq q(k)$
then use Method D-3;
otherwise, use Method D-2. []
```

Procedure Q2-b:

```
If $q(i) \leq q(j) + q(k)$ then use Method D-3;
otherwise, use Method D-2. []
```

Procedure Q2-c:

```
If $q(i) \leq \min[\, q(j), q(k) \,]$
then use Method D-3;
otherwise, use Method D-2. []
```

V. Simulation Experiments

5.1 Assumptions

Simulation experiments have been conducted to investigate the efficiency of the tree machines that execute Prolog programs. The following assumptions have been made in the simulation experiments:

Assumption 1
Any number of messages can be stored in an input queue of a PE. []

Assumption 2
The computation time needed to manipulate the message queue (such as insert and remove) is small enough to ignore compared to the computation time needed to process a clause. []

The unit of the computation time is the time needed to move one Byte of data between memory cells. The computation time needed to compare two data items (one Byte each) is assumed to take **one unit** of time, and the computation time to transfer one Byte of data item between adjacent PE's is assumed to take **two units** of time because the message link is slower than the internal bus in a PE.

Let T(n) be the computation time required to solve a problem by a tree machine that has n PE's. Then the **speedup rate Rs**(n) of the tree machine is defined by:

$$Rs(n) = T(1) / T(n)$$

Note that the minimum configuration of the single-tree engine has only one PE.

The following test problems were solved in the simulation experiments:

(1) Quicksort for 50 items [Okun84],
(2) Ancestor/descendant database search problems [Sano84],
and
(3) N-Queens problems [Okun84], where all solutions are searched.

5.2 Experimental Results

Experimental results for the above problems are summarized in Figures 4 through 7. In each figure, the x coordinate denotes the number of PE's in the single-tree engine of the tree machine model, on a log scale; and the y coordinate denotes the average speedup rate (Rs), also on a log scale. From these figures, the following observations can be made.

(1) Quicksort

The results for the quicksort problem for the control procedure Q1 are summarized in Figure 4. The speedup rate of the system did not exceed 1.5, due to the lack of parallelism in the quicksort algorithm. The results when control procedures D and Q2 are used are almost the same as those shown in Figure 4.

(2) Ancestor/Descendant Database Search

The results are summarized in Figure 5. Here, the results for the control procedure D are shown in Figure 5 (a), and the one for the control procedure Q1 are shown in Figure 5 (b).

When control procedure D is used, the best speedup rate was obtained by setting $D_t = 1$. And better speedup rates were obtained by setting $D_t = 2$, 0, and 3, in this order. When control procedure Q1 is used, almost the same best speedup rate was obtained by setting $Q_t = 1$, 2, or 3, except 0. The results when control procedure Q2 is used are almost the same as those shown in Figure 5 (b).

(3) N-Queens Problems

The results for the 5-Queens problem are summarized in Figure 6. Here, the results for the control procedure D are shown in Figure 6 (a), and the one for the control procedure Q1 are shown in Figure 6 (b). When control procedure D is used, the best speedup rate was obtained by setting $D_t = 3$.

When control procedure Q1 is used, the speedup rate was only slightly affected by the value of parameter Q_t. The results when control procedure Q2 is used are almost the same as those shown in Figure 6 (b).

(4) The Effect of the Amount of Parallelism in the Problems

The efficiency of the control procedure is affected by the properties of problems being solved. The speedup rates of the tree machine for the 5-Queens Problem and the 6-Queens Problem are shown in Figure 7. The control procedure used in this experiment is Q1 with $Q_t = 0$.

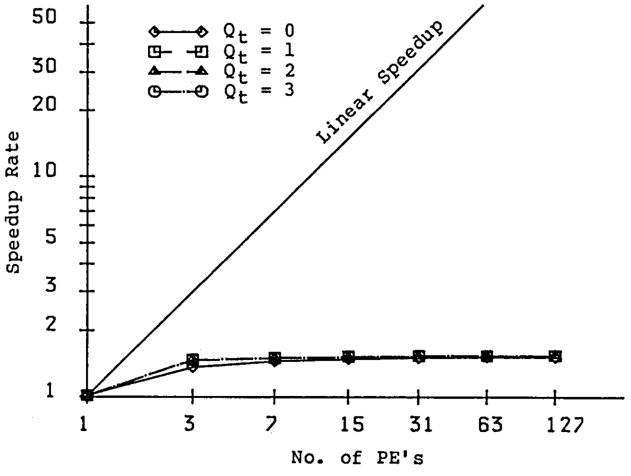
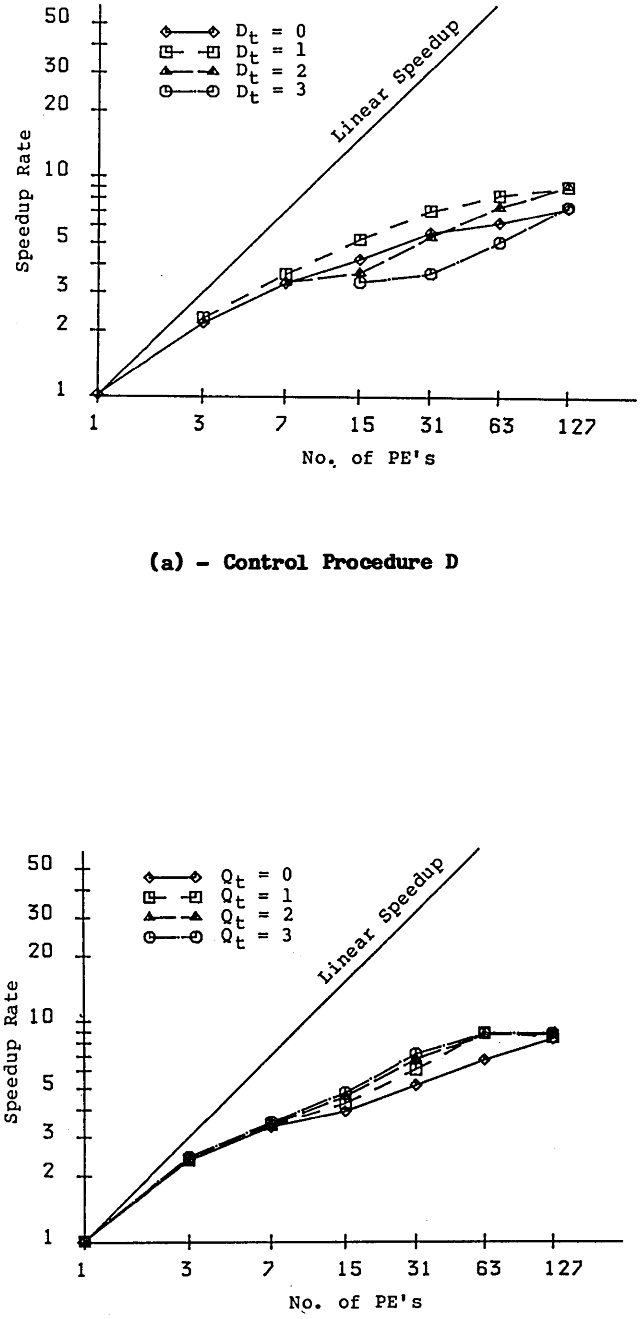


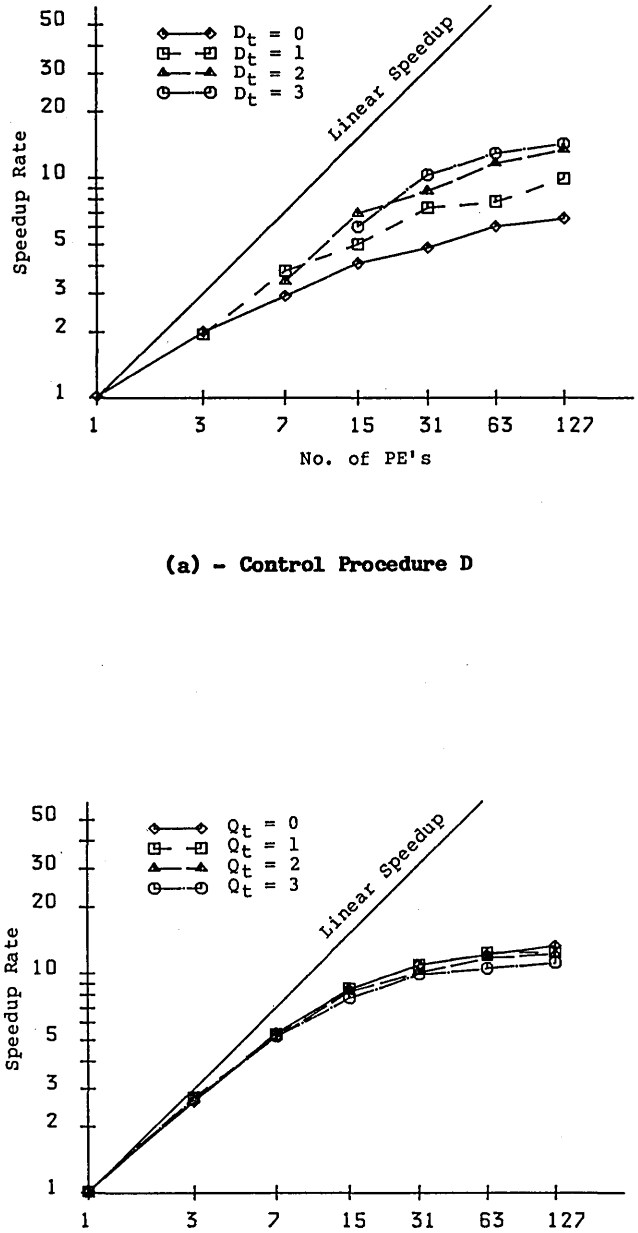
Figure 4 - Speedup Rate for the Quicksort Problem (Control Procedure Q1)



(a) - Control Procedure D

(b) Control Procedure Q1

Figure 5 - Speedup Rate for the Ancestor/Descendant Database Search Problems



(a) - Control Procedure D

(b) - Control Procedure Q1

Figure 6 - Speedup Rate for the 5-Queens Problem

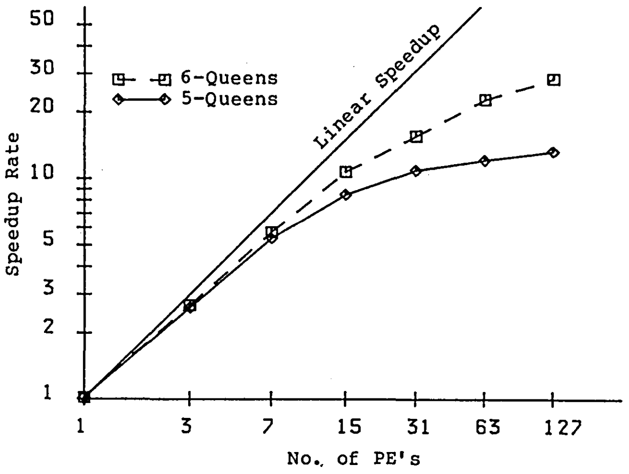


Figure 7 - Comparison of the Speedup Rates for 5-Queens Problem and 6-Queens Problem
(Control Procedure Q1 with $Q_t = 0$)

5.3 Discussion

From the above observations, the following discussion can be made.

(1) The Choice of Control Procedure

Comparing this result with that of the database search problems, it is known that the optimum value of parameter D_t changes depending on the problems to be solved. And the efficiency of control procedure D is also affected by the types of problems to be solved. Though this procedure might bring the best results by chance, it does not steadily yield a good efficiency. Therefore, it is not considered the best method.

On the other hand, control procedures Q1, Q2-a, Q2-b, and Q2-c show similar results. The value of the parameter Q_t slightly affects the efficiency of the control procedure Q1. In regard to the ancestor/descendant relation database search problem, procedure Q1 is a little more efficient than Q2 procedures. In conclusion, control procedure Q1 is considered the best method among procedures D, Q1, and Q2's.

(2) The Effect of the Amount of Parallelism in Problems

Comparing the results of the Quicksort Problem (Figure 4), 5-Queens Problem, and 6-Queens Problem

(Figure 7) with each other, it is known that the speedup rate of the tree machines is improved for larger problems which have more parallelism. The efficiency of the tree machine is affected by the possible parallelism included in the problems to be solved. If the problems have enough parallelism, the tree machines show a good speedup rate compared to the number of PE's.

(3) Comparison with Other Machines

The speedup rate of the tree machine model is comparable to that of another parallel inference machine PIM-R designed by ICOT [Onou85], where the speedup rate of the PIM-R machine is up to three times as fast for nine PE's. Comparison with other machines, such as DADO or NON-VON, awaits future work.

(4) Load Balancing

In regard to the load balancing, the load tends to be concentrated at PE's near the root of the tree, depending on the problems being solved. Especially, in problems like the Ancestor/Descendant Database Search Problems where the branching factor is large, the load still tends to be concentrated at the root PE.

For example, according to the simulation results, when these problems are solved on the tree machine with 127 PE's (except the system controller) with the control procedure Q1 ($Q_t = 0$), the maximum and average number of messages stored in the message queue of the root PE were 39 and 14 respectively.

Accordingly, the efficiency of the tree machines could be improved by developing more sophisticated control procedures than those proposed in this paper. The improvement of algorithms is one of the future research problems.

VI. Conclusion

In this paper, a tree machine model was described to implement parallel Prolog schemes. Then, an OR-parallel Scheme for Prolog implementation was shown, and several control procedures were proposed in order to make the load better balanced. Finally, the efficiencies of these control procedures were compared through simulation experiments.

Among the proposed control procedures, Procedure Q1 showed the best results, which switches the decomposition methods according to the number of messages in the message queue. Though the efficiency of the tree machine is affected by the potential parallelism included in the problems, if the problems have enough parallelism, the speedup rate of tree machines is comparable to that of other multiprocessor systems that have been investigated.

In conclusion, tree machines are promising candidates for future super computer systems that solve complicated knowledge-based problems.

The following will be included in future research projects:

(1) Introduction of control structures to Prolog which play the role of the "cut" operator.
(2) Study on the implementation of the predicates that cause side effects, such as database manipulation operations.
(3) Development of more sophisticated algorithms that achieve better load balancing and better utilization.
(4) Study on the implementation of the AND-Parallel Scheme, in particular the Stream-Parallel Scheme, on tree machines.
(5) Study on the parallel Prolog machines based on the Search-Parallel Scheme, that have a distributed database system.

Acknowledgements

The authors would like to thank Mr. Sano of Sony Corp. who designed the preliminary version of the simulation program as a part of his Master's project. The authors are also grateful to President Honda of Toyohashi University of Technology and Prof. M. Everson of Nagoya University for their helpful comments and encouragement. This work was supported in part by the Ministry of Education, Culture and Science of Japan.

References

[Aida83] H. Aida, H. Tanaka, T. Moto-oka: On Parallel Processing System "Paralog", Jour. of the IPS of Japan, Vol. 24, No. 6, pp. 830-837 (Nov. 1983); in Japanese.

[Bent79] J.L. Bentley, H.T. Kung: A Tree Machine for Searching Problems, Proc. Int'l Conf. on Para. Proc., pp. 259-268 (1979).

[Brow80] S.A. Browning: The Tree Machine: A Highly Concurrent Computing Environment, Ph.D. Thesis, Caltech (Jan. 1980).

[Cloc81] W.F. Clocksin, C.S. Mellish: Programming in prolog, Springer-Verlag Pub. Co., (1981).

[Cone81] J.S. Conery, et al.: Parallel Interpretation of Logic Programs, Proc. of Functional Prog. Lang. and Comp. Archi., ACM (1981).

[Feig81] E.A. Feigenbaum, P. McCorduck: The Fifth Generation: Artificial Intelligence and Japan's Computer Challenge to the World, Addison-Wesley, Reading, Mass. (1983).

[Horo83] E. Horowitz, A. Zorat: Divide-and-Conquer for Parallel Processing, IEEE, Trans. on Comp., Vol. C-32, No. 6, pp. 582-585 (June 1983).

[Ito_84] Y. Ito, Y. Masuta, et al.: The Architecture of a Parallel Inference Machine based on the Data-Flow Computation, ICOT, proc. of the 1984 Logic Programming Conference (1984); in Japanese.

[Imai84] M. Imai, et al.: The Architecture and Efficiency of DON: A Combinatorial Problem Oriented Multicomputer System, Proc. of the 4th Int'l Conf. on Distributed Computing Systems, pp. 174-182 (May, 1984).

[Kowa79] R. Kowalski: Logic for Problem Solving, North Holland Pub. Co, (1979).

[Maru84] T. Maruyama, K. Hirata, et al.: The Architecture of A highly Parallel Inference Engine - PIE - and its Simulator, IECE of Japan, Papers of Tech. Group of Electrical Computer, EC 84-45, pp. 1-11, (1984); in Japanese.

[Miur85] H. Miura: Implementation and Efficiency of the OR Parallel Prolog System on a Double-Tree Structured Multicomputer System DON, Master Thesis, TUT (1985); in Japanese.

[Moto82] T. Moto-oka: Fifth Generation Computer Systems, North-Holland Pub. Co., (1982).

[Nils80] N.J. Nilsson: Principles of Artificial Intelligence, Tioga Pub. Co. (1980).

[Nitt84] K. Nitta: Parallel Prolog, Proc. of IPS of Japan, Vol. 25, No. 12, pp. 1353-1359 (Dec. 1984); in Japanese.

[Okun84] H. Okuno: Proposal of the Bench Mark Programs for the Third Lisp Contest and the First Prolog Contest, IPS of Japan, Papers of the Tech. Group on Symbolic Processing, SP 28-4 (1984); in Japanese.

[Onou85] M. Onouchi, et al.: Software Simulation of the Parallel Inference Machine PIM-R, IPS of Japan, Proc. of the 30th Nat. Conf., 6C-9, pp. 201-202 (1985); in Japanese.

[Pete81] F.J. Peters: The Tree Machines and Divide-and-Conquer Algorithms, Lect. Note in Comp. Sci., Vol. 111, pp. 25-36 (1981).

[Shap81] E.Y. Shapiro: A Subset of Concurrent Prolog and its Interpreter, Tech. Rep. TR-003, ICOT (1983).

[Shaw81] D.E. Shaw: NON-VON: A Parallel Machine Architecture for Knowledge-Based Information Processing, Proc. of the 7th Int'l Conf. on AI, pp. 961-963 (1981).

[Stol82] S.J. Stolfo, D.E. Shaw: DADO: A Tree Structured Machine Architecture for Production Systems, Proc. of Nat. Conf. on AI (Aug. 1982).

[Taka81] Y. Takahashi, et al.: A Binary Tree Multiprocessor: CORAL, Jour. of IPS of Japan, Vol. 3, No. 4, pp. 230-237 (1981).

[Yasu83] Yasuhara, et al.: On an OR Parallel Model: ORBIT, Proc. of the 1983 Logic Programming Conference, ICOT (1983); in Japanese.

IECE: Institute of Electronics and Communication Engineers
IPS : Information Processing Society

OPTIMAL GRANULARITY OF PARALLEL EVALUATION OF AND TREES

Guo-Jie Li and Benjamin W. Wah
Department of Electrical and Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Avenue
Urbana, IL 61801

ABSTRACT

AND-tree evaluation is an important technique in artificial intelligence and operations research. An example is the divide-and-conquer algorithm, which can be considered as the evaluation of a precedence graph consisting of two opposing AND-trees. In this paper, the optimal granularity of parallelism of AND-tree computations is quantitatively analyzed. The efficiency analysis is based on both preemptive and nonpreemptive critical-path scheduling algorithms. It is found that the optimal grain depends on the complexity of the problem to be solved, the shape of the precedence graph, and the task-time distribution along each path. The major results consist of tight bounds on the number of processors, within which the optimal grain can be sought efficiently. In view of the optimal granularity, architectural requirements for parallel AND-tree evaluations are also discussed.

INDEX TERMS: AND trees, critical-path scheduling, divide-and-conquer algorithms, granularity, processor-time efficiency, processor utilization.

1. INTRODUCTION

A wide class of problems arising in artificial intelligence, operations research, decision making, and various scientific and engineering fields involve finding a solution of a problem, which is made up of a large number of subproblems to be solved. Solving these subproblems can be represented as AND-tree computations. Examples include evaluating arithmetic expressions, searching possible solution trees of logic programs, evaluating functional programs, scheduling operations in assembly lines, finding the extremum, merge-sorting, and quick-sorting.

There are two kinds of AND-trees, *intrees* and *outtrees*. In an intree (resp. outtree), each node has at most one immediate successor (resp. predecessor), and the root is an exit node (resp. entry node). The intrees and outtrees specify the precedence relationships among the nodes. Every node is reachable from the entry node for an outtree or can reach the exit node for an intree. In recursive computations, such as divide-and-conquer algorithms, a problem is partitioned into smaller and distinct subproblems, and the solutions are found for the subproblems and are combined into a solution for the original problem. The procedure is applied recursively until the subproblems are so small that they can be solved directly. In this way, the evaluation can be viewed as a process with two phases, the decomposition of subproblems based on an outtree and the composition of results based on an intree. Hence, the precedence graph is composed of an intree and an outtree. We call this particular graph an *outin tree*. Deterministic programs can be represented by outin trees. Functional programming, which is considered as an important programming style to resolve the von-Neumann bottleneck, deals exclusively with AND-graphs [1]. Similarly, data-flow graphs are AND-graphs [7].

Outin trees have the following characteristics. First, there is no cycle in an outin tree. Second, in contrast to general forests, an outin tree consists of one outtree and one intree and have a one-to-one correspondence of all the leaves in the two trees. We call these leaves the ***leaves of the outin tree***. In this paper, we will mainly discuss outin trees. However, the results derived apply to intrees and outtrees as well. Here, AND-trees and outin trees are used synonymously. Figure 1(a) illustrates an outin tree, which reflects the precedence relationships among tasks in a merge-sort problem shown in Figure 1(b) to sort six elements. The nonterminal nodes in the outtree part represent decompositions, each of which split a (sub-)list into two smaller sublists, whereas the nonterminal nodes in the intree part represent composition, each of which generate a sorted list based on two smaller sorted sublists.

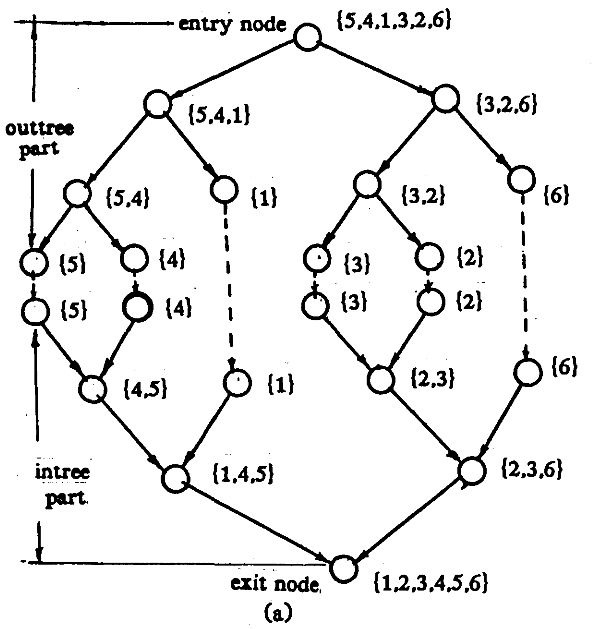
Evaluation of outin trees naturally suggests implementation on parallel computers due to the independence of subproblems. AND-tree evaluations are important in evaluating logic programs, especially when parallel processing is used [5]. Studies conducted on parallel computers for executing divide-and-conquer algorithms can be classified into three types. First, multiprocessors that are connected in the form of a tree, especially a binary tree, can be used to exploit the potential parallelism of divide-and-conquer algorithms [13, 23]. A second approach is the virtual tree machine [2], which consists of a number of processors with private memory connected by an interconnection network, such as the binary n-cube, and a suitable algorithm to decide when and where each subproblem should be solved. The third approach is a variation of the above approaches using a common memory. All processors are connected to the memory by a common bus [14].

To evaluate an AND-tree in parallel, it is necessary to schedule the subproblems to achieve high throughput and processor utilization. An important problem is to determine the proper granularity of parallelism, that is, the minimum size of a subproblem that should be computed by a single processor. If the grain is too large, then the processors can be loosely coupled but may be under-utilized. In contrast, if the grain is too small, then the processors can be better utilized, but tight coupling may be necessary, and the communication overhead may be prohibitive. The grain must be properly chosen to obtain a proper balance between processor utilization and communication overhead.

In previous studies, one can find different points of view on the issue of granularity. Some researchers advocate a fine grain, while others suggest a coarse grain. For example, in designing the FFP machine [21], a small grain is chosen based on the hypothesis that appropriately designed small-grain multiprocessors will prove superior to large-grain ones in supporting ease and generality of parallel computations. In contrast, in

Research supported by National Science Foundation Grant DMC 85-19649.

ACM-IEEE Computer Society 1986 Fall Joint Computer Conference.



```
Procedure mergesort
  integer low, high
  if low < high
    then call split (low, high, mid)
      call mergesort (low, mid)
      call mergesort (mid+1, high)
      call merger (low, mid, high)
  endif
end mergesort
```

(b)

Figure 1. The merge-sort problem represented as an outin tree.

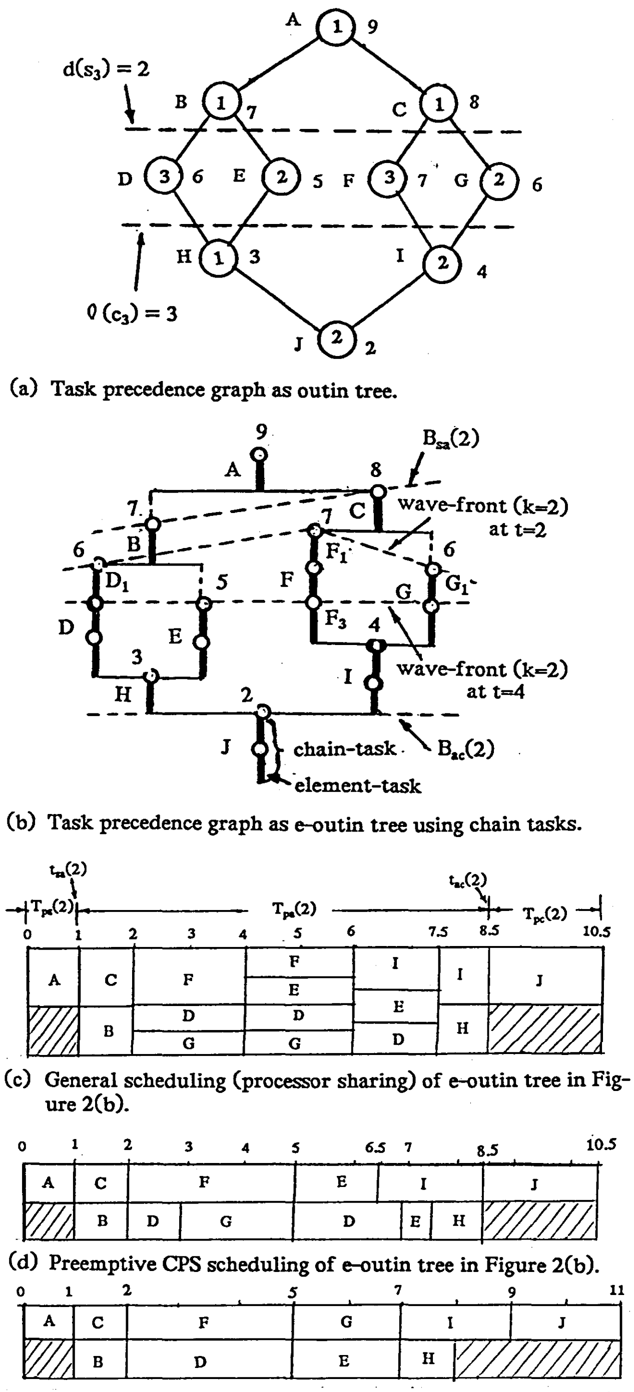
Rediflow [17], large-grain parallelism is used to minimize communication overheads. Moreover, most previous studies on granularity were discussed qualitatively. In this paper, we will analyze the optimal granularity of parallel evaluation of outin trees quantitatively. We will identify the factors that influence the optimal grain, in particular, the relationship between the optimal granularity and the problem complexity.

2. SCHEDULING PARALLEL OUTIN-TREE EVALUATIONS

To analyze the optimal granularity of parallel evaluation of outin trees, an asynchronous model for parallel computation is adopted here. The precedence graph of an outin tree is oriented such that the entry node is at the top of the figure and the exit node is at the bottom. An arc is assumed to be always directed towards the bottom of the graph. The number inside a node is the task execution time, while the number next to a node, called its *length*, is the sum of the task execution times for nodes in the longest path from this node to the exit node. Figure 2(a) is an example of an outin tree.

The execution time of a task can be interpreted as either its maximum processing time or its expected processing time. In the former case, the worst-case time to complete the schedule is considered, while in the latter case the length of the schedule represents a rough estimate of the average time of computation. In some outin-tree problems, the execution time of each task can be predicted quite accurately. For example, in evaluating arithmetic expressions, the time to execute a primitive operation, such as a multiplication, is known. In other cases, the average execution times may have to be estimated from statistics or from previous experience. In all cases, the communication overhead is non-trivial when preemptions are allowed, and the task time should also include the overhead of preemptions.

Our goal is to choose an algorithm that minimizes the maximum completion time for scheduling outin trees on a set of P identical processors, and to find the optimal granularity based on



(a) Task precedence graph as outin tree.

(b) Task precedence graph as e-outin tree using chain tasks.

(c) General scheduling (processor sharing) of e-outin tree in Figure 2(b).

(d) Preemptive CPS scheduling of e-outin tree in Figure 2(b).

(e) Nonpreemptive CPS scheduling of outin tree in Figure 2(a).

Figure 2. Outin tree and CPS scheduling.

this scheduling algorithm. Our scheduling problem is similar to the P/tree/C_{max} scheduling problem in which tree precedence graphs are considered [12, 6, 4]. Note that the proper granularity is related to the scheduling algorithm. If the underlying scheduling algorithm does not minimize the completion time, then the granularity found with respect to this scheduling algorithm is suboptimal. In this paper, our analysis of optimal granularity is based on optimal scheduling algorithms. The method of granularity analysis used here can be applied to choosing the best grain for other scheduling algorithms.

If preemption is allowed, P/preemption,intree/C_{max} can be solved optimally either by Muntz and Coffman's Critical Path Scheduling (CPS) algorithm in O(N) time [22], or by other polynomial-time algorithms [9]. In the CPS algorithm, the next job chosen is the one with the longest length of unexecuted jobs. This longest path is called the *critical path*. If preemption is not allowed, then optimal scheduling algorithms have been obtained only for two cases: (a) all tasks have equal execution times and the precedence relationships are in the form of an intree (Hu's algorithm) [15] and (b) when two processors are used [3]. Hu's optimal scheduling algorithm is indeed a CPS algorithm. Many other cases have been proved to be NP-hard [25, 19]. Besides efficient and optimal, the CPS algorithm is easy to implement and, consequently, is one of the most common scheduling algorithms [20, 18].

In case that the precedence graph is a tree, that all processors are identical, and that each task requires t_i, $0<t_i \leqslant t_{max}$, units of time to complete, the nonpreemptive CPS algorithm turns out to be almost-optimal in the sense that

$$T_p(k) \leqslant T_{np}(k) \leqslant T_p(k) + t_{max} \qquad (2.1)$$

where $T_{np}(k)$ and $T_p(k)$ are, respectively, the total times required by the nonpreemptive and preemptive CPS algorithms using k processors [16]. Some researchers have strived for nonpreemptive scheduling algorithms to solve scheduling problems with tree precedence [12, 8, 18]. Recently, Garey, Dolev, et al. have studied the scheduling of forests consisting of intrees and outtrees. Given a fixed number of processors, polynomial algorithms with high complexities to find an optimal schedule of these forests have been developed [10, 8].

In our research, we are interested in a special, but widely used, case of outin trees. From the point of view of optimal granularity, evaluating an outin tree can be divided into the *splitting*, *all-busy*, and *combining* phases with respect to k, the fixed number of processors. In the splitting phase, the problem is decomposed, and the number of busy processors is increased from one up to at most k−1 (the number of busy processors must always be less than k if the number of available tasks at any time is less than k). In the combining phase, the subproblems are composed, and the number of busy processors is decreased from at most k−1 to one. During these two phases, some processors are idle. In contrast, in the all-busy phase, all the k processors are busy. Schindler has proved that the schedule of a precedence graph is optimal if either the computations can be completed in only the all-busy or combining phase, or it can be partitioned into the all-busy and combining phases by a "heightline" [24]. We will show that this result can be extented to scheduling outin trees, and that the CPS algorithm guarantees the optimal preemptive scheduling and near-optimal nonpreemptive scheduling of outin-tree evaluations.

To analyze the properties of *preemptive CPS*, in short, *PCPS*, algorithms, it will be more convenient to represent a task (a node of the outin tree) of execution time t_i by a chain-task, which is t_i *element-tasks* (or *element-nodes*, or in short, *e-tasks* or *e-nodes*), each of which has one unit of execution time (see Figure 2(b)). We will use a subscript i in the task identifier to indicate the i'th e-task in the chain-task. Hence, F_2 is the second e-task of task F. The new outin tree is called the *element-outin tree* (or *e-outin tree*). For each chain-task, the e-task farthest from the exit e-node of the e-outin tree is called a *task-head* e-node. It is easy to verify that the length of the task-head e-node is the same as the length of the original multiunit task. Two e-nodes are said to be in the *same e-level* of the e-outin tree if their lengths are identical, that is, the e-level number of an e-node is equal to its length assuming that the exit e-node is in Level 0. To distinguish between nodes and levels in the original outin tree (as exemplified by Figure 2(a)) and in the e-outin tree (as exemplified by Figure 2(b)), we will use *tasks* and *levels* with respect to the original outin tree and *e-task* and *e-level* with respect to the e-outin tree in subsequent discussions.

There is another variation of preemptive scheduling algorithms called General Scheduling (GS) discipline [22], which is extremely useful in studying the granularity of parallel outin-tree evaluations. In the GS algorithm, each processor in the system is considered to have a certain amount of computing capacity rather than as a discrete unit, and this computing capacity can be assigned to tasks in any amount between zero and the equivalence of one processor. For example, if we assign half of a processor to task P_i with execution time t_i, then it will take $2 \cdot t_i$ units of time to complete P_i. In the GS discipline, one processor is assigned to each of the k e-nodes farthest from the exit e-node of the e-outin tree to be evaluated. If there is a tie in the lengths among u e-nodes for the last v, u>v, processors, then v/u of a processor is assigned to each of these u e-nodes. Each time when either (a) a chain-task of the e-outin tree is completed, or (b) a point is reached where, if we continue with the present assignment, some e-nodes will be computed at a faster rate than other e-nodes that are farther from the exit e-node, then the processors are reassigned to the remaining tree according to the CPS principle. Situation (b) occurs when an e-node that is being computed has the same length as that of some unexecuted task-head e-node(s). In this case, one (or part of a) processor must be assigned to the unexecuted task-head e-node.

The GS discipline is illustrated in Figure 2(c). Muntz and Coffman have proved the equivalence between the GS and PCPS algorithms [22]. That is, if preemptions are permitted, then the "processor-sharing" capability is not needed for optimal scheduling. To illustrate this equivalence, Figure 2(d) shows the preemptive schedule for the corresponding e-outin tree in Figure 2(b). Note that in the scheduling algorithms discussed in this paper, all idle processors, if any, must be used to compute an available executable task.

In practice, preemptions are usually restricted at the beginning of a time unit, so the overhead of a practical PCPS algorithm is equal to that of Hu's algorithm, which assumes that tasks have unit execution times. From Eq. (2.1), we have [22]

$$T_p(k) = T_{gs}(k) \leqslant T_h(k) \leqslant [T_{gs}(k) + 1] = [T_p(k) + 1] \quad (2.2)$$

where $T_h(k)$ and $T_{gs}(k)$ are the times required by Hu's and GS algorithms, respectively. Eq. (2.2) shows that the behavior of GS is very close to that of any PCPS algorithm that only allows preemptions at the beginning of a time unit. In subsequent discussions, the results will be derived without any restriction on the allowable times for preemptions. Moreover, we will use GS as a model to analyze the properties of PCPS algorithms. The granularities derived are the same as those when the PCPS algorithm is used.

At time t, an e-node is said to be *active* if either a processor or part of a processor is assigned to it. The total number of active e-nodes may be greater than the number of processors since some e-nodes may share processors. All active e-nodes form a wave-front in the e-outin-tree evaluation. Two particular times of the wave-front are of special interest: $t_{sa}(k)$ and $t_{ac}(k)$. The computation enters the all-busy phase at $t_{sa}(k)$ and enters the combining phase at $t_{ac}(k)$. In both times, the wave-fronts serve as phase-boundaries. We call the former phase-boundary $B_{sa}(k)$ and the latter $B_{ac}(k)$.

For the task graph in Figure 2(a), if the PCPS algorithm is employed, then $t_{sa}(2)$=1 and $t_{ac}(2)$=8.5 (see Figures 2(c) and 2(d)). The corresponding phase-boundaries $B_{sa}(2)$ and $B_{ac}(2)$ are indicated in Figure 2(b).

If a preemptive (resp. nonpreemptive[*]) CPS algorithm is applied, then the computational times required by k processors to complete the splitting, all-busy and combining phases are denoted by $T_{ps}(k)$, $T_{pa}(k)$ and $T_{pc}(k)$ (resp. $T_{nps}(k)$, $T_{npa}(k)$ and $T_{npc}(k)$). In the intree part of an e-outin tree, each e-node corresponds to a path to the exit e-node, while each e-node in the outtree part may have more than one path to the exit e-node. The longest path from an e-node to the exit e-node is selected as the *execution-path* through this e-node. For an e-node, if more than one such longest path exist, then a left-to-right orientation or any tie-breaking rule is used to break the tie. In the outtree part, if an e-node has q immediate successors, one of which is in its execution-path called the *immediate execution-successor*, then the other q−1 immediate successors serve as heads of new execution-paths, called *path-heads*. As a result, each active e-node corresponds to a unique execution-path to the exit e-node.

For example, in Figure 2(b), the execution-path from e-node A_1 is (A_1, C_1, F_1, F_2, F_3, I_1, I_2, J_1, J_2) and e-node B_1 is the head of the execution-path (B_1, D_1, D_2, D_3, H_1, J_1, J_2). Note that when k processors are used, only the topmost k−1 path-heads are active in the splitting phase. Other path-heads are active in the all-busy phase.

Let A(t,k) be the set of active e-nodes at time t when k processors are used. This set can be divided into two classes in terms of the lengths of the corresponding execution-paths. At time t, the active e-nodes whose execution-paths are the shortest among all active e-nodes belong to a subset $A_s(t,k)$, and lie in a single e-level, called the *minimal active e-level*. The other active e-nodes belong to another subset $A_h(t,k)$. (If t and k are obvious in the context, they will be omitted for brevity). For example, in Figure 2(b), when t is 2, e-nodes D_1 and G_1 belong to $A_s(2,2)$, and e-node F_1 belongs to $A_h(2,2)$.

In the following four propositions, we will show the properties of the PCPS algorithm, which are related to the optimal granularity of parallel outin-tree evaluations. It is easy to observe that the active e-nodes are executed at different rates depending on whether the assigned processor is shared or not. Let $r_i(t)$ be the processing rate in e-node per time unit of active e-node i at time t. The following proposition distinguishes the processing rates under various conditions.

Proposition 2.1: During a parallel evaluation of an outin tree with k processors, $r_i(t)$, the processing rate of active e-node i at time t, satisfies the following equations.

$$r_i(t) \begin{cases} = 1 & \text{if e-node } i \in A_h(t,k) \text{ or } |A(t,k)| \leqslant k \\ < 1 & \text{otherwise} \end{cases}$$

where |A(t,k)| is the number of active e-nodes in the set A(t,k).
Proof: This follows from the GS strategy immediately. □

Proposition 2.1 reflects the following facts. First, in the splitting and combining phases, the processing rate for any e-node is one, that is, an e-node is processed in each time unit. Second, if an active e-node is not in the minimal active e-level, then one processor (rather than part of a processor) has to be assigned to it, that is, its corresponding processing rate is one. Third, in the all-busy phase, if the number of active e-nodes is equal to the number of processors, then the processing rates for e-nodes in the minimal active e-level is also one. Only when the number of active e-nodes is larger than the number of processors used, the processing rates of e-nodes in the minimal active e-level are less than one.

Let h_{max} be the length of the critical path in the outin tree to be evaluated. Since at least one time unit is needed to complete each e-node, it is evident for any scheduling algorithm that

$$T(k) \geqslant h_{max} \tag{2.3}$$

where T(k) is the completion time using k processors under a preemptive or nonpreemptive scheduling discipline. The following proposition shows the relationship between $T_p(k)$ and the shape of the phase-boundary.

Proposition 2.2:[**] (a) $T_p(k) > h_{max}$ implies that all active e-nodes on the phase-boundary B_{ac} are located in the same e-level. (b) If an active e-node on the phase-boundary B_{ac} belongs to A_h, then $T_p(k) = h_{max}$.

The example in Figure 2 illustrates Property (a) above. At time t=4, e-node G_1 in the critical path "enters" the set A_s, and hereafter all active e-nodes are in the same e-level. Proposition 2.2 reflects the fact that preemptive scheduling algorithms distribute work uniformly among the available processors, thereby reducing the computational time required in the combining phase. This is the reason for a preemptive algorithm to run faster than the nonpreemptive counterpart.

To investigate the optimal granularity, we need to examine the phase-boundaries when different numbers of processors are used. The following proposition compares two boundaries with respect to k and k+1 processors. In subsequent discussions, the phase-boundary in a *single e-level* will mean that all active e-nodes on this boundary have execution-paths with the same length.

Proposition 2.3:[**] If phase-boundary $B_{ac}(k+1)$ is in a single e-level, then the phase-boundary $B_{ac}(k)$ must be in a single e-level.

If the phase-boundary $B_{ac}(k+1)$ is not in a single e-level, then there are k,' 0<k'<k, active e-nodes belonging to $A_h(k+1)$ on this phase-boundary. During the splitting and all-busy phases, we can partition the (k+1) processors into two groups. The first group consists of k' processors that only evaluate e-nodes in the k' execution-paths from the topmost k' path-heads to the k' e-nodes. Other (k+1−k') processors constitute the second group, which evaluate all e-nodes in the two phases except for e-nodes in the aforementioned k' execution-paths. Accordingly, we can prove the following proposition.

Proposition 2.4:[**] If an outin tree is evaluated by the PCPS algorithm, then $T_{ps}(k+1) \geqslant T_{ps}(k)$, and $T_{pc}(k+1) \geqslant T_{pc}(k)$.

The following theorem shows that the PCPS algorithm can be used to find the optimal preemptive schedule for outin trees.

Theorem 2.1.: PCPS is a minimum-completion-time scheduling algorithm for an outin tree.
Proof: Let $\Phi_{ps}(k)$ and $\Phi_{pc}(k)$ be the total amount of idle times in the splitting and combining phases when the PCPS algorithm is applied and k processors are used. Clearly,

$$T_p(k) = \frac{\Phi_{ps}(k) + T(1) + \Phi_{pc}(k)}{k} \tag{2.4}$$

Minimizing $T_p(k)$ implies minimizing ($\Phi_{ps}+\Phi_{pc}$). In the PCPS algorithm, once an e-node is available, that is, its predecessor node has been finished, a processor is assigned to it immediately. The time spent in the splitting phase for any schedule cannot be shorter than that in the PCPS algorithm. It also means that Φ_{ps} for the PCPS algorithm is the minimum. We now consider Φ_{pc}. If the phase-boundary B_{ac} is not in a single e-level, then $T_p(k)=h_{max}$ according to Proposition 2.2. That is, the PCPS algorithm achieves the minimum computational time T(k) according to Eq. (2.3). What we need to consider is the case when the phase-boundary B_{ac} is in a single e-level. This boundary is indicated by a dark line B in Figure 3.

Suppose that an arbitrary scheduling algorithm is used, the corresponding phase-boundary is denoted by B', which is shown as a dashed line in Figure 3. Note that it is impossible for all e-nodes on boundary B' to be beneath boundary B, otherwise at least one processor is idle before the wave-front achieves B,' which implies that B' is not a phase-boundary. In other words, at least one e-node on boundary B' is above or on boundary B_{ac}. Similarly, it is impossible for all e-nodes of boundary B' to be

[*] We will discuss nonpreemptive CPS algorithms in Section 4.

[**] Due to space limitation, the proofs of Propositions 2.2 thru 2.4, Lemmas 3.1 thru 3.4, and all lemmas and theorems in Section 4 are omitted.

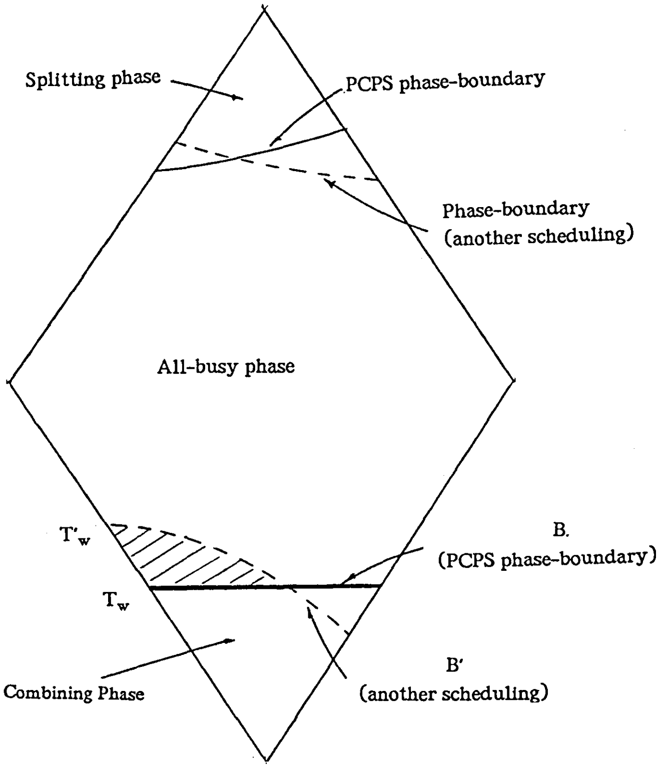


Figure 3. Proof of Theorem 2.1.

above boundary B.

Let N_{pc} and N_c' be the amount of task times in the combining phase of the PCPS and another scheduling algorithm. Let $T_c'(k)$ and $\Phi_c'(k)$ be the computational time and the total idle time in the combining phase when an arbitrary scheduling algorithm using k processors is adopted. If $T_{pc}(k)$ equals $T_c'(k)$, then $N_{pc}(k) \geqslant N_c'(k)$, hence, $(\Phi_c' - \Phi_{pc}) = (N_{pc} - N_c') \geqslant 0$. If $T_{pc}(k)$ is less than $T_c'(k)$, since at least one e-node on boundary B' is beneath or on boundary B_{ac}, $(N_c' - N_{pc})$ cannot be larger than the amount of task times for e-tasks beneath B' and above B (the shaded area in Figure 3). Since, from Proposition 2.1, the processing rate for any path in the combining phase is one, then after $(T_c' - T_{pc})$ time units, all e-nodes in the shaded area in Figure 3 must be completed. As less than k e-nodes can be completed during a time-unit in the combining phase, the amount of e-nodes in the shaded area must be less than $k(T_c' - T_{pc})$. Therefore,

$$(\Phi_c' - \Phi_{pc}) = [k(T_c' - T_{pc}) - (N_c' - N_{pc})] > 0.$$

This means that Φ_{pc}, the total idle time in the combining phase, is also minimum for the PCPS algorithm. The proof does not imply that the PCPS algorithm is the unique optimal algorithm, but rather that the amount of idle times introduced by the PCPS algorithm is the minimum. □

3. OPTIMAL GRANULARITY IN PREEMPTIVE SCHEDULING

The criteria generally used to define the optimal granularity are the processor utilization (PU), kT^2, and AT^2, where k is the number of processors, T is the computational time, and A is the area of a VLSI implementation. The complexity of divide-and-conquer algorithms in an SIMD model and the conditions to assure the optimal processor utilization have been studied [14]. However, processor utilization increases monotonically with decreasing number of processors, which means that PU achieves the maximum when one processor is used. Hence, PU is not an adequate measure for the effects of parallel processing. A more appropriate measure is the kT^2 criterion, which considers both PU and computational time, since

$$kT^2(k) = \frac{T(1)T(k)}{PU} \quad \text{where } PU = \frac{\text{speedup}}{k} \text{ and } T(k) = \frac{T(1)}{\text{speedup}}$$

To minimize kT^2 means to reduce the computational time and to maximize the processor utilization. kT^2 is linearly related to AT^2 if the area of connection wires is proportional to the area of processing elements, as in systolic arrays. Both computational time and processor utilization are important in many applications, hence, kT^2 is a good criterion to optimize. In other applications, such as real-time processing, the completion time may be more critical and the PU is a secondary consideration. In this case, a different optimization criterion may have to be used.

In this paper, we have adopted kT^2 as a criterion of processor-time efficiency to derive the optimal granularity for parallel outin-tree evaluations. That is, given an outin-tree, we need to either choose k to minimize kT^2, or given a fixed k, determine the type of outin trees (their shapes, complexities, etc.) and its proper size that can be solved most efficiently by this system.

It is difficult to find the optimal granularity with respect to kT^2 directly because the optimal granularity depends on the execution time of each task and the shape of the outin tree. We now try to find an efficient and systematic method to determine the optimal grain via an intermediate variable, the total idle time. Let $\Phi_p(k)$ (resp. $\Phi_{np}(k)$) be the total amount of idle times when a preemptive (resp. nonpreemptive) scheduling algorithm with k processors is used. $\Phi_p(k)$ takes into account the idle times in both the splitting and combining phases. Clearly, $\Phi_p(k) = [\Phi_{ps}(k) + \Phi_{pc}(k)]$ and

$$kT(k) = T(1) + \Phi(k) \qquad (3.1)$$

Eq. (3.1) holds for both preemptive and nonpreemptive scheduling algorithms.

The total idle time $\Phi_p(k)$ is related to both k and kT^2. The following two lemmas show the difference between the total idle times when different number of processors are used.

Lemma 3.1:** Suppose that an outin tree is evaluated by the PCPS algorithm, then

$$[\Phi_p(k+1) - \Phi_p(k)] \leqslant h_{max} \qquad (3.2)$$

where h_{max} is the length of the critical path.

Lemma 3.2:** Suppose that an outin tree is evaluated by the PCPS algorithm, then

$$[\Phi_p(k+1) - \Phi_p(k)] \geqslant [T_{ps}(k) + T_{pc}(k)] > 0 \qquad (3.3)$$

Lemma 3.3:** Suppose that an outin tree is evaluated by the PCPS algorithm, then

$$T_p(k+1) \leqslant T_p(k)$$

The above lemmas reveal that when the number of processors used are increased, the total idle times must increase, and the difference of the total idle times with respect to k_1 and k_2 processors is bounded by $(k_2 - k_1)[T_{ps}(k) + T_{pc}(k)]$ and $(k_2 - k_1) \cdot h_{max}$, respectively. From these facts, we can determine the conditions under which kT^2 is either monotonically increasing or decreasing with respect to k. The following theorem shows the relation between $\Phi_p(k)$ and kT^2.

Theorem 3.1: Suppose that an outin tree is evaluated by the PCPS algorithm. $kT_p^2(k)$ is monotonically increasing with k if $[\Phi_p(k+1) - \Phi_p(k)] > T_p(k)/2$. $kT_p^2(k)$ is monotonically decreasing with k if $[\Phi_p(k+1) - \Phi_p(k)] < T_p(k)/(2 + 1/k)$.

Proof: By Eq. (3.1), we get

$$(k+1)T_p^2(k+1) - kT_p^2(k) \qquad (3.4)$$
$$= \frac{[T_p(1) + \Phi_p(k+1)]^2}{k+1} - \frac{[T_p(1) + \Phi_p(k)]^2}{k}$$
$$= \frac{k[T_p(1)+\Phi_p(k+1)]^2 - k[T_p(1)+\Phi_p(k)]^2 - [T_p(1)+\Phi_p(k)]^2}{k(k+1)}$$
$$= \frac{[\Phi_p(k+1)-\Phi_p(k)]\,[2T_p(1)+\Phi_p(k+1)+\Phi_p(k)] - kT_p^2(k)}{k+1}$$
$$= \frac{[\Phi_p(k+1)-\Phi_p(k)]\,[(k+1)T_p(k+1)+kT_p(k)] - kT_p^2(k)}{k+1}$$

Since, from Lemma 3.2, $\Phi_p(k+1) > \Phi_p(k)$, hence

$$(k+1)T_p(k+1) = [T_p(1)+\Phi_p(k+1)] \qquad (3.5)$$
$$> [T_p(1)+\Phi_p(k)] = kT_p(k)$$

From Eq's (3.4) and (3.5), we conclude that if $[\Phi_p(k+1)-\Phi_p(k)] > T_p(k)/2$, then $(k+1)T_p^2(k+1) > kT_p^2(k)$. By Lemma 3.1, Eq's (3.4) and (2.3), we obtain the following condition.

$$\text{If} \quad [\Phi_p(k+1) - \Phi_p(k)] < \frac{T_p(k)}{2 + 1/k},$$
$$\text{then} \quad [(k+1)T_p^2(k+1) - kT_p^2(k)]$$
$$< \left| \frac{kT_p(k)}{2k+1} [2kT_p(k)+T_p(k)] - kT_p^2(k) \right| = 0 \quad \square$$

Theorem 3.1 restricts the region within which we need to find a value k that minimizes $kT_p^2(k)$. In other words, the approximate condition that adding a processor will not degrade the processor-time efficiency is that all processors will be busy at least half of the time.

In the example shown in Figure 2, $T_p(1) = 18$, $T_p(2) = 10.5$, $\Phi_p(2) = 3$, $T_p(3) = 9$, and $\Phi_p(3) = 9$. (Readers are suggested to schedule this outin tree with three processors). Since $\Phi_p(2) = 3 < T_p(1)/3 = 6$, and $[\Phi_p(3) - \Phi_p(2)] = 6 > T_p(2)/2 = 5.25$, according to Theorem 3.1, we can conclude that the use of two processors minimizes kT^2 for this outin tree.

A question about the monotonicity of $[kT_p^2(k+1) - kT_p^2(k)]$ now arises naturally. If $[kT_p^2(k+1) - kT_p^2(k)]$ is increasing monotonically with k, then $kT_p^2(k)$ is a unimodal function of k, and the optimal value of k can be found easily. This monotonicity will be proved in the following theorem.

Theorem 3.2: Suppose that an outin tree is evaluated by the PCPS algorithm, then $kT_p^2(k)$ is a concave function of k, that is, $kT^2(k)$ achieves the minimum when $k = k$,' and $kT^2(k)$ is monotonically decreasing (resp. increasing) with k when $k < k'$ (resp. $k > k'$).

Proof: To show $kT_p^2(k)$ is a concave function of k, we need to prove that its second-order difference is positive, namely, $[(k+2)T_p^2(k+2) - (k+1)T_p^2(k+1)] > [(k+1)T_p^2(k+1) - kT_p^2(k)]$. Let $\Delta(kT_p^2(k)$ denote $[(k+1)T_p^2(k+1) - kT_p^2(k)]$. Then

$$\Delta(kT_p^2(k)) = k[T_p^2(k+1) - T_p^2(k)] + T_p^2(k+1) \qquad (3.6)$$

$$\Delta((k+1)T_p^2(k+1)) = (k+1)[T_p^2(k+2)-T_p^2(k+1)]+T_p^2(k+2) \qquad (3.7)$$

Subtracting Eq. (3.6) from Eq. (3.7) and applying Eq. (3.1) yields

$$\Delta((k+1)T_p^2(k+1)) - \Delta(kT_p^2(k)) \qquad (3.8)$$
$$= (k+2)[T_p^2(k+2) - T_p^2(k+1)] - k[T_p^2(k+1) - T_p^2(k)]$$
$$= [T_p(k+2) + T_p(k+1)]\,[\Phi_p(k+2) - \Phi_p(k+1) - T_p(k+1)]$$
$$- [T_p(k+1) + T_p(k)]\,[\Phi_p(k+1) - \Phi_p(k) - T_p(k+1)]$$

From Eq's (2.3) and (3.8) and Lemmas 3.1, 3.2 and 3.3, we conclude that $\{\Delta((k+1)T_p^2(k+1)) - \Delta(kT_p^2(k))\} > 0$. □

We have found the condition under which kT_p^2 is increased or decreased based on the intermediate variable, $\Phi_p(k)$, and that kT_p^2 is a concave function. Next, we will determine the number of processors such that kT_p^2 is minimum for a given outin tree.

Note that in the original outin tree (see Figure 2(a)), each node is a multiunit task, and tasks in a level may have different lengths. If there are m(i) tasks in level i, then there are m(i) paths from level i to the exit node. Among these paths, the minimum length is denoted by ℓ (i). Similarly, we can define the *depth* of a node as the sum of task-times along a path from the entry node to and including this node, and denote the shortest depth from the entry node to level i by d(i).

Given k processors, we can find c_k, a particular level in the intree part of the original outin tree, such that $m(c_k)$, the number of tasks in this level, is less than k, but $m(c_k+1) \geqslant k$. This particular level is called the *minimum-all-busy level*. Likewise, in the outtree part, there is a level called the *maximum-all-busy level* and denoted by s_k, such that $m(s_k) < k$ and $m(s_k-1) \geqslant k$. By recognizing the minimum-all-busy and maximum-all-busy levels, we can roughly estimate the locations of the phase-boundaries. Recall from Proposition 2.2 that $T_p(k) = h_{max}$ if the phase-boundary B_{ac} is not in a single e-level. In this case, $(k+1)T_p^2(k+1) > kT_p^2(k)$. To achieve the minimum $kT_p^2(k)$, the number of processors should be reduced until the phase-boundary appear in a single e-level. (When $B_{ac}(k)$ is not in a single e-level but $B_{ac}(k-1)$ is, $kT_p^2(k)$ may be minimum.) This observation shows that the use of the minimum-all-busy level to estimate T_{pc} is accurate in most cases. The following lemma shows that the shortest length from the minimum-all-busy (resp. maximum-all-busy) level to the exit (resp. entry) node gives the lower-bound computational time in the combining (resp. splitting) phase.

Lemma 3.4:** Suppose that an outin tree is evaluated by k processors. If s_k and c_k are the maximum-all-busy and minimum-all-busy levels, then (a) $T_{ps}(k) \geqslant d(s_k)$, and (b) $T_{pc}(k) \geqslant \ell(c_k)$. Further, if the phase-boundary $B_{ac}(k)$ lies in a single e-level, then $T_{pc} = \ell(c_k)$.

This lemma is illustrated by the example in Figure 2(a). Suppose that three processors are used, the maximum-all-busy level contains tasks B and C, and the minimum-all-busy level contains tasks H and I. $T_{ps}(3)$ (resp. $T_{pc}(3)$) cannot be less than 2 (resp. 3) because if either task B or C, which are associated with the shortest depth from the entry node to this level, is not finished, then the computation cannot enter the all-busy phase. Likewise, if task H has been assigned to a processor, then the computation must have entered the combining phase. Since $B_{ac}(2)$ lies in a single e-level, $T_{pc}(2) = \ell(c_2) = 3$.

For an arbitrary outin tree,

$$T_{ps} \geqslant t_{en}, \; T_{pc} \geqslant t_{ex}, \text{ and } \Phi_p \geqslant (t_{en} + t_{ex}) \qquad (3.9)$$

where t_{en} and t_{ex} are the task times of the entry and exit nodes, respectively.

When more than one processor are used, some processors must be idle when the entry and exit nodes are evaluated. If the times spent in evaluating the entry and exit nodes dominate over all other computations, then parallel processing is definitely inefficient. The following corollary identifies the condition under which sequential computation is better than parallel processing.

Corollary 3.1: Suppose that an outin tree is scheduled by the PCPS algorithm and $(t_{en}+t_{ex}) > T(1)/2$, then sequential processing, i.e., $k=1$, achieves the minimum kT^2.

Proof: This follows from Theorem 3.1 immediately. □

Having proved a series of propositions, lemmas, and theorems, the main theorem to derive the optimal granularity under the PCPS scheduling algorithm can be obtained now. In the following theorem, the region on k in which we can find the optimal granularity of parallel outin-tree evaluation is given.

Theorem 3.3: Suppose that an AND-tree is evaluated by the PCPS algorithm and $k > 1$, then

$$(k+1)T_p^2(k+1) > kT_p^2(k) \text{ if } k > \frac{2T_p(1)}{h_{max}} \text{ and} \tag{3.10}$$

$$(k+1)T_p^2(k+1) < kT_p^2(k) \text{ if } k < \left\{\frac{T_p(1)+t_{en}+t_{ex}}{2h_{max}} - \frac{1}{2}\right\} \tag{3.11}$$

Proof: From Lemma 3.2,

$$\Delta\Phi_p(k) \geqslant T_{ps}(k) + T_{pc}(k). \tag{3.12}$$

Since the idle time of each processor cannot be larger than $(T_{ps}+T_{pc})$, we have

$$\Phi_p(k) \leqslant (k-1)\,[T_{ps}(k) + T_{pc}(k)] \tag{3.13}$$

By Theorem 3.1, Eq's (3.1), (3.12) and (3.13), the condition that guarantees the monotonic increase of $kT_p^2(k)$ with k is

$$(k+1)T_p^2(k+1) > kT_p^2(k) \tag{3.14}$$

$$\text{if } [T_{ps}(k)+T_{pc}(k)] > \frac{T_p(1) + (k-1)[T_{ps}(k)+T_{pc}(k)]}{2k}.$$

On the other hand, when all tasks but those in the critical path can be completed by (k-1) processors during $h_{max} - [T_{ps}(k)+T_{pc}(k)]$ time-units, i.e.,

$$(k-1) > \frac{T_p(1) - h_{max}}{h_{max} - [T_{ps}(k)+T_{pc}(k)]}, \tag{3.15}$$

the phase-boundary B_{ac} must not be in a single e-level. As discussed before, the optimal grain cannot be larger than the RHS of Eq. (3.15) plus one. By Lemma 3.4, Eq's (3.14) and (3.15),

$$(k+1)T_p^2(k+1) > kT_p^2(k) \text{ if}$$

$$k > \min\left\{\frac{T_p(1)}{d(s_k)+\ell\,(c_k)} - 1,\ \frac{T_p(1)}{h_{max} - [d(s_k)+\ell\,(c_k)]}\right\} \tag{3.16}$$

The condition described in Eq. (3.10) is obtained from Eq. (3.16).

Note that $\Delta\Phi_p(k) \leqslant h_{max}$, and that $kT_p(k) > [T_p(1)+t_{en}+t_{ex}]$ according to Lemma 3.1, Eq's (3.1) and (3.9). By Theorem 3.1, the following result can be derived.

$$(k+1)T_p^2(k+1) < kT_p^2(k) \text{ if } h_{max} < \frac{T_p(1) + t_{en} + t_{ex}}{2k+1}$$

which is equivalent to Eq. (3.11). □

To find optimal granularity, we need to search the small region of k defined by Theorem 3.3. The lower and upper bounds of this region are, respectively, $\{T_p(1)/(2h_{max}) - 1\}$ and $\{2T_p(1)/(h_{max}) + 1\}$. Note that we have not made any assumption about the distribution of task times in deriving these bounds. Since kT_p^2 is a concave function of k (Theorem 3.2), the desirable number of processors can be found easily by a binary search. The binary search can be completed within about $\log_2(T_p(1)/h_{max})$ steps. Each step in the binary search tests whether $\Delta(kT_p^2(k))$ is positive. If it is, then a smaller value of k will be checked in the next step, otherwise, a larger k will be tested.

For any k inside the search region, the phase-boundary $B_{ac}(k)$ is in a single e-level, hence the location of $B_{ac}(k)$ can be uniquely determined without knowing the detailed schedule. Accordingly,

$$\Phi_p(k) = k[T_{ps}(k) + T_{pc}(k)] - [N_{ps}(k) + N_{pc}(k)]$$

where $N_{ps}(k)$ and $N_{pc}(k)$ are the amount of task times in the splitting and combining phases. From Lemma 3.4, $T_{pc}(k) = \ell\,(c_k)$, and $T_{ps}(k)$ can be found directly from the e-outin tree. As a result,

$$kT_p^2(k) = \frac{[T_p(1) + \Phi_p(k)]^2}{k}$$

For instance, suppose that N items need to be sorted. It is well-known that $T(1) = N{\cdot}\log_2 N$ if a merge-sort algorithm is used. In this case, the overhead in the intree part dominates that of the outtree part. For the intree part, $h_{max} = N + N/2 + \cdots + 1 = 2N-1$, so the lower and upper bounds of the search region can be determined from Theorem 3.3, which are close to $(\log_2 N)/4$ and $\log_2 N$, respectively. Since there are only $(3{\cdot}\log_2 N)/4$ candidate values in this search region, $\log_2\log_2 N$ steps of a binary search can guarantee to find the optimal grain of parallel merge sorting. For problems such as evaluating numerical or logic expressions and finding the maximum (or minimum) value, all task times are identical. Theorem 3.3 predicts that the optimal grain is between $N/(2{\cdot}\log_2 N)$ and $2N/\log_2 N$. Figure 4 shows the simulation results of applying a nonpreemptive CPS algorithm to a binary intree of 4096 terminal notes and t_i=1 for all i. Since all tasks have unit execution times, the performance of the nonpreemptive CPS algorithm is very close to that of GS algorithm (see Eq. (2.2)). In this example, kT^2 is minimum when 431 processors are used, which is between $N/(2{\cdot}\log_s N)$ (=170) and $2N/\log_2 N$ (=683).

The above analysis reveals that the optimal grain of an outin-tree evaluation is related to the following parameters:

(a) T(1), the time required by a sequential evaluation, which is the sum of all task times in the outin tree;
(b) h_{max}, the length of the critical path;
(c) $d(s_k)$, the shortest depth from the entry node to s_k, the maximum-all-busy level; and
(d) $\ell\,(c_k)$, the shortest length from c_k, the minimum-all-busy level, to the exit node (recall that c_k and s_k depend on k).

$T(1)/h_{max}$ reflects the shape of the outin tree, while $T(1)/d(s_k)$ and $T(1)/\ell\,(s_k)$ reflect the distribution of the task-times. If the outin tree is "wide" and nearly balanced, i.e., $T_p(1)/h_{max}$ is large, then a fine grain is more appropriate, otherwise, a coarse grain is more suitable. Further, if tasks in levels closer to the entry and exit nodes have longer execution times, i.e., $T_p(1)/[d(s_k)_+\ell\,(c_k)]$ is small, then the optimal grain should be larger, otherwise, a finer grain is better with respect to kT^2. Both $T(1)/h_{max}$ and $T(1)/[d(s_k)+\ell\,(c_k)]$ are related to the problem complexity. We will again show the influence of problem complexity on the

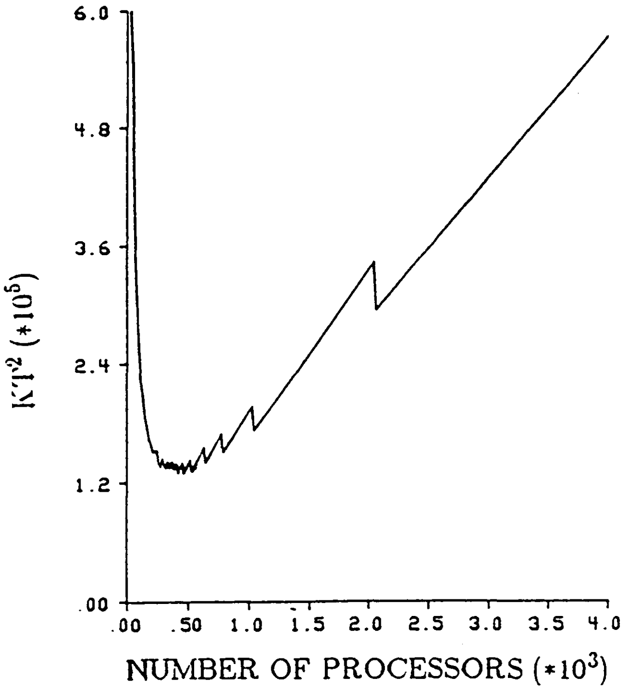


Figure 4. Simulation results to find the optimal granularity of evaluating an intree with 4096 leaves and unit execution times for all nodes.

optimal granularity in the next section, where nonpreemptive algorithms will be used.

4. OPTIMAL GRANULARITY IN NONPREEMPTIVE SCHEDULING

Nonpreemptive CPS algorithms are similar to the PCPS algorithm except that preemption is not allowed. In the nonpreemptive CPS algorithm, one processor is assigned to each of the k nodes farthest from the exit node. If there is a tie in lengths among more than one node, then a left-to-right tie-breaking rule is used to assign a processor to one of these nodes. When a task of the outin tree is completed, the free processor is assigned to the node farthest from the root in the remaining outin tree to be evaluated. Figure 2(e) illustrates an example of nonpreemptive CPS scheduling. In general, nonpreemptive scheduling is more practical due to the smaller task-switching overheads; however, it is more difficult to predict its performance and determine the optimal grain in parallel processing.

The problem of determining the optimal granularity of nonpreemptive CPS algorithm is complicated by its anomalous behavior. Graham has proved that if an AND-tree is evaluated twice by using k_1 and k_2 processors, respectively [11], then

$$\frac{T_{np}(k_1)}{T_{np}(k_2)} \leqslant \left\{ 1 + \frac{k_2 - 1}{k_1} \right\}$$

The above inequality implies that the anomaly $T_{np}(k+1)/T_{np}(k) < k/(k+1)$ is possible. In other words, $kT_{np}^2(k)$ is generally not a concave function of k and cannot be searched by a binary search or other efficient search methods.

In a special case, if the execution times of tasks of an outtree are monotonically decreasing as the tree is decomposed, then it will be shown below that $\Phi_{np}(k_2) > \Phi_{np}(k_1)$ holds for $k_2 > 2k_1$. Likewise, the same relation holds for the case when the execution times of tasks of an intree are monotonically increasing as the tree is composed. Here, the optimal granularity of outin-tree computations based on a nonpreemptive CPS algorithm can be bounded in a relatively small region. The assumption on monotonic distribution of task times is valid in divide-and-conquer algorithms.

In this section, we will develop conditions under which kT^2 is monotonically increasing or decreasing with k for the special case in which the task times are monotonically decreasing in the outtree and monotonically increasing in the intree. We will investigate the difference of the total idle times with respect to different number of processors under nonpreemptive CPS. The following lemma gives the lower and upper bounds of $[\Phi_{np}(k_2) - \Phi_{np}(k_1)]$.

Lemma 4.1:** Suppose that an outin tree is scheduled by a nonpreemptive CPS algorithm and that $t_i > t_j$ if task i is a predecessor (resp. successor) of task j in the outtree (resp. intree) part, then

$$[\Phi_{np}(k_2) - \Phi_{np}(k_1)] \geqslant \{(k_2 - k_1)[T_{ps}(k_1) + T_{pc}(k_1)] - k_1 t_{npa}(k_1)\} > 0 \quad \text{if } k_2 > 2k_1 \qquad (4.1)$$

$$[\Phi_{np}(k_2) - \Phi_{np}(k_2)] \leqslant \{(k_2 - k_1)[T_{ps}(k_2) + T_{pc}(k_2)] + k_2 t_{npa}(k_2)\} \quad \text{if } k_2 > k_1 \qquad (4.2)$$

where $t_{npa}(k)$ is the longest task-time among all tasks in the all-busy phase when k processors are used.

We should point out that the above lemma holds for the case in which a part of the phase-boundary is in the intree and another part is in the outtree. The above lemma is true because the task time of a node in the all-busy phase is less than either T_{ps} or T_{pc} from the assumption of monotonically distributed task times. That is, $[T_{ps}(k_1) + T_{pc}(k_1)] > t_{npa}(k_1)$ is always true regardless the location of the phase-boundary.

Similar to Theorem 3.1, we first study the relationship between kT^2 and the idle times. The following theorem gives the conditions under which kT^2 is monotonically increasing or decreasing based on the intermediate variable $\Phi_{np}(k)$.

Theorem 4.1:** Suppose that an outin tree is scheduled by a nonpreemptive CPS algorithm and that $t_i > t_j$ if task i is a predecessor (resp. successor) of task j in the outtree (resp. intree) part, then

$$k_2 T_{np}^2(k_2) > k_1 T_{np}^2(k_1) \text{ if } [\Phi_{np}(k_2) - \Phi_{np}(k_1)] > \left| \frac{k_2 - k_1}{2} T_{np}(k_1) \right| \text{ and } k_2 > 2k_1; \qquad (4.7)$$

$$k_2 T_{np}^2(k_2) < k_1 T_{np}^2(k_1) \text{ if } [\Phi_{np}(k_2) - \Phi_{np}(k_1)] < \left| \frac{2(k_2 - k_1)k_1}{2k_1 + 3k_2} T_{np}(k_1) \right| \text{ and } k_2 > k_1 \qquad (4.8)$$

The main theorem to find the optimal granularity can be derived from Theorem 4.1. Before this theorem is proved, the following lemma is needed.

Lemma 4.2:** For a given outin tree, suppose that both PCPS and nonpreemptive CPS algorithms are applied, then $[T_{nps}(k) + T_{npc}(k)] \leqslant [T_{ps}(k) + T_{pc}(k) + t_{npa}(k)]$.

The example in Figure 2 illustrates this lemma. Here, $[(T_{nps} + T_{npc}) - (T_{ps} + T_{pc})] = 1$, which is less than t_{npa} (=3).

Theorem 4.2:** Suppose that an outin tree is scheduled by a nonpreemptive CPS algorithm and that $t_i > t_j$ if task i is a predecessor (resp. successor) of task j in the outtree (resp. intree) part, then k, the number of processors that minimizes $kT_{np}^2(k)$, is bounded between $[T_{np}(1) + t_{en} + t_{ex}]/(8h_{max})$ and $3T_{np}(1) /[d(s_k) + \ell(c_k) - 2t_{npa}(k)]$

As an example, we can determine the area within which the optimal granularity can be found for the parallel merge-sort of N elements. In this problem, the computational overhead in the intree is dominant, so only the part of the intree has to be considered in the scheduling. From Theorem 4.2, the lower bound of the search region is $(\log_2 N)/16$, since $T_{np}(1) = N \cdot \log_2 N$ and $h_{max} < 2N$. If N is large enough, then $[d(s_k) + \ell(c_k) - 2t_{npa}(k)]$ will be larger than 1.5N, hence, the upper bound of the search region is $2 \cdot \log_2 N$.

Comparing these bounds with Theorem 3.3, we see that the range within which an optimal-grain for a nonpreemptive schedule can be found is larger than that of a preemptive schedule. Moreover, kT^2 is not monotonically decreasing or increasing with k for nonpreemptive scheduling, i.e., kT^2 is not a unimodal function of k, hence, an exhaustive search is required to find the optimal grain.

To predict the optimal order-of-magnitude granularity in general, we now briefly discuss the asymptotically optimal granularity of parallel outin-tree evaluations with nonpreemptive CPS algorithm. Let C(n) be the overhead of a node in the intree, which has n leaves rooted by this node. C(n) represents the overhead of combining the results from its immediate predecessor nodes in the intree. Likewise, let D(n) be the overhead of a node in the outtree, which has n leaves rooted by this node. D(n) represents the overhead of decomposing the given node into its immediate successor nodes in the outtree. For an outin tree with N leaves, C(N) and D(N) represent the overheads of the exit and entry nodes, respectively. Let Θ be the set of functions of the same order. For problems such as summing a set of numbers, finding the maximum of N numbers, and returning logical values to the main goal in evaluating logic programs, $C(n) = \Theta(1)$. In quicksort and merge sort, $C(n) + D(n) = \Theta(n)$.

The asymptotically optimal grain depends on the complexities of C(n) and D(n). The higher the order-of-magnitude complexity of C(n) and D(n) are, the larger the granularity is. When the order-of magnitude complexity of C(n) (and/or D(n)) is large, the time spent in the combining (and/or splitting) phase is dominating the time in the all-busy phase, and the performance gain in the all-busy phase with finer grains is negligible. In other words, a small granularity may result in underutilization of processors.

To isolate the impact of the complexities C(n) and D(n) on the optimal granularity from the shape of the outin tree, we discuss the complete binary outin tree, and assume that, for all nodes in a level of the intree (resp. outtree) part, the order-of-magnitude complexities of C(n) (resp. D(n)) are identical. This assumption enables us to estimate T(1). The following theorem gives the condition under which the asymptotically minimum kT^2 is achieved for various complexities of C(n) and D(n).

Theorem 4.3:** Suppose that a nonpreemptive CPS algorithm is applied to evaluate an outin tree of N leaves by k processors. Assume that, for all nodes in a level of outin tree, the order-of-magnitude complexities of C(n) (and D(n)) are the same and that $t_i > t_j$, if task i is a predecessor (resp. successor) of task j in the part of the outtree (resp. intree). Then the order-of-magnitude $kT_k^2(N)$ is the minimum if $\Theta(T_{npa}(k(N))) = \Theta(T_{nps}(k(N))+T_{npc}(k(N)))$.

The above theorem shows that if the number of leaves of an outin tree is very large, then, to achieve the minimum $kT_p^2(k)$, the number of processors should be chosen such that the times required by the all-busy phase and the total times required by the other two phases are approximately equal. This result also shows the relationship between the processor utilization and kT^2. Let N_{sc} be the amount of task-time in the splitting and combining phases, and $T_{npa}(k) = [T_{nps}(k)+T_{npc}(k)]$. Then, for arbitrary outin tree computations, an asymptotically optimal granularity is achieved when

$$PU(k) = \frac{kT_{npa}(k) + N_{sc}}{2kT_{npa}(k)}$$

Since $T_{npa}(k) \leqslant N_{sc} \leqslant (k-1)T_{npa}(k)$, we conclude that the corresponding processor utilization is between 0.5 and 1. In other words, when a problem is solved by a parallel divide-and-conquer algorithm and there are a large number of leaves in its outin-tree representation, to pursue more than 50% processor utilization will reduce the utilization-time efficiency. According to Theorem 4.3, the asymptotically optimal granularities with respect to various C(n)+D(n) are summarized in Table 1.

Complexity of C(n)+D(n) $1 \leqslant n \leqslant N$	Optimal Granularity	Architectural Requirements
$\Theta(\log_2^s n)$ $s \geqslant 0$	$\Theta\left(\frac{N}{\log_2^{s+1} N}\right)$	A very large number of processors; tree or other efficient interconnection
$\Theta(n^r \log_2^s n)$ $0 < r < 1$ $s \geqslant 0$	$\Theta\left(\frac{N^{1-r}}{\log_2^s N}\right)$	A large number of processors; tree or other efficient interconnection
$\Theta(n \log_2^s n)$ $s \geqslant 0$	$\Theta(\log_2 N)$	A small number of processors; loosely coupled; simple interconnection
$\Theta(n^p)$ $p > 1$	$\Theta(1)$	Single or few processors; shared memory

Table 1. Asymptotically optimal granularities in parallel processing of outin trees with respect to order-of-magnitude kT^2 (N is the number of leaves of the outin tree).

5. CONCLUDING REMARKS

In this paper, we have derived tight bounds within which the optimal granularity of parallel AND-tree evaluations under preemptive and nonpreemptive critical path scheduling can be found. For nonpreemptive scheduling, the asymptotically optimal granularities with respect to various problem complexities have been derived. These theoretical results provide an upper bound on the number of processors to achieve the minimum kT^2 criterion.

According to our efficiency analysis, we found that the optimal granularity depends on the problem complexity, the shape of the precedence graph (balanced or skewed), and the task-time distribution along each path (random or monotonic). It is usually difficult to predict the shape and the task-time distribution. One possible way is by statistical analysis. In contrast, the complexity of a problem to be solved is generally known before the problem is solved.

The complexity of each node in the outin tree is an important factor that influences the optimal granularity. As illustrated in Table 1, if C(n)+D(n) is $\Theta(n^p)$, $p>1$, and a large number of processors are used, then the processor-time efficiency, kT^2, must be poor regardless of the capacity of the interconnection network. In this case, the time needed to evaluate a subproblem will be increased quickly during the decomposition process in the outtree and the composition process in the intree. Hence, the root and exit nodes of the tree are obvious bottlenecks. In contrast, if C(n) and D(n) are $\Theta(1)$, then the time needed to evaluate any subproblem is bounded by a constant, and the root and exit nodes will not be bottlenecks. Examples of this kind of problems include finding the maximum and evaluating an arithmetic expression. Here, a fine-grain architecture is appropriate, and a large speedup will be obtained by using a large number of processors. Tree-structured computer architectures [13, 21] and virtual-tree computers [2] are good candidates in these applications. In cases when C(n) equals either $\Theta(n)$ or $\Theta(\log^s n)$, $s \geqslant 0$, the time needed to evaluate a subproblem is increased slowly during the decomposition process in the outtree and the composition process in the intree. A medium-grain architecture will be more cost-effective. For example, to sort 4000 elements by a parallel merge-sort algorithm, using ten to twelve processors will be a good choice. For many practical problems, especially when divide-and-conquer algorithms are used, the precedence graph is nearly balanced, and the task-times of all nodes in each level are approximately equal, hence, the nonpreemptive critical path scheduling algorithm may be viewed as a parallel breadth-first search. In this case, well balanced workloads with overlapped process communications can be assigned to the processors working under an SIMD model. The optimal grains will, therefore, be close to the theoretical ones predicted in Sections Three and Four. The architectural requirements for various cases are summarized in Table 1.

In many problems, the order-of-magnitude complexities of C(n) and D(n) may be different. For example, for the quicksort algorithm, $C(n)=\Theta(1)$ and $D(n)=\Theta(n)$, that is, most of the computational overhead is spent in the decomposition phase, and the composition operation is trivial. In contrast, for the merge-sort algorithm, $C(n)=\Theta(n)$ and $D(n)=\Theta(1)$. In many logical and functional programs, the return operation is usually simple, i.e., $C(n)=\Theta(1)$, but the complexity of a subgoal or function call depends on the number of the parameters passed and the method of copying data. For these problems, the optimal grain can be determined by the part of the tree that has dominant overhead.

The shape of the AND-tree and its task-time distribution are also important factors to be considered. Let $T_p(1)/h_{max}$ be "average width" of an AND-tree. The optimal granularity is found to depend strongly on the average width. If the AND-tree is "wide," then the degree of parallelism is high and the granularity can be small. On the other hand, if the AND-tree is "narrow," then the degree of parallelism is low and the granularity is necessarily large. Here, the tree may have to be restructured to arrive at a different representation.

In many problems involving AND-trees, the trees are usually irregular, and the workloads may be data dependent. An important functional requirement is, therefore, the ability to dynamically distribute the workload in the architecture. For a computer architecture with a small granularity, an efficient interconnection network is needed. In a loosely coupled system with a coarse grain, an effective load balancing mechanism is necessary. Here, process communications may not be well overlapped with computations, and the corresponding task-times should include the communication overhead. As a result, the optimal number of processors may be less than the theoretical values predicted in Sections Three and Four.

REFERENCES

[1] J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Algebra of Programs," *Comm. of the ACM*, vol. 21, no. 8, pp. 613-641, ACM, 1978.

[2] F. M. Burton and M. M. Huntbach, "Virtual Tree Machines," *IEEE Trans. on Computers*, vol. C-33, no. 3, pp. 278-280, 1984.

[3] E. G. Coffman, Jr. and R. H. Graham, "Optimal Scheduling for Two Processors Systems," *Acta Informatica*, vol. 1, no. 3, pp. 200-213, 1972.

[4] E. G. Coffman, Jr. (ed.), *Computer and Job-Shop Scheduling Theory*, Wiley, New York, NY, 1976.

[5] J. S. Conery and D. F. Kibler, "AND Parallelism and Nondeterminism in Logic Programs," *New Generation Computing*, vol. 3, no. 1, pp. 43-70, OHMSHA Ltd. and Springer-Verlag, 1985.

[6] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.

[7] J. B. Dennis, "Data Flow Supercomputers," *Computer*, vol. 13, no. 11, pp. 48-56, IEEE, Nov. 1980.

[8] D. Dolev and M. Warmuth, "Profile Scheduling of Opposing Forests and Level Orders," *SIAM Journal of Algorithm and Discrete Mathematics*, vol. 6, no. 4, pp. 665-687, Oct. 1985.

[9] M. R. Garey and D. S. Johnson, "Scheduling Tasks with Nonuniform Deadlines on Two Processors," *Journal of ACM*, vol. 23, no. 3, pp. 461-467, 1976.

[10] M. R. Garey, D. S. Johnson, R. E. Tarjan, and M. Yannakakis, "Scheduling Opposing Forests," *SIAM Journal of Algorithm and Discrete Mathematics*, vol. 4, no. 1, pp. 72-93, March 1983.

[11] R. L. Graham, "Bounds for Certain Multiprocessing Anomalies," *The Bell System Technical Journal*, vol. 45, no. 9, pp. 1563-1581, Nov. 1966.

[12] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. N. Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Ann. Discrete Mathematics*, vol. 5, pp. 287-326, 1979.

[13] J. A. Harris and D. R. Smith, "Simulation Experiments of a Tree Organized Multicomputer," *Proc. 6th Annual Symp. on Computer Architecture*, pp. 83-89, IEEE/ACM, April 1979.

[14] E. Horowitz and A. Zorat, "Divide-and-Conquer for Parallel Processing," *Trans. on Computers*, vol. C-32, no. 6, pp. 582-585, IEEE, June 1983.

[15] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, pp. 841-848, 1961.

[16] M. Kaufman, "An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem," *Trans. on Computers*, vol. C-23, no. 11, pp. 1169-1174, IEEE, 1974.

[17] R. M. Keller, F. C. H. Lin, and J. Tanaka, "Rediflow Multiprocessing," *Proc. COMPCON Spring*, pp. 410-417, IEEE, 1984.

[18] M. Kunde, "Nonpreemptive LP-Scheduling on Homogeneous Multiprocessor Systems," *SIAM Journal of Computing*, vol. 10, no. 1, pp. 151-173, Feb. 1981.

[19] J. K. Lenstra, A. R. Kan, and P. Brucker, "Complexity of Machine Scheduling Problems," *Proc. Discrete Mathematics*, pp. 343-362, North-Holland, 1977.

[20] E. L. Lloyd, "Critical Path Scheduling with Resource and Processor Constrains," *Journal of ACM*, vol. 29, no. 3, pp. 781-811, 1982.

[21] G. Mago, "Making Parallel Computation Simple: The FFP Machine," *Proc. COMPCON Spring*, pp. 424-428, IEEE, 1985.

[22] R. Muntz and E. Coffman, Jr., "Preemptive Scheduling of Real-Time Task on Multiprocessor Systems," *Journal of the ACM*, vol. 17, no. 2, pp. 324-338, ACM, April 1970.

[23] F. J. Peters, "Tree Machine and Divide-and-Conquer Algorithms," *Lecture Notes CS 111 (CONPAR81)*, pp. 25-35, Springer-Verlag, 1981.

[24] S. Schindler, "On Optimal Scheduling for Multiprocessor Systems," *Proc. of Princeton Conf. on Information Science and Systems*, pp. 219-223, 1972.

[25] J. D. Ullman, "NP-Complete Scheduling Problems," *Journal of Computer and System Sciences*, vol. 10, pp. 384-393, 1975.

PARALLEL PREPROCESSING AND POSTPROCESSING IN FINITE-ELEMENT ANALYSIS ON A MULTIPROCESSOR COMPUTER[1]

P. S. TSENG
Dept. of Electrical
and Computer Engineering
Carnegie-Mellon University
Pittsburgh, PA 15213

Kai HWANG
Computer Research Institute
Univ. of Southern California
Los Angeles, CA 90089-0781

Abstract

Finite-element analysis requires extensive computations in the preprocessing and postprocessing stages, which could become the real bottleneck. This paper presents a multiprocessing and macropipelining approach to overcome this difficulty. We demonstrate through timing analysis that using a linear array of processors in a macropipelined fashion, one can achieve a significant throughput improvement in preprocessing the data elements and outputing the results after a finite-element analysis. We provide analytical methods for matching the processing bandwidths among the host computer, the preprocessor, the linear system solver, and the postprocessor. The time savings in these preprocessing and postprocessing steps will greatly improve the overall system performance.

1. Introduction

The Finite Element Method (FEM) for solving Partial Differential Equations (PDE) consists of three phases : the input of data elements, the solution of a linear system of equations, and the output of analytical results[1]. In the past, most efforts have concentrated in the middle phase of solving a linear system of equations Kx = f, where K is an nxn stiffness matrix obtained by discretizing the given PDEs. Very little has been done in the speedup of the input and output phases.

Finite-element analysts often refer to the input phase as *preprocessing* and the output phase as *postprocessing*. Besides I/O of elemental data and results, computations are also needed in these two phases in order to produce the stiffness matrix and to display graphically the analytical results obtained in the middle phase. In this article, we propose a pipelined approach for speeding up the extensive computations involved in the preprocessing and postprocessing phases.

Finite-element discretization often yields a linear system of equations with $n = 10^5$ or more unknowns in structural engineering. In most cases, the stiffness matrices obtained are highly sparse and symmetric positive definite. Sequential algorithms such as adaptive *Successive Over Relaxation* (SOR), preconditioned *Conjugate Gradient* (CG) methods[2] have been developed for solving sparse linear systems on conventional uniprocessor computers. These algorithms can produce the solution in $O(n^{1.5})$ time, contributed by $O(n^{0.5})$ iterations with O(n) time required for each iteration[3].

Pipelined vector processors have been used in supporting computations of this type[4, 5]. Solving a linear system with $n=10^6$ unknowns can now be done in minutes using a Cray X-MP or a Floating-Point Systems attached processor[5, 6]. However, these sequential algorithms do not fully exploit parallelism contained in all three phases.

The parallel algorithm obtained by Pan and Reif[7] decomposes a stiffness matrix into recursively nested dissections, which requires $O(\log^3 n)$ time and $O(n^{1.5})$ processors. After the decomposition, $O(\log^2 n)$ time and O(n) processors are needed to solve the linear system of equations. In order to match the high speed of parallel computations performed in the middle phase, special hardware is needed to speedup the preprocessing and postprocessing phases. This will be crucial to the overall system performance.

In this paper, we propose a method to speedup the pre/postprocessing steps. We present the limitations in speeding up the pre/postprocessing using a data-transfer model. A linear array of processors is used to yield optimal performance under the revealed limitations on I/O bandwidths.

2. The System Architecture

A finite-element machine consists of five major components: the host computer, the preprocessor, the interface unit, the parallel linear system solver, and the postprocessor as shown in Fig. 1. The host computer generates finite elements by applying a discretization algorithm to the space domain. The preprocessor

[1] The research was supported in part by NSF grant DMC-84-21022 and in part by AFOSR grant 86-0008

generates elemental stiffness matrices and corresponding force vectors. The interface unit assembles elemental stiffness matrices and force vectors to yield the global stiffness matrix K and force vector f.

The linear system solver solves the system Kx = f for the unknown vector x. The results are then sent back to the interface unit, which associates finite elements with their solved nodal values and sends the combined terms to the postprocessor. The postprocessor interpolates the final results and sends them back to the host computer. The host computer then displays the final solution to the user.

The use of the host computer, the interface unit, and the highly parallel linear system solver have been similarly practiced in the MPP[8], the connection machine[9], and the finite-element machine[10]. What we propose here is an architectural setup for faster preprocessing and postprocessing. The intention is to match the processing bandwidth of all the component subsystems involved in finite-element analysis. The parallel system solver demands the use of a large memory space to accommodate the huge stiffness matrix involved. The interface unit demands also a large memory space to accommodate the discretized space domains and their relations to the stiffness matrix. The pre/postprocessors are designed as functional units with a small working memory.

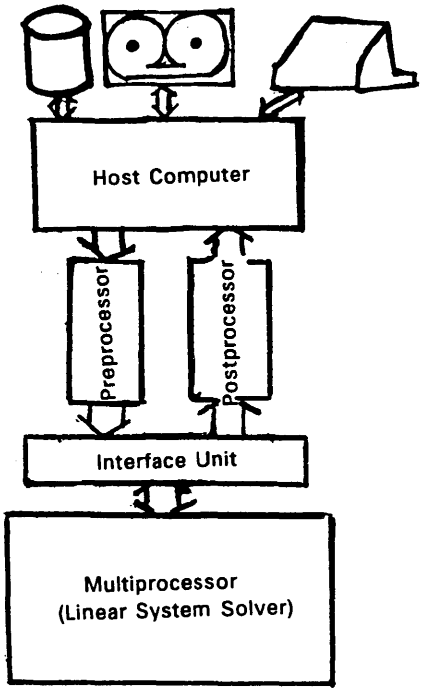


Figure 1: The system architecture of a parallel FEM-PDE solver

3. Preprocessing and Postprocessing in the FEM

Pre/postprocessings are essential in supporting real-time input of data elements and the generation of 3-D graphic displays. Most simulation studies in structural analysis, flight control, oil reservoir modeling, biological and seismological experiments demand a heavy degree of pre/processing. We define an *elemental computation* as follows : Consider an element E_i in the discretized domain, Let X be the input set , Y be the output set and F be a function independent of the index i. Then we have Y = F(X) associated with element E_i.

In preprocessing, the set X is generated from the host computer, and the set Y is sent to the interface unit. In postprocessing, X is generated from the interface unit, and Y is destined to the host computer. These elemental computations are performed repeatedly for the generation of elemental stiffness matrices and force vectors , the derivation of dependent variables, graphical curve interpolations, and geometrical transformations. All of them are main operations contained in the pre/postprocessing phases.

There is no data dependence between different elemental computations This implies the existence of massive parallelism in these elemental computations. However, the limited data transfer bandwidth between system components constrains this parallelism. We define several parameters below which will be used in the architectural designs of the preprocessor and the postprocessor. :

T_i The average time for transferring one elemental data set from the host computer to the preprocessor.

T_o The average time for transferring one elemental data set from the the postprocessor to the host computer.

T_c The preprocessing time of one element.

T_c' The postprocessing time of one element.

N_e The number of elements in a discretized PDE problem.

P The number of processors used in the preprocessor.

P' The number of processors used in the postprocessor.

Let T be the total time required for a preprocessing procedure and T' be that for a postprocessing procedure. Then we can prove that:

$$T > N_e T_i \quad \text{and} \quad T' > N_e T_o \qquad (1)$$

where, N_eT_i is the minimum time required to transfer all data from the host computer to the preprocessor. Since the overall processing time must be larger than the time for loading the data from the host computer, the inequality is thus obvious. Similar arguments hold for the postprocessing. Furthermore, one can easily show that:

$$T > (N_eT_c)/P \quad \text{and} \quad T' > (N_eT_c')/P', \qquad (2)$$

where $(N_eT_c)/P$ is the minimum time required if P processors are used in preprocessing. P' is the number of processors used in postprocessing.

For the best performance, the above results lead to the use of $P = N_eT_c/T$ and $P' = N_eT_c'/T'$ respectively. This leads to the following conditions:

$$T = (N_eT_c)/P \quad \text{if} \quad T_c > T_iP$$

$$T' = (N_eT_c')/P' \quad \text{if} \quad T_c' > T_oP' \qquad (3)$$

These findings reveal that the maximum speedups obtainable in pre/postprocessing are T_c/T_i and T_c'/T_o respectively. If the speedup factor does not match with the speed of the linear system solver, multiple lines of pre/postprocessors may be needed to support parallel data streams flowing between the host and interface unit. We concentrate on a single stream of pre/postprocessors in this study.

4. Linear Array and Macropipelining

A simple linear array of processors is proposed in Fig.2 for either preprocessing or postprocessing. Linearly connected processors requires minimum communication links between adjacent processors. The linear array can be designed to be modularly extensible, easily synchronizable[11], and fault tolerant[12]. In order to avoid the use of a large amount of data memory in the array, systolic processing is preferable. Most computations for FEM pre/postprocessing are complex, they may not map nicely with a linear systolic array. For this reason, we propose a *macropipelining* approach to achieve a pseudo systolic data flow. Macropipelining refers to the practice of pipelining in the processor level, as originally suggested in Händler[13] and Briggs, et al[14].

In macropipelining, the data context of each element are processed on the fly similarly to that of a systolic array. Once a processor receives the data context of an element, it continues the computation of the element for T_c/P (or T_c'/P') time and passes the partial results to the next processor. The processors in the linear array pass the results to the interface unit in preprocessing stage or to the host computer in postprocessing stage. The purpose is to average the load of all processors involved and to achieve an optimal pipeline rate.

The original sequential program is repeated in each processor, no recoding is needed. Only a hardware timer is required in each processor for generating interrupt signals every T_c/P time to pass the elemental context between processors. Only a small amount of memory is used in each processor to accommodate the context of three elements, one being received, one being processed, and the third one being sent out.

The pipelining of multiple processors into a macropipeline demands a special processor architecture as suggested in Fig.3. The *dual port* memory uses one port for data transfer between processors, and the other for data processing within the processor. The numerical processing unit performs floating point arithmetic. The input/output of data are done through parallel data channels. The input/output link processors contain data buffers to interface communicating processors in the array.

Besides parallel data links between processors, there are links connecting the control units of each processor. These interprocessor control links are used to synchronize the computations and communications along the linear array. The control unit sequences the local operations in each processor and communicates with the neighboring processors. The timer is designed to support macropipelining, which generates interrupt signals to the control unit and initiates the context switching between processors.

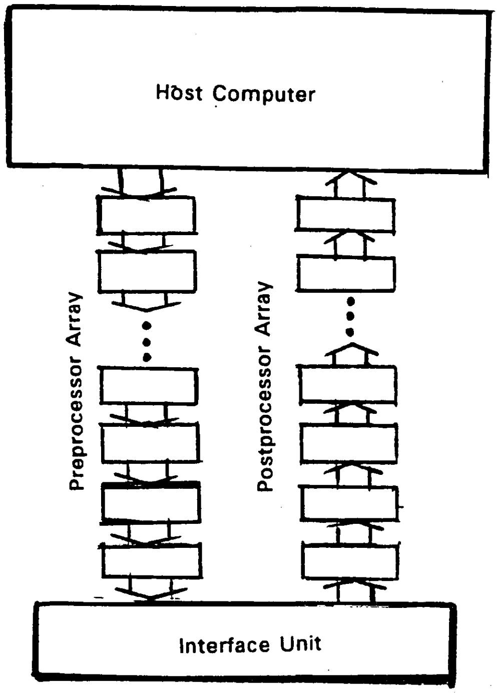


Figure 2: Suggested processor architecture for the construction of I/O processing units

Macropipelining is limited by excessive context size of each element. A large context size may create a data transfer bottleneck in the array. This interprocessor data transfer rate may substantially limit the throughput of the linear array. Several parameters thus are crucial to the performance of the preprocessor array are defined below.

t_1 The average time for transferring one byte through the data links between adjacent processors in the linear array.

n_c The size of the context of an element, measured by bytes.

t_i The average time for transferring one byte through the data channel between host computer and the preprocessor.

n_i The size of input data set of each element, measured by bytes. Note that $T_i = t_in_i$.

Using the above parameters, we found the following conditions for optimal design of the preprocessor array:

$$T_c > Pn_ct_1 \qquad (4)$$

This condition implies that T_c/P delay is needed in processing each element in a processor, where n_ct_1 is the time required to transfer the context of each element between adjacent processors in the macropipeline. Similarly, to yield optimal performance in the postprocessor array, we need to satisfy:

$$n_it_i/n_ct_1 > 1 \qquad (5)$$

This condition leads to the maximum speedup T_c/T_i, where $T_i = n_it_i$ and $T_c > (T_c/T_i)n_ct_1$.

We define a *software ratio* $\alpha = n_i/n_c$ between the size of input data set and the size of context per element, and a *hardware ratio* $\beta = (t_i/t_1)$ between interprocessor link bandwidth and the host-preprocessor channel bandwidth. These two ratios are crucial to the performance of the two arrays used. In most cases, $\alpha < 1$, since the context must contain the input data set. The value of α provides a guideline for choosing the hardware parameter β. In practice, we prefer the use of smaller value of β, say $\beta < 8$.

The condition $\alpha\beta > 1$ is necessary in applying macropipelining in the preprocessor array to achieve maximum speedup. However, if the number of processors available in the system is less than T_c/T_i, we have a loose condition for obtaining optimal performance from macropipelining as specified above. The performance of the preprocessing macropipeline is analyzed below under two assumptions, $T_c > PT_i$ and $T_c > Pn_ct_1$. The total processing time of the preprocessing array is:

$$T = T_c + Pn_ct_1 + (T_c/P)N_e < 2T_c + (T_cN_e)/P, \qquad (6),$$

where $T_c + Pn_ct_1$ is the time for the interface unit to receive the first output, it is the pipeline startup time. The throughput of the liner pipeline is T_c/P. As a result, it takes $(T_c/P)N_e$ to process the last elements. The total processing time is thus $T_c + Pn_ct_1 + (T_c/P)N_e$. The condition $T_c > Pn_ct_1$ leads to the inequality in Eq. 6.

The above result shows that macropipelining often results in a nearly optimal performance with small start up overhead. Once a solution vector is produced by the linear system solver, it needs to be displayed graphically. To generate graphical primitives from the discretized solutions, one need to perform geographical transformations, perspective transformations, and polynomial interpolations. The outputs from the postprocessor array are mainly the graphic primitives needed by the host computer for display purpose.

Macropipelining can be similarly used in the postprocessor array, To measure the performance of the postprocessor array, the following parameters are used:

t_o the average time of transferring one byte between the postprocessor and the host computer.

n_o the size of the output data set of each element transferred to the host computer, measured by bytes.
Note that, $T_o=t_on_o$.

n_c' the size of the data context of an element in postprocessing, measured by bytes.

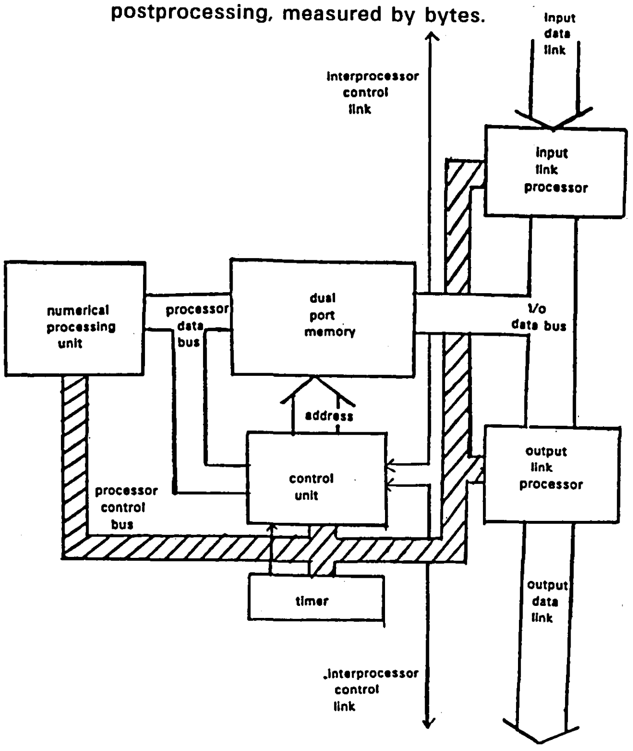


Figure 3: Proposed architecture of the processors used in the preprocessing and postprocessing arrays

α' the ratio of n_o/n_c.

β' the ratio of t_o/t_1.

In case $T_c' > T_oP'$ and $T_c' > Pn_c't_1$, the total postprocessing time $T' < 2T_c' + (N_eT_c')/P'$ can be similarly derived.

The use of a linear array of processors can achieve a nearly optimal speedup in FEM pre/postprocessing. However, the conditions for achieving this performance, $T_c > Pn_ct_1$ ($T_c' > P'n_c't_1$) and $\alpha\beta > 1$ ($\alpha'\beta' > 1$) should be carefully examined in real applications. The gains from exploiting overlapped parallelism in these I/O operations must be balanced by the increased cost of the pre/processors arrays.

5. Parallel Elemental Computations

Examples of FEM pre/postprocessing are given below. We consider first the computational demand in FEM preprocessing, namely for the generation of elemental stiffness matrices and elemental force vectors. To simplify the explanation, consider a 2-D 4-node quadrilateral Lagrange element for a poisson equation :

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{7}$$

The elemental stiffness matrix $K = [k_{ij}]$ is an 4x4 matrix defined by the following equation :

$$k_{ij} = \int\int [\frac{\partial \psi_i}{\partial x}, \frac{\partial \psi_i}{\partial y}] [\frac{\partial \psi_j}{\partial x}, \frac{\partial \psi_j}{\partial y}]^T \ dxdy \tag{8}$$

where ψ_i's are the basic shape functions associated with each node. These shape functions are predefined from a standard square element on a *natural coordinate system* (ξ, η). The standard element and a real element are linked by an isoparametric transformation between the two coordinate systems (ξ, η) and (x, y) as illustrated in Fig.4 . To perform the integration, Gauss quadrature is performed as follows:

$$K = \sum_{k=1}^{2} \sum_{l=1}^{2} B(\xi_k,\eta_l)^T B(\xi_k,\eta_l) \det(J(\xi_k,\eta_l)) W_{kl} \tag{9}$$

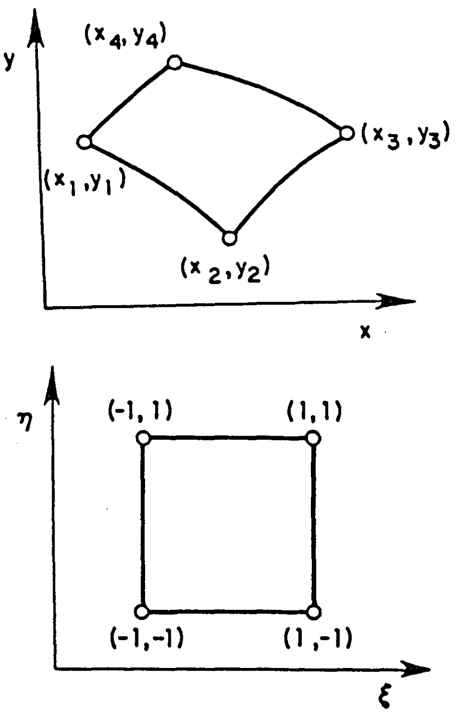
where (ξ_k, η_l) are the Gauss points inside an element. In the above example 2 x 2 Gauss points are used. J is a 2x2 Jacobian matrix on the Gauss point, which depends on 4 nodal coordinates of real elements. B is a 2x4 matrix defined on each Gauss points by the following equation :

$$B=J^{-1}N, \quad N_{2x4} = [n_{ij}] \tag{10}$$

$$n_{1j}=\frac{\partial \psi_j}{\partial \xi} \qquad n_{2j}=\frac{\partial \psi_j}{\partial \eta}$$

N is a precomputed constant matrix for the standard square element. The computations involved are elemental. The inputs are the nodal coordinates of each real element, and the outputs are the elemental stiffness matrices. Since the stiffness matrix is symmetric, only the diagonal and upper triangle of the stiffness matrix need to be computed. The values of n_i and n_c are estimated to yield the ratio α. With the ratio α determined, the ratio β for macropipelining can be determined accordingly.

The input data set corresponds to the coordinates of 4 nodal points, which requires 8 words or n_i = 32 bytes. The context of the computation includes the input coordinates of 4 nodal points, the diagonal and upper triangle of the stiffness matrix, the Jacobian matrix, the B matrix, auxiliary integer indexes and other temporary variables. A context of 32 words is large enough for the given example, which makes n_c = 128 bytes. Note that, the constants are part of the program which does not belong to the context of an element. The ratio $\alpha = n_i/n_c = 1/4$ implies that $\beta > 4$ is required to achieve maximum speedup.



$$x = \sum_{i=1}^{4} a_i\phi_i(\xi, \eta) \qquad y = \sum_{i=1}^{4} b_i\phi_i(\xi, \eta)$$

Figure 4: The isoparametric transformation of a quadralateral finite element

The total number of floating point operations involved determines the ratio T_c/T_i, which corresponds to the maximum parallelism achievable in the preprocessing array. Two parameters are used :

n_f The total number of floating point operations (multiplication or addition) used in the element based computation for an element.

t_f The average processing time of a floating point operation in a processor.

Since most operations involved in the stiffness matrix generation are floating point computations, T_c is estimated to be $n_f t_f$. The speedup for the given example is thus upper bounded by

$$T_c/T_i=(n_f/n_i)(t_f/t_i)=\theta\gamma \qquad (11)$$

The ratio $\theta = (n_f/n_i)$ is a *software parameter* independent of the hardware configuration. The larger is the value of θ, the higher parallelism is available in the preprocessing phase. The ratio $\gamma = t_f/t_i$ is a *hardware parameter* that reflects the hardware configuration used in the linear array. Since a fast communication channel is assumed in the system architecture, we can assume $\gamma >> 1$. An estimation of n_f for the given example is 640, which implies θ=20. In case γ = 10, a speedup of 200 is possible in the preprocessing stage.

The computational properties of various finite elements in solving a linear elastic problem are given below for a 3-D problem.

$$\varepsilon_x = \frac{\partial u}{\partial x} \qquad \varepsilon_y=\frac{\partial v}{\partial y} \qquad \varepsilon_z=\frac{\partial w}{\partial z}$$

$$\varepsilon_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \qquad \varepsilon_{xz} = \frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \qquad \varepsilon_{yz} = \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}$$

$$\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{xz}}{\partial z} = -f_x$$

$$\frac{\partial \tau_{yx}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{yz}}{\partial z} = -f_y \qquad (12)$$

$$\frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{zy}}{\partial y} + \frac{\partial \sigma_z}{\partial z} = -f_z$$

$$[\sigma_x, \sigma_y, \sigma_z, \tau_{xy}, \tau_{xz}, \tau_{yz}]^T = E[\varepsilon_x, \varepsilon_y, \varepsilon_z, \varepsilon_{xy}, \varepsilon_{xz}, \varepsilon_{yz}]^T$$

where E is a 6x6 constant matrix. For structures of thin thickness, it can be simplified to a 2-D problem and qudrilateral plane element can be applied in discretization. For a general problem, a 3-D cubic solid element needs to be used for the discretization. To generate the local stiffness matrix K and force vector f, two numerical integrations are performed:

$$K = \int_0^1\int_0^1\int_0^1 B^T E B \det[J] \; d\xi d\eta d\zeta$$

$$f = \int_0^1\int_0^1\int_0^1 (B^T E \{\varepsilon_0\} - B^T\{\sigma_0\} + N^T\{f_0\})\det[J] \; d\xi d\eta d\zeta \qquad (13)$$

where matrix B depends on nodal coordinates of an element, J is the the Jacobian matrix which links the real coordinate system (x, y, z) to the natural coordinate system (ξ, η, ζ). Matrix N depends on the shape functions used. The size of matrices B and N depends on the number of nodal points used to model the elements. To compute this integration numerically, the Gauss Quadrature method is applied. The number of Gauss points used for the numerical integration depends on the order of the shape function.

In Table.1, we summarize the computational properties for various elements used in a linear elastic structural analysis. These properties correspond to the hardware independent parameters which are used in the performance analysis and design of the preprocessor. The geometrical shapes of these elements are shown in Fig.5 . In all cases the value of α and θ are large enough to justify the use of a linear array as a preprocessor.

Table 1: Computational Properties For Preprocessing Of Various Finite Elements in Fig.4

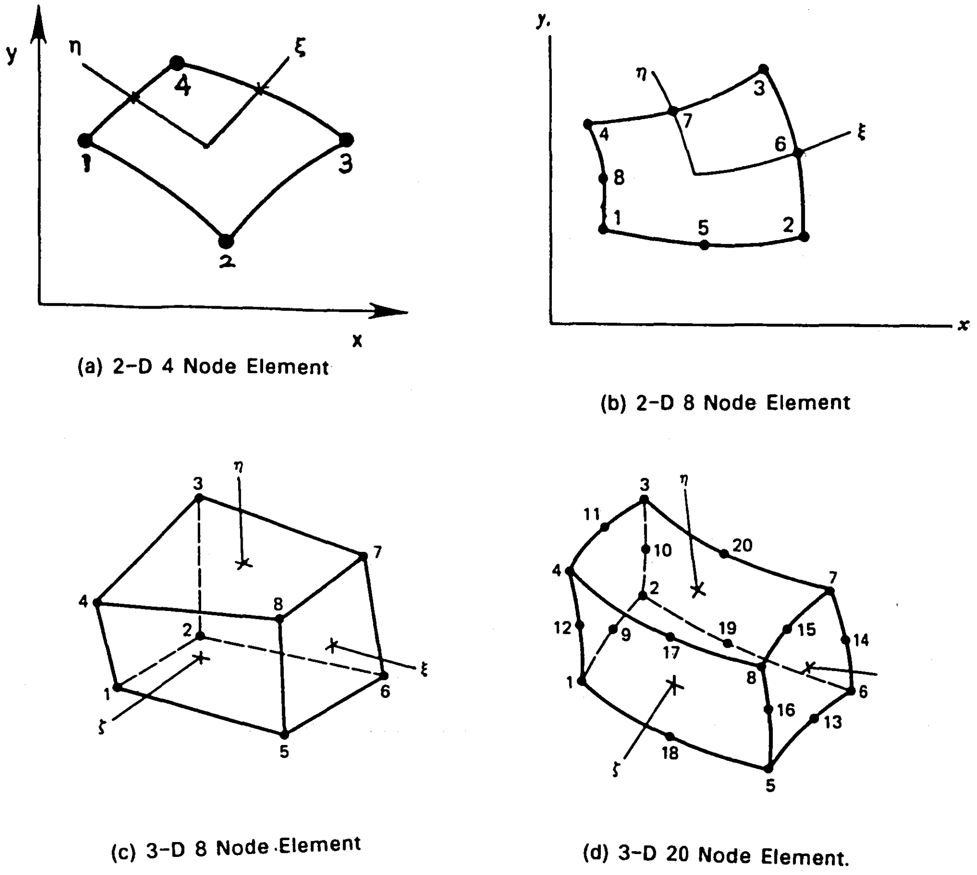
2-D plane elements

nodes	d.o.f	G-points	n_c	n_i	n_f	α	θ
4	8	2x2	512	176	2400	0.34	13.6
4	8	3x3	512	176	5400	0.34	30.7
4	8	4x4	512	176	9600	0.34	54.5
8	16	3x3	1120	352	10,800	0.31	30.7
8	16	4x4	1120	352	19,200	0.31	54.5
8	16	5x5	1120	352	21,000	0.31	59.6

3-D solid elements

nodes	d.o.f.	G-points	n_c	n_i	n_f	α	θ
8	24	3x3x3	3136	672	135,000	0.21	200
8	24	4x4x4	3136	672	320,000	0.21	476
20	60	4x4x4	12,000	1680	768,000	0.14	457
20	60	5x5x5	12,000	1680	1,500,000	0.14	893

Note that :
nodes : number of nodes used in one element.
d.o.f. : degree of freedom in the finite element.
G-points : number of Gauss points used in each dimension.



(a) 2-D 4 Node Element

(b) 2-D 8 Node Element

(c) 3-D 8 Node Element

(d) 3-D 20 Node Element.

Figure 5: Various finite elements for an elastic structural analysis defined in Eq. (12)

Using the same linear elastic elements, consider a simple postprocessing algorithm which interpolates the solution vector, generates the graphical primitives, and sends them to the host computer. The interpolated picture provides a clear contour of the solution vector or the derived solution vector. The software parameter θ'is determined to examine achievable speedup. The value θ' can be estimated from the computation performed to generate one interpolation point and the number of bytes used for coding one such point in graphics. For example, to interpolate the solution for a point (ξ_p, η_p) in a 2-D element, first the norm of the interpolated solution at point P (u_p, v_p) needs to be computed :

$$u_p = \sum_{k=1}^{\text{nodes}} u_k \psi_k(\xi_p, \eta_p)$$
$$v_p = \sum_{k=1}^{\text{nodes}} v_k \psi_k(\xi_p, \eta_p) \qquad (14)$$
$$c_p = (u_p^2 + v_p^2)^{1/2}$$

Then the natural coordinate of point $P(\xi_p, \eta_p)$ is transformed into its real coordinate (x_p, y_p) by :

$$x_p = \sum_{k=1}^{\text{nodes}} x_k \psi_k(\xi_p, \eta_p)$$
$$y_p = \sum_{k=1}^{\text{nodes}} y_k \psi_k(\xi_p, \eta_p) \qquad (15)$$

Then the value of (x_p, y_p, v_p) is scaled into integers corresponding to the graphical primitives. Table 2 lists the 2-D elastic elements discussed in the previous section with an estimated θ' value. The estimations are based on the fact that 4 bytes are used to code one interpolation points inside an element. The estimated θ' value for 3-D elements are also given in the table. In all cases, α' are greater than 1/2. This implies that the hardware with β'= 2 is good for achieving the maximum speedup. Again, the table justifies the choice of a linear array for FEM postprocessing.

Table 2: Computational Properties For Postprocessing Of Various Elastic Finite Elements in Fig.4

2-D plane elements

nodes	d.o.f.	deg(ψ)	θ'
4	8	2	10
4	8	3	12
4	8	4	14
4	16	3	29
4	16	4	33
4	16	5	37

3-D solid element

nodes	d.o.f.	deg(ψ)	θ'
8	24	3	26
8	24	4	30
20	60	4	72
20	60	5	82

Note that :
deg(ψ) : the degree of the shape functions as a polynomial.

6. Conclusions

The importance of using fast pre/postprocessing systolic arrays in a finite-element machine becomes apparent with the increasing use of highly parallel linear system solvers. Computational properties of FEM pre/postprocessing steps reveal that linear arrays are indeed a good choice for these steps. The hardware demand is low and the array is modularly expandable. Macropipelining avoids the problem of software recoding. Our case studies reveal that high speedup factors of 100 or 1000 (assuming γ=10) can be achieved with the macropipelining approach. To reach the maximum speedup, the data transfer rate between processors should be 4 to 8 times faster than the host computer channel speed. The results being presented should be useful to those who are involved in developing efficient I/O and parallel PDE solvers on a multiprocessor supercomputer[15].

References

1. R. D. Cook, *Concepts and Applications of Finite Element Analysis, 2nd ed.,* John Wiley & Sons, 1981.

2. L. A. Hageman and D. M. Young, *Applied Iterative Methods,* Academic Press, 1981.

3. A. Jennings, *Matrix Computation for Engineers and Scientists,* John Wiley & Sons, 1978.

4. K. Hwang, "Multiprocessor Supercomputers for Scientific/Engineering Applications", *IEEE Computer Magazine*, June 1985.

5. O. Lubeck, J. Moore and R. Mendez, "A Benchmark Comparision of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20 and Cray X-MP/2", *IEEE Computer Magazine*, December 1985.

6. A. E. Charlesworth and J. L. Gustafson, "Introducing Replicated VLSI to Supercomputing : the FPS-164/MAX Scientific Computer", *IEEE Computer Magazine*, March 1986.

7. V. Pan and J. Reif, "Efficient Parallel Solutions of Linear Systems", *Proc. of the 17th Annu. ACM Sympo. on Theory of Computing*, May 1985.

8. K. E. Batcher, "Design of a Massively Parallel Processor", *IEEE Trans. on Computers*, September 1980.

9. W. D. Hillis, *The Connection Machine,* MIT Press, 1985.

10. P. B. Schneck, S. L. Squires, J. Lehmann, D. Mizell and K. Wallgren, "US Government Parallel Processor Programs", *IEEE Computer Magazine*, June 1985.

11. A. L. Fisher and H. T. Kung, "Synchronizing Large VLSI Processor Arrays", *IEEE Trans. on Computers*, August 1985.

12. H. T. Kung and M. Lam, "Wafer-Scale Integration and Two-level Pipelined Implementation of Systolic Arrays", *Journal of Parallel and Distributed Computing*1984, pp. 32-63.

13. W. Händler, "The Impact of Classification Schemes on Computer Architecture", *Proc. of Int'l. Conf. on Parallel Processing* , 1977.

14. F.A. Briggs, K.S. Fu, K. Hwang and B.W. Wah, "PUMPS Architecture for Pattern Analysis and Image Database Management", *IEEE Trans. Computers*, October 1982.

15. K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, , 1984.

A NEW CLASS OF PARALLEL ALGORITHMS FOR SOLVING LINEAR TRIDIAGONAL SYSTEMS *

S. Lakshmivarahan and Sudarshan K. Dhall
Parallel Processing Institute
School of Electrical Engineering & Computer Science
University of Oklahoma
Norman, OK 73019, U.S.A.

ABSTRACT

A new class of parallel algorithms for solving linear tridiagonal systems, which compares very favorably with the existing parallel algorithms, is described.

1. INTRODUCTION

There are basically two classes of methods for solving linear tridiagonal systems in parallel. The first of these due to Stone [4,5] is called the method of recursive doubling. This method introduces parallelism in the now-classical Gaussian elimination algorithm. The second method is called the method of cyclic reduction and is due to Hockney [1,9]. There are in fact two versions of this algorithm: one is called the odd-even reduction [3] (also known as serial cyclic reduction [1]) and the other is called the odd-even elimination [3] (also known as parallel cyclic reduction [1]). A version of the recursive doubling given in [1] takes 24logn (all logarithms are to base 2) units of time using a parallelism of size n. Refer to Table 1. The odd-even reduction, on the other hand, takes only 19logn units of time using a parallelism of size $n'/2$ where $n'=n+1=2^t$ for some $t\geq 1$ [1]. The odd-even elimination, however, using a parallelism of size n', takes only $14\log n'$ units of time. It can be easily shown that both the recursive doubling and odd-even elimination require a total of O(nlogn) scalar operations but odd-even reduction needs only O(n) (that is, about 19n) scalar operations. Thus, for large n, when the available parallelism is limited, the odd-even reduction is widely recommended [3]. For a succinct summary of these and many other parallel algorithms, refer to the survey by Heller [3]. The performance of these algorithms on vector machines has been thoroughly analyzed in Stone [5] and Lambiotte and Voigt [11].

In this paper, we describe a new class of parallel algorithms for solving linear tridiagonal systems. Like the method of recursive doubling, this method also introduces parallelism in the solution of the recurrences arising from the Gaussian elimination algorithm. Our method is very flexible, permits vector or multi-processor or a combination of vector and multiprocessor implementations and is adaptable to a wide range of the size of parallelism. This new algorithm is based on the shared memory model called PRAM [10] which allows simultaneous read but only single write in the same memory location. More specifically, it is shown that a linear tridiagonal system of size n can be solved in $(n/p[(32/3)+6\log p]-3$ units of time using $(3/2)p$ processors where $n=2mp$ for m, a multiple of 3, and $p=2^k$ for some $k\geq 1$. Thus, using only 3 processors (p=2) this method takes only 8.33n-3 units but the widely preferred odd-even reduction needs 9n units of time. When p=n/3, that is, using a total of n/2 processors, the new method takes 18logn units which is nearly the same performance as the odd-even reduction method. Further, if p=n/2 that is, using 3n/4 processors, our method needs only 12logn+7 units of time. Further, using n/2 processors, our new method is 55% more processor efficient compared to recursive doubling and 31% more processor efficient compared to odd-even elimination.

In Section 2, we describe the mathematical framework. The algorithm itself is given in Section 3. Section 4 provides a thorough comparison of all the existing methods and suggests improvements in implementing odd-even reduction should more processors become available. Concluding remarks are given in Section 5.

* This work was supported in part by the AMOCO Research Center in Tulsa and Energy Resource Center at the University of Oklahoma.

CH2345-7/86/0000/0315$01.00 © 1986 IEEE

2. A MATHEMATICAL FRAME-WORK

Consider a linear tridiagonal system

$$Ax = K \qquad (2.1)$$

where

$$A=\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & \cdots & & & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & \cdots & & & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & \cdots & & & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & & & & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & & & & \cdot \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & a_n & b_n \end{bmatrix}$$

and

$$K = (k_1,k_2,k_3\ldots k_n)^T$$

where T denotes the transpose. The classical Gaussian elimination algorithm provides a framework for our analysis. We follow the notations given in [1]. Let A=LU be the factorization of A, where

$$L=\begin{bmatrix} e_1^{-1} & 0 & 0 & \cdots & 0 & 0 \\ a_2 & e_2^{-1} & 0 & \cdots & 0 & 0 \\ \cdot & a_3 & e_3^{-1} & & \cdot & \cdot \\ \cdot & \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & \cdot & & \cdot & \cdot \\ 0 & 0 & \cdots & & a_n & e_n^{-1} \end{bmatrix}$$

and

$$U=\begin{bmatrix} 1 & W_1 & 0 & \cdots & & 0 \\ 0 & 1 & W_2 & \cdots & & 0 \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & \cdot & & & W_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

Given the factors L and U, it is well known that solving (2.1) is equivalent to solving a system of three recurrences in the following order:

(a) $$W_i=\frac{c_i}{(b_i-a_iW_{i-1})}, \text{ for } n-1\geq i\geq 2 \qquad (2.2)$$

$W_1=\dfrac{c_1}{b_1}$. Notice that $e_i=\dfrac{W_i}{c_i}$

(b) $$g_i=\frac{(k_i-a_i\ g_{i-1})}{(b_i-a_i\ W_{i-1})}, \text{ for } n\geq i\geq 2 \qquad (2.3)$$

$g_1=\dfrac{k_1}{b_1}$

and

(c) $$x_i=g_i-W_ix_{i+1}, \quad \text{for } n-1\geq i\geq 1 \qquad (2.4)$$

$x_n=g_n$

Solving these three recurrences serially requires not more than 8n scalar operations.

Our method, like the recursive doubling, consists in first converting the first-order nonlinear recurrence (2.2) into a first-order linear recurrence in vector form. More specifically, define

$$W_i=\frac{Y_{i-1}}{Y_i}, \ 1\leq i\leq n-1,$$

$Y_0=1$ and $Y_1=b_1/c_1$

Substituting in (2.2), we obtain

$$V_i=A_iV_{i-1}, \text{ for } n-1\geq i\geq 2 \qquad (2.5)$$

where $V_i=(Y_{i-1},Y_i)^T$, $V_1=(1,b_1/c_1)^T$ and

$$A_i = \begin{bmatrix} 0 & 1 \\ -\dfrac{a_i}{c_i} & \dfrac{b_i}{c_i} \end{bmatrix} \qquad (2.6)$$

Thus, in this framework, solving (2.1) reduces to one of solving the following two sub-problems:

<u>Problem 1</u>: Solve for V_i, i=2 to n when

$$V_i = A_iV_{i-1} \qquad (2.7)$$

where V_i and A_i are 2x1 and 2x2 matrices respectively, given V_1 and A_i's.

<u>Problem 2</u>: Solve for Z_i, i=2 to n when

$$Z_i = f_iZ_{i-1}+d_i \qquad (2.8)$$

where $Z_1,(f_2,f_3,\ldots,f_n)$ and $(d_1,d_2,\ldots,d_n)$ are given.

The following section contains our main result. For other methods of solving linear recurrences refer to [2,6,7,8].

3. A NEW PARALLEL ALGORITHM

We begin by presenting a new parallel algorithm for solving Problem 1. The solution $V_i, i\geq 2$, to (2.7) in closed form is given by

$$V_i = A_iA_{i-1}\dots A_2V_1 = [\prod_{j=2}^{i} A_j]V_1 \quad (3.1)$$

As a first step in computing V_i's define for $u\geq v$

$$B[u,v] = A_uA_{u-1}\dots A_v = \prod_{j=v}^{u} A_j$$

Since the matrix product is not commutative, the order of multiplication is very important to our development. (Vacuous products are taken to be unit matrices). The following properties of the B[.,.] function are obvious:

Lemma 3.1

(a) For any r>1, B[u+r,v] = B[u+r,u+1]B[u,v]

(b) $V_1 = B[i,2]V_1$ for all $i\geq 2$.

This lemma readily suggests the following parallel algorithm. Let $p=2^k$, n=2mp, q=p/2 where $k\geq 1$ and m is a multiple of 3. The following algorithm uses 3q processors.

Algorithm V: Let Q_j, $0\leq j\leq q-1$, refer to a group of 3-processors. For consistency assume, that A_1 is a 2x2 identity matrix.

Stage 1: The processor group Q_0 computes the following:

```
FOR i = 1 to 4m
  $V_i = A_iV_{i-1}$
END
```

Each of the processor group Q_j, $1\leq j\leq q-1$ computes the following in parallel.

```
FOR i = 1 to 4m
  compute B[4mj+i, 4mj+1]
END
```

Stage 2:

```
FOR s = 1 to k-1
  base = 4m x $2^{s-1}$
  Using $2^s$ 3-processor groups, compute
  the following in parallel.
  FOR i = 1 to 4m x $2^{s-1}$
  $V_{base + i}$ = B[base + i, base + 1] $V_{base}$
  END
  FOR g = 1 to $2^{k-s-1}-1$
  using $2^s$ 3-processor groups compute the
  following in parallel.
     FOR i = 1 to 4m x $2^{s-1}$
     Compute B[m(1+2g)$2^{s+1}$+i,mg$2^{s+2}$+1]
     END
  END
END.
```

The following example illustrates the above algorithm.

Example 3.1: Let p=16, q=8 and n=32m for some $m\geq 3r, r\geq 1$. Various steps in each of the stages of the Algorithm V are given in Figure 1.

The following theorem is immediate.

Theorem 3.1: Algorithm V computes V_i's given in (3.1) for $2\leq i\leq n$ using 3q processors in

(n/p)[2+4logp]-3

units of time.

Proof: In stage 1, there are q independent segments of computations. The first segment involves a chain of 4m-1 matrix-vector products and each of the other segments involves chains of 4m-1 matrix products. Allot a 3-processor group to each of the q segments. The multiplication of two 2x2 matrices, using 3-processors can be done in 5-units of time. However, since there is a chain of products, referring to Figure 2, it is readily seen that the chain of (4m-1) matrix products can be completed in 4(4m-1)+1 = 16m-3 units of time. Likewise, the chain of (4m-1) matrix-vector products can be completed in 3(4m-1) = 12m-3 units of time. Clearly, the matrix products dominate, and stage 1 takes (16m-3) units of time.

For any s=1 to k-2 in stage 2, there are 2^{k-s-1} segments of independent computations. The first of these segments involves matrix-vector products and the rest of the segments involve matrix products. By allocating 2^s sets of 3-processor groups to each of the segments, (since the matrix products dominate) it is easily seen that each processor performs (2/3)m matrix products. (Remember that m is a multiple of 3). Thus, the computation in each step s=1 to k-2 can be completed in (2/3)mx12 = 8m units of time.

The last step (s=k-1) involves only the chain of matrix-vector products. Each of the 3-processor group performs only 2m matrix-vector products and this step takes (2m/3)x6=4m units. Thus the total time needed by the algorithm V is (16m-3)+(k-2)(8m)+4m=4m(1+2k)-3. Since

$n=2mp$, the theorem follows.

We now present a parallel algorithm for solving Problem 2 [12,13].

The solution to the linear first-order recurrence (2.8) in closed form is given by

$$Z_i = \sum_{j=0}^{i} [\prod_{s=j+1}^{i} f_s] d_j \qquad (3.2)$$

In the following we present a new parallel algorithm for computing (3.2). To this end, we introduce two functions A[u,v] and Y[s,t] of a_i's and d_i's.

Let

$$A[u,v] = \prod_{r=u+1}^{v} f_r \qquad (3.3)$$

where vacuous products are taken to be unity and let

$$Y[s,t] = \sum_{j=s}^{t} A[j,t]d_j, \text{ for } t \geq s \quad (3.4)$$

A number of useful properties of the functions A[.,.] and Y[.,.] are developed in the following lemmas.

<u>Lemma 3.2</u>: The function A[u,v] satisfies the following:

(a) $A[u,v]=1$ for all $u \geq v$

(b) $A[u,v+m]=A[u,v] \cdot A[v,v+m]$ for any integer $m>0$

The proof of this lemma is straightforward.

<u>Lemma 3.3</u>: The function $Y[s,\cdot]$ also satisfies the recurrence of the type (1.1), that is,

$$Y[s,t]=f_t Y[s,t-1]+d_t \qquad (3.5)$$

<u>Proof</u>: Rewriting (3.4) we readily obtain

$$Y[s,t] = \sum_{j=s}^{t-1} A[j,t]d_j + A[t,t]d_t$$

Since $A[t,t]=1$ and $A[j,t]=f_t A[j,t-1]$, the lemma follows.

<u>Lemma 3.4</u>: $Z_t=Y[1,t]$ for all $1 \leq t \leq n$. The proof follows from the definitions.

The following lemma which is a generalization of the Lemma 3.3 provides the basis for our algorithm.

<u>Lemma 3.5</u>: If $t \geq s$ then for $i>0$

$$Y[s,t+i]=A(t,t+i\}Y[s,t]+Y[t+1,t+i] \qquad (3.6)$$

<u>Proof</u>: From equation (3.4) it follows that

$$Y[s,t+i] = \sum_{j=s}^{t+i} A[j,t+i]d_j$$

$$= \sum_{j=s}^{t} A[j,t+i]d_j + \sum_{j=t+1}^{t+i} A[j,t+i]d_j \qquad (3.7)$$

It is readily seen from equation (3.4) that the second term is in fact Y[t+1,t+i]. From Lemma 3.2

$$A[j,t+i]=A[j,t]A[t,t+i] \qquad (3.8)$$

Substituting (3.8) in the first term of (3.7), the lemma follows.

Let $n=2mp$, $p=2^k$ $q=p/2$ where $m \geq 1$ and $k \geq 1$. In the following we use 3q processors. Let $1 \leq s \leq k$ and $0 \leq g \leq 2^{k-s}-1$. Our algorithm for solving (2.8) consists of two parts: (1) a parallel <u>Algorithm A</u> for computing A[u,v] and (2) a parallel <u>Algorithm Y</u> for computing Y[s,t]. In fact Algorithm A computes all those A[u,v] which are needed in the computation of Y[s,t]. Since these two algorithms are working in parallel, the total number of processors is equal to the sum of the processors used in both the algorithms but the time is maximum of the time needed by the two algorithms. The Algorithm Y uses p processors $P_0, P_1 \ldots P_{p-1}$ and Algorithm A uses q processors $Q_0, Q_1 \ldots Q_{q-1}$.

<u>Algorithm A</u>:

<u>Stage 1</u>: Each processor $Q_j, 0 \leq j \leq q-1$, computes the following:

```
FOR i=1 to 2m
Compute A[2m(1+2j),2m(1+2j)+i]
END
```

In this stage totally 2qm number of the A[.,.] functions are updated using q processors. Each update involves one multiplication. The stage takes n/p(=2m) units of time.

<u>Stage 2</u>: Each processor $Q_j, 0 \leq j \leq q-2$, computes the following:

```
FOR i=1 to 2m
Compute A[4m(1+j),4m(1+j)+i]
END
```

In this stage totally 2m(q-1) number of A[.,.] functions are updated using (q-1) processors. This stage also takes n/p units of time.

Stage 3:

```
FOR s=1 to k-1
  FOR j=0 to $2^{k-s}-2$
  Let $h_j=6m2^{s-1}+2^{s+1}mj$
    FOR i=1 to 2ms
    Compute A[$h_j$-2ms, $h_j$+i]
    END
  END
END
```

In any step s, there are $2^{k-s}-1$ groups of computations and within each group 2ms number of A[.,.] functions are updated. Since $2ms(2^{k-s}-1) \leq 2m(q-1)$, using not more than (q-1) processors, each of the above steps can be finished in not more than n/p units of time. Thus the entire Algorithm using q processors does not take more than (n/p[1+logp] units of time.

Algorithm Y:

Stage 1: Each processor $P_i, 0 \leq i \leq p-1$, computes the following:

```
FOR j=0 to m-1,
Y[2im+1,2im+1+j]=$a_{2im+1+j}$Y[2im+j]
+$d_{2im+1+j}$
END
```

Stage 2: Each of the processors $P_i, 0 \leq i \leq p-1$, computes the following:

```
FOR j=0 to m-1,
Y[2im+1,(2i+1)m+1+j]=
          $a_{(2i+1)m+1+j}$Y[2im+1),
          (2i+1)m+j]+$d_{(2i+1)m+1+j}$
END
```

In each of the stages 1 and 2, totally n/2 of the Y[.,.] functions are updated using p processors. Since each update requires two operations (a multiplication and an addition) each of the above stages takes n/p units of time.

Stage 3:

```
FOR s=1 to k
  FOR g=0 to $2^{k-s}-1$
    FOR i=1 to $m2^s$
    Y[$2^{s+1}$mg+1,$2^s$m(1+2g)+i] =
    A[$2^s$m(1+2g),$2^s$m(1+2g)+i]Y[$2^{s+1}$mg+1,
    $2^s$m(1+2g)]+Y[$2^s$m(1+2g)+1,$2^s$m(1+2g)+i]
    END
  END
END
```

It is readily seen that at any step s in stage 3, the algorithm identifies 2^{k-s} groups indexed by g and within each group there are 2^s processors working in parallel. Thus within each group $2^s m$ number of Y[.,.] functions are updated. It readily follows that computations in each group take the same amount of time equal to n/p units. Thus the entire stage 3 takes (n/p)logp units of time.

For the correct functioning of the Algorithm Y, the values of A[2^sm(1+2g), 2^sm(1+2g)i] for $0 \leq g \leq 2^{k-s}$ and $1 \leq i \leq k$ must be made available for each $1 \leq s \leq k$. In fact, these functions are generated in parallel by the Algorithm A. A close scrutiny reveals that all the values of the A[.,.] functions that are needed in stage 3 of Algorithm Y are in fact generated by Algorithm A at least one step ahead of time. Thus the Overall time for solving for Z_i's in (2.8) is decided by the running time of the Algorithm Y.

We now illustrate the Algorithms Y and A through an example.

Example 3.2: Let p=8, q=4, k=3, m=6, n=96. The various stages of the Algorithm Y are shown in Figure 3. In Figure 4 further details of the computations in Stage 3 of Algorithm Y are illustrated. Figure 5 illustrates various stages of Algorithm A. Comparing Figures 4 and 5, it is readily seen that the values of the A[.,.] function are made available at least one step ahead of the time they are needed in Algorithm Y.

Theorem 3.2: Using a total of 3q processors, Algorithms Y and A compute Y[1,t] for $1 \leq t \leq n$ in (n/p)[2+logp] units of time.

Proof: At the end of stage 1 of Algorithm Y, Y[1,t] for $1 \leq t \leq m$ and at the end of stage 2, Y[1,t] for $1 \leq t \leq 2^{i+1}m$ are computed. Thus the entire sequence of Y[1,t] for $1 \leq t \leq n$ is available at the end of stage 3. This proves the correctness of the Algorithm Y.

Of the (3/2)p processors, Algorithm Y uses p and Algorithm A uses p/2. Further, since Algorithms A and Y are acting in parallel and Algorithm Y takes longer time compared to Algorithm A, the total time is that of Algorithm Y. It is readily seen from the description of the algorithms, that this latter algorithm takes (n/p)[2+logp] units of time and the theorem follows.

We now state our main result.

Theorem 3.3: Let $p=2^k$, q=(p/2), n=2mp and $k \geq 1$ and m is a multiple of 3. A linear tridiagonal system of size n can be solved, using 3q processors in

(n/p)[(32/3)+6logp]-3

units of time.

Proof: Setting up each matrix A_i involves 2 divisions and 1 sign change operation. Since there are (n-1) matrices and an initial condition to be computed, this overall initialization, using 3q processors takes (2n/p) units of time. Solving for W_i, and setting up the g_i computation in the form (2.9) needs the computation of (W_i/c_i), $k_i(W_i/c_i)$ and $a_i(W_i/c_i)$. This part takes, using 3q processors (8n/3p) units. Then solving for g_i and x_i using Algorithms A and Y takes (2n/p)[2+logp] units. Adding up all these factors the theorem follows.

4. A COMPARISON OF VARIOUS ALGORITHMS

Table 1 gives the time and processor requirements for a number of known algorithms. It is readily seen from Table 1 that, with (n/2) processors, the new method has 55% more processor efficiency compared to recursive doubling and 31% more processor efficiency compared to odd-even elimination.

For the purposes of comparing the new method with the widely preferred odd-even reduction, we describe the latter as follows [1]:

Let $n' = n+1 = 2^t$ for some t>0. Define $p_i^{(u)} = (a_i^{(u)}, b_i^{(u)}, c_i^{(u)}, k_i^{(u)})$ where $p_i^{(o)} = (a_i, b_i, c_i, k_i)$.

Odd-Even Reduction:

Stage 1: This stage computes a set of vectors $p_i^{(u)}$:

```
FOR u=1 step 1 unitl t-1
  FOR i=$2^u$ step $2^u$ until n-$2^u$
      $H = 2^{u-1}$
      $\alpha i = a_i^{(u-1)}/b_{i-H}^{(u-1)}$
      $\gamma i = c_i^{(u-1)}/b_{i-H}^{(u-1)}$
    $a_i^{(u)} = -\alpha_i\ a_{i-H}^{(u-1)}$
    $c_i^{(u)} = -\gamma_i\ a_{i-H}^{(u-1)}$
    $b_i^{(u)} = b_i^{(u-1)} - \alpha_i\ c_{i-H}^{(u-1)} - \gamma_i\ a_{i-H}^{(u-1)}$
    $k_i^{(u)} = k_i^{(u-1)} - \alpha_i\ k_{i-H}^{(u-1)} - \gamma_i\ k_{i-H}^{(u-1)}$
    END
END.
```

Stage 2: This stage recovers the solution recursively. Let $X_0 = X_{n'} = 0$

```
FOR u=t step -1 until 1
  FOR i=$2^{u-1}$ step $2^u$ unitl n´-$2^{(u-1)}$
    $H=2^{u-1}$
    $x_i=[k_i^{(u-1)} - a_i^{(u-1)} x_{i-H}$
        $-c_i^{(u-1)} x_{i+H}]/b_i^{(u-1)}$
  END
END
```

This algorithm requires a total of only 19n scalar operations and thus is widely preferred when only a small number of processors are available. It is readily seen that the body of the loop in stage 1 involves a total of 14 scalar operations and that in stage 2 involves a total of 5 scalar operations. Analyzing the data dependency in the computations of the above loops, it can be seen that with 3 processors, stage 1 takes 5n units and stage 2 takes 4n units requiring a total of 9n units. However, with 3 processors the new algorithm requires only 8.33n-3 units.

Likewise, if there are 6 processors available, each iteration through the loop in stage 1 takes 4 units and that in stage 2 takes 4 units, requiring a total of 8n units. But the new method takes only 5.67n-3 units. Thus, the new method compares favorably with the odd-even reduction.

5. CONCLUSION

A new class of parallel algorithms for solving tridiagonal systems that are suitable for implementation on shared memory multi-vector architectures such as CRAY-XMP, Alliant, etc. is described. While our analysis primarily relates to the arithmetic complexity, it should be interesting to actually implement and compare its performance with other parallel algorithms for solving tridiagonal systems.

REFERENCES

1. R.W. Hockney and C.R. Jesshope, Parallel Computers, Adam and Hilger Ltd, Bristol, 1981, Chapter 5.

2. D.J. Kuck, The Structure of Computers and Computations, Vol. 1, Addison-Wesley, 1980, Chapter 2.

3. D. Heller, "A Survey of Parallel Algorithms in Numerical Algebra," SIAM Review, Vol. 20, 1978, pp. 740-777.

4. H.S. Stone, "An efficient Parallel Algorithm for the Solution of Tridiagonal System of Equations," Journal of ACM, Vol. 20, 1973, pp. 27-38.

5. H.S. Stone, "Parallel Tridiagonal Equation Solvers," ACM Transactions on Mathematical Software, Vol. 1, 1975, pp. 289-307.

6. S.C. Chen, D.J. Kuck and A.H. Sameh, "Practical Parallel Band Triangular System Solvers," ACM Transactions on Mathematical Software, Vol. 4, 1978, pp. 270-277.

7. A.H. Sameh and R.P. Brent, "Solving Triangular Systems on a Parallel Computer," SIAM Journal on Numerical Analysis, Vol. 14, 1977, pp. 1101-1113.

8. D.D. Gajski, "An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines," IEEE Transactions on Computers, Vol. 30, 1981, pp. 190-206.

9. B.L. Buzbee, G.H. Golub and C.W. Nielson, "On Direct Methods for Solving Poisson's Equations," SIAM Journal on Numerical Analysis, Vol. 7, 1970, pp. 627-655.

10. S. Fortune and J. Wylie, "Parallelism in Random-Access Machines," Proceedings of the 10th Annual ACM Symposium on Theory of Computing, 1978, pp. 114-118.

11. J.L. Lambiotte, Jr. and R.G. Voigt, "The Solution of Tridiagonal Linear Systems on CDC-Star 100 Computer," ACM Transactions on Mathematical Software, Vol. 1, 1975, pp. 308-329.

12. S. Lakshmivarahan and S.K. Dhall, "New Parallel Algorithms for Solving First-Order and Certain Classes of Second-Order Linear Recurrences," International Conference on Parallel Processing, August 20-23, 1985, pp. 843-845.

13. S.K. Dhall, S. Lakshmivarahan and M.V.R. Seshacharyulu, "Solving for Cascade Sums and First-Order Linear Recurrence on the HEP," Proceedings of the Workshop on Parallel Processing Using the HEP, March 20-21, 1985, pp. 303-326.

TABLE 1

METHOD	TOTAL NUMBER OF PROCESSORS	TIME	SPEED-UP RATIO	EFFICIENCY	TOTAL NUMBER OF SCALAR OPERATIONS
Serial Gaussian Elimination [1]	1	8n	-	-	8n
Recursive Doubling [1]	n	24logn	$\frac{n}{3logn}$	$\frac{1}{3logn}$	O(nlogn)
odd-even* reduction [1],[3]	$n'/2$	$19logn'-14$	$\frac{8n}{19logn'-14}$	$\frac{16}{19logn-14}$	$19n'$
odd-even* elimination [1],[3]	n'	$14logn'+1$	$\frac{8n}{14logn'+1}$	$\frac{8}{14logn+1}$	$14n'logn'$
New Method	n/2	18logn	$\frac{4n}{9logn}$	$\frac{8}{9logn}$	
	3n/4	12logn+7	$\frac{8n}{12logn+7}$	$\frac{32}{36logn+19}$	
	3p/2 $2\leq p\leq n/2$	(n/p)[(32/3)+6logp]-3	$\frac{8p}{(32/3)+logp-(3p/n)}$	$\frac{16}{32+18logp-9p/n}$	

*$n'=n+1=2^t$ for some t>0.

STAGE 1 $1\leq i\leq 4m$	STAGE2 Step 1 $1\leq i\leq 4m$	 Step 2 $1\leq i\leq 8m$	 Step 3 $1\leq i\leq 16m$
Q_0 $V_i=A_iV_{i-1}$			
Q_1 $B[4m+i,4m+1]$	Q_0-Q_1 $V_{4m+i}=$ $B[4m+i,4m+1]V_{4m}$		
Q_2 $B[8m+i,8m+1]$		Q_0-Q_3 $V_{8m+i}=$ $B[8m+i,8m+1]V_{8m}$	
Q_3 $B[12m+i,12m+1]$	Q_2-Q_3 $B[12m+i,8m+1]$		
Q_4 $B[16m+i,16m+1]$			Q_0-Q_7 $V_{16m+i}=$ $B[16m+i,16m+1]V_{16m}$
Q_5 $B[20m+i,20m+1]$	Q_4-Q_5 $B[20m+i,16m+1]$		
Q_6 $B[24m+i,24m+1]$		Q_4-Q_7 $B[24m+i,16m+1]$	
Q_7 $B[28m+i,28m+1]$	Q_6-Q_7 $B[28m+i,24m+1]$		

Figure 1: An Illustration of Algorithm V. $p=16$, $q=8$, $n=32m$ for some $m\geq 1$.

Let

$$\begin{bmatrix} p_1 & q_1 \\ r_1 & s_1 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix}$$

and for $i\geq 1$

$$\begin{bmatrix} p_{i+1} & q_{i+1} \\ r_{i+1} & s_{i+1} \end{bmatrix} = \begin{bmatrix} p_i & q_i \\ r_i & s_i \end{bmatrix} \begin{bmatrix} a_{i+2} & b_{i+2} \\ e_{i+2} & d_{i+2} \end{bmatrix}$$

Time →

Processor ↓	0	1	2	3	4	5	6	7	8	9	10	
1	a_1a_2	a_1b_2	c_1a_2	c_1b_2	p_1a_3	p_1b_3	r_1a_3	r_1b_3	p_2a_4	p_2b_3	....	....
2	b_1c_2	b_1d_2	d_1c_2	d_1d_2	q_1c_3	q_1d_3	s_1b_3	s_1d_3	q_2c_4	q_2d_4	....	....
3		p_i	q_1	y_2	s_2	p_2	q_2	y_2	s_2	p_3	q_3	....

|◄—— 4 units ——►|◄—— 4 units ——►|◄—— 4 units ...

Figure 2: A schedule for obtaining the chain of matrix products required in Algorithm Y in stage 1.

STAGE 1	STAGE 2	STAGE 3		
0≤j≤5	0≤j≤5	s=1 1≤i≤12	s=2 1≤i≤24	s=3 1≤i≤48
P_0: Y[1,1+j]				
	P_0: Y[1,7+j]			
P_1: Y[13,13+j]		P_0-P_1 Y[1,12+i]		
	P_1: Y[13m19+j]			
P_2: Y[25,25+j]			P_0-P_3 Y[1,24+i]	
	P_2: Y[25,31+j]			
P_3: Y[37,37+j]		P_2-P_3 Y[25,36+i]		
	P_3:Y[37,43+j]			
P_4:´ Y[49,49+j]				P_0-P_7 Y[1,48+i]
	P_4: Y[49,55+j]			
P_5: Y[61,61+j]		P_4-P_5 Y[49,60+i]		
	P_5: Y[61,67+j]			
P_6: Y[73,73+j]			P_4-P_7 Y[49,72+i]	
	P_6: Y[73,79+j]			
P_7: Y[84,84+j]		P_6-P_7 Y[73,84+i]		
	P_7: [84,91+j]			

Figure 3: Various Stages of Algorithm Y when n = 96, p = 8, m = 6.

s = 1, 1 ≤ i ≤ 12

g = 0: Y[1,12+i] = A[12,12+i]Y[1,12] + Y[13,12+i]

g = 1: Y[25,36+i] = A[36,36+i]Y[25,36] + Y[37,36+i]

g = 2: Y[49,60+i] = A[60,60+i]Y[49,60] + Y[61,60+i]

g = 3: Y[73,84+i] = A[84,84+i]Y[73,84] + Y[85,84+i]

s = 2, 1 ≤ i ≤24

g = 0: Y[1,24+i] = A[24,24+i]Y[1,24] + Y[25,24+i]

g = 1: Y[49,72+i] = A[72,72+i]Y[49,72] + Y[73,72+i]

s = 3, 1 ≤ i≤ 48

g = 0, Y[1,48+i] = A[48,48+i]Y[1,48] + Y[49,48+i]

Figure 4: Details of the Stage 3 Computation for Algorithm Y when n = 96, p = 8 and m = 6.

STAGE 1	STAGE 2	STAGE 3	
$1 \leq i \leq 12$	$1 \leq i \leq 12$	$s = 1$ $1 \leq i \leq 12$	$s = 2$ $1 \leq i \leq 24$
Q_0: A[12,12+i]			
	Q_0: A[24,24+i]		
Q_1: A[36,36+i]		Q_0: A[24,36+i]	
	Q_1: A[48,48+i]		Q_0-Q_1 A[48,72+i]
Q_2: A[60,60+i]		Q_1: A[48,60+i]	
	Q_2: A[72,72+i]		
Q_3: A[84,84+i]		Q_2: A[72,84+i]	

Figure 5: Various Stages of the Algorithm A when $n = 96$, $m = 6$ and $p = 8$.

A PARALLEL COMPUTER BASED ON CUBE CONNECTED CYCLES FOR WAFER SCALE INTEGRATION

Moon Jung Chung, Edward J. Toy*, Aarti,Gupta

Rensselaer Polytechnic Institute, Troy NY 12180

ABSTRACT

The wafer scale integration (WSI) implementation of a single instruction multiple data (SIMD) computer with processing elements (PEs) grouped in a Cube Connected Cycles (CCC) network is presented. Such a network is a good candidate for WSI due to the homogeneity of the network's PEs and the localized processor connections. A RISC based instruction set and an architecture supporting such an instruction set has been developed for the processing element. The data path design for a single chip implementation of this processing element is outlined, along with initial timing and area estimates. Programming is accomplished by the specification of logical networks of processors, providing great flexibility in using the CCC network. Yield considerations for the implementation of this parallel computer with 512 PEs capable of operating on up to 128k data points for a 2μm CMOS process in WSI are given.

Keywords: Cube Connected Cycles, Parallel Processing, Wafer Scale Integration, VLSI, Computer Architecture.

Introduction

Scientists and engineers are frequently confronted by computationally intense problems. Solving these problems by traditional sequential algorithms are proving inadequate due to their lengthy processing times. In response, parallel algorithms for such problems as Fourier Transforms, convolution and matrix manipulation have been developed[1,21,26,29,36] To realize the computational power of these algorithms, massively parallel computers need to be developed. Application of this technology in such diverse fields as physics, computational geometry, signal processing and design automation and could potentially lead to the solutions of problems beond the scope of present computers.

This work describes the implementation of a parallel computer consisting of 512 processors functioning as an SIMD machine, connected with the Cube Connected Cycles (CCC) network[32], capable of operations on problem sizes of up to 128k data points. Wagner has developed a boolean vector computer based on this network[39]. While this machine is only capable of boolean operations on three 1 bit variables, it serves to demonstrate the versatility of the network. The salient features of the CCC network are: 1) interprocessor connections are strictly limited to three; 2) the network control is highly regular; 3) all processing elements are identical. A CCC network can also efficiently emulate other networks (e.g. Shuffle Exchange, Tree based, Mesh networks). It is also our intention to demonstrate that WSI is an effective means of implementation of such systems. The design will be discussed from the point of view of both hardware and software.

The paper is divided as follows. Wafer Scale Integration is introduced first, giving the rational, problems and proposed solutions of the technology. A brief review of the CCC interconnection network, followed by a novel method of supplying instruction streams to the network is then described. The envisioned programming environment is then outlined. Detailed description of the processing element, covering the instruction set, data path archictecture, timming, floor paln and initial area estimates are given. In conclusion, implemetaion of this computer in both a wafer scale integration and hybrid packaging enviornments are considered. A summary of this work was presented by Chung et. al[6,7]

Wafer Scale Integration (WSI)

Large digital circuits can be fabricated through WSI technology to yield a complete component encompassing an entire silicon wafer. The silicon wafer is then packaged intact and can be used as a component in conventional (i.e. board level) design methods, forming a complete system. In WSI, these large circuits are partitioned into modules which form the die sites on the wafer. The fabrication of these die sites is identical to that of current VLSI fabrication technology. After fabrication, the devices are tested and functioning dice are connected in place to form a working component. The main deviation from current IC technology is that the wafer is left intact. Not all the dice fabricated will be functional and hence WSI is dependent upon the wafer's yield, implying the need for redundancy and fault tolerance in design[8,24,31,33]. An effective method must exist to interconnect the functioning devices on the wafer if WSI is to be considered a viable technology[23,25,33,34] The following section gives the rationale behind WSI. An overview of the obstacles in WSI, along with some of the proposed solutions for these obstacles are outlined. Concluding our discussion of WSI will be a description of the technology selected to implement this project.

* Supported by a Digital Equipment Corporation Fellowship

Why Wafer Scale Integration?

The concept of wafer scale integration (WSI) to produce large scale integrated (LSI) circuits was first proposed by [22]. Initially, it was seen as a method for combatting poor yield in fabricating LSI circuits. As semiconductor manufacturing matured, increased die sizes permitted the required levels of integration within a single chip. At that point, the WSI concept was no longer considered a viable technology[25].

Over the past two decades, the majority of the cost in electronics systems has migrated from the raw components to packaging and assembling the components[8]. Avoidance of costly printed circuit packaging has led to interest into better utilization of silicon interconnect[8,27]. While the cost of a silicon wiring layer is approximately ten times that of a printed circuit wiring layer, three orders of magnitude increase in wire density is achieved[8]. The result is a system with near a factor of 100 increase in function/cost with silicon wiring[8]. By using WSI, an order of magnitude increase from conventional packing density can also be achieved[19,27]

Another concern in electronic systems is the communication speeds between modules. With conventional technologies, significant delays can accrue when signals travel between packages[38]. By incorporating signal paths within a single package, the inter-module delay will be reduced due to shorter wire lengths[34]. If all the critical paths reside within the wafer, significant performance gains can be achieved[38]. Further gains in signal speed can be realized by fabrication of thick wires on the wafer ($\sim 5\mu m$ thick X $10\mu m$ wide on $5\mu m$ of dielectric). Signal paths of these dimensions exhibit LC-transmission line behavior as opposed to RC charging lines, reducing the delay to that due to propagation at the speed of light (in the dieletric medium being used) over the length of the wire[2,10]. Silicon wiring also reduces the capacitive loads seen by the chip's drivers, resulting in smaller drivers and reduced power supply requirements[25,27,38]. Although the power consumption will decrease, the heat flow (watts/cm^2) will increase due to the much denser system.

With WSI the overall system reliability can be improved. The major failure sites of modern electronic systems occur at the interface of the various packaging levels used[15,25,31]. It has been noted in [16] that the chip to pin connections are the major source of device failure. Causes of such failures include weakened die to package bonds[15] and thermal stresses[16]. By incorporating a majority of the connections within a single package (i.e. a wafer), the interconnection system remains mono-metallic and will therefore be more reliable. If the number of external I/O ports on the wafer are less than the pin to chip connections in a printed circuit board, the overall system reliability will increase. The entire wafer would finally be hermetically sealed, further reducing the system's vulnerability[27].

WSI Problems & Solutions

Current problems inhibiting the development of WSI systems include developing a feasible interconnection technology, heat dissipation, package design, CAD tool development and architecture selection. The development of a method to reconfigure wafers is crucial for the successful implementation of WSI systems. Proposed methods for restructuring the wafer include laser programming [4,13,], electrically alterable signal paths[17], electron beam programming of floating gate transistors[34] and discretionary wiring[22,25]. (Discretionary wiring connects working modules with additional layers of interconnect). Some functioning wafer scale systems have been fabricated with such techniques[12,17,33,34]. If conventional geometries are used to form the interconnect, coupled with parasitics introduced in the restructuring process, the speed of the interconnect can restrict the overall system performance[10]. WSI packages employing water[31], liquid nitrogen and air[19] cooling techniques have been developed for heat flows of up to 50 W/cm^2[31]. Although extensive CAD tools have been developed for the routing of WSI circuits[9,19,31,33], simulation tools for such large systems are still unavailable.

Using WSI for logic intensive applications has been criticized in the past due to the redundancy required when using different reticles on the same wafer[5,8,17]. To avoid such problems, the architectures considered in this work have been limited to the Single Instruction Multiple Data (SIMD) type. Since the wafer will be populated with a single die type (neglecting the I/O circuitry), any defective processor can be replaced with any available spare. Another advantage with the SIMD approach is that the PEs will require a limited amount of control circuitry, reducing the overall area of the die significantly. A number of SIMD networks were considered, with the CCC being selected as the best available.

The discretionary wiring approach as outlined in [25] will provide the implementation vehicle for the CCC computer. An intermediate step towards this goal using the Wafer Transmission Module (WTM)[10,37] will also be used. Both these issues are briefly outlined below.

Discretionary Wiring With Wafer Transmission Lines

The approach to wafer customization in [25] combines discretionary wiring with a thick film liftoff process[2]. The wiring formed by this approach will be of transmission line quality. Layers are provided for the power and ground planes, two signal layers and three via layers for interconnection of signals between layers. First a $10\mu m$ dielectric (polyimide) layer is deposited on the wafer. A pattern is then defined and etched, providing stud vias from the die sites to the remaining signal layers. Another layer of polyimide is applied, patterned and etched defining the the ground plane. The process is then repeated for successive layers. Note that this process results in planar layers of metalization on the wafer.

Since each wafer will have a unique interconnection pattern, an efficent method of transferring this pattern to the wafer is required. The use of optical masks would be inefficent since such masks would be used once and then be discarded. A system to directly transfer the interconnection pattern from a computer data base to the wafer is being developed using an IBM EL-2 Direct Write Electron Beam system[9]. The discretionary wiring must have a 100% yield, or a system to repair wiring faults will have to be developed. Since no fabrication process can provide a 100% yield, a wire repairing system is being developed through the use of a scanning electron microscope to locate faults in conjunction with a focussed ion beam to repair such faults[9].

The Wafer Transmission Module (WTM)

To avoid the initial fabrication problems associated with discretionary wiring, the Wafer Transmission Module (WTM) was proposed[10,37]. The WTM is a hybrid package incorporating signal interconnect, bypass capacitors and power buses on a substrate. Differences between the WTM and current hybrid packaging include the use of thick film interconnect and a silicon wafer as a substrate. Using a silicon substrate permits batch processing of the substrate with current IC processes, while also minimizing the thermal stresses within the package. Chips are fabricated, tested and scribed as in conventional processing. Functioning chips are then affixed to the wafer by wire bonding or solder bumps. Faults in the WTM interconnect are avoided by providing redundant signal paths such as 10 bit lines for an 8 bit bus, 2 control lines where 1 is required etc. The interconnect would be tested prior to mounting the chips, which are then connected to only functioning wires. If a wafer does not have enough good wires, it will be discarded. The WTM eliminates the dependence on yield and reconfiguration in the WSI approach while achieving comparable speed, density and reliability. A similar package has been fabricated using thin film interconnect for a four chip Wallace multiplier[18].

The Cube Connected Cycles (CCC)

The CCC is derived from the binary n-cube[30] such that the network has a bounded node degree with no significant loss in processing speed. Galil and Paul[11] have used the CCC in defining a universal parallel machine, implying that the CCC can be used to simulate any special purpose parallel computer with a reasonable slow down factor ($\approx O[\log^2{}_2 n]$ time). The CCC can also do certain permutations in $O[\log_2 n]$ time[32].

The CCC network consists of $n=2^k$ processing elements, k being an integer such that $k \leq r+2^r$, r being the smallest integer that satisfies this equation. Each PE has a k bit address M, which is partitioned into two fields $[\Phi,p]$ such that $M=\Phi 2^r+p$. The PEs are grouped into $2^{(k-r)}$ cycles of 2^r nodes per cycle. The (k-r) bits of the Φ field specifies the PE's cycle, while the r bits of the p field gives the PE's position within the cycle. The inter-cycle connections form a (k-r) dimensional cube or hypercube. A 64 PE CCC network is shown in figure 1 with n=64, k=6 and r=2. Note that there are $2^{(k-r)}=16$ cycles of $2^r=4$ nodes per cycle. The connections between PEs of the same cycle are termed links, while connections between processors in different cycles are called sheaves. The PEs of the CCC network have three I/O ports denoted forward (F), backward (B) and lateral (L). The F, B, and L PE interconnections are defined by the following equations:

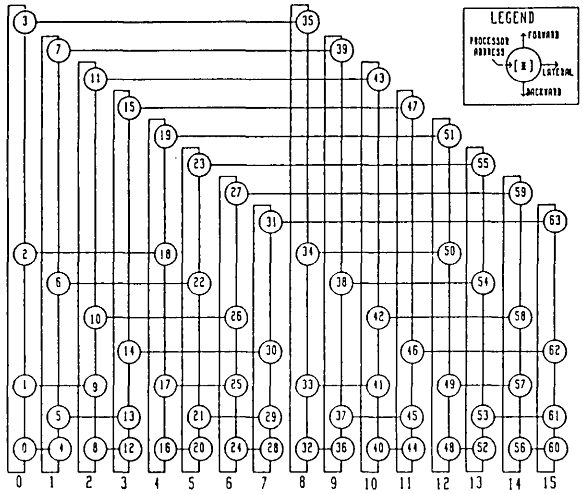


Figure 1 : 64 PE CCC Network

1) $F(\Phi,p)= B(\Phi,(p+1)\bmod 2^r)$
2) $B(\Phi,p)= F(\Phi,(p-1)\bmod 2^r)$
3) $L(\Phi,p)= L(\Phi+\epsilon .2^p,p)$;
   where $\epsilon = 1 - 2.\mathrm{bit}_p(\Phi)$

The term $\mathrm{bit}_p(\Phi)$ is the value of the p^{th} bit of the binary expansion of Φ, and can take a value of 1 or 0. Note that the F-B connections are links and the L connection is a sheaf. The CCC network of figure 1 will be controlled by a host computer. The controller will provide both instruction and data streams, along with other control signals, to the CCC network. The controller's design is the subject of future research.

Instruction Broadcasting

In a synchronous SIMD machine (as in this case), all PEs must execute the same instructions at the same time. Ideally, it would be desirable to broadcast instructions to each processor in the network concurrently. Unfortunately, in this network it is also desirable to minimize the global wiring since signal transmission via this wiring would be slow and consumes much power. An alternative to global broadcasting is necessary that minimizes the delay of instruction distribution while ensuring that all PEs execute the same instruction at the same time. The approach taken restricts communication of instructions to occur only between adjacent processors within a cycle (i.e. those whose p addresses differ by 1). Instructions are broadcast from the controller to the first node of each cycle (i.e. to node of address (*,0)). The instructions are then transferred from this node to the rest of the nodes within the cycle. The instruction transfer is pipelined so that only one instruction per execution phase need be transmitted to the cycle. This method is similar to that of filling a data path pipeline before a result can be output.

To minimize delay in instruction transfer, each PE is equipped with an instruction queue (IQ) which stores the instructions to be executed by the PE. Consider the instruction stream that is to be executed as shown in figure 2 and the CCC cycle in figure 3A. Note that the length of the

Instruction number	Instruction
0	INST A
1	INST B
2	INST C
•	•
•	•
•	•
2^r-2	INST (Q-2)
2^r-1	INST (Q-1)
2^r	INST (Q)

Figure 2- Instruction stream for input to the cycle

instruction queue is different for each node and depends on its position within the cycle. The length of the instruction queue can be determined from the cycle length and the node's p address field by the following equation:

$$\text{Queue Length (QL)}=2^r+1-p$$

Where p is the node's position within the cycle and 2^r is the number of nodes within the cycle. Before instruction execution can begin, we must distribute the first 2^r+1 instructions to each cycle within the network.

The process begins by transmitting the first instruction to the first node of the cycle (see figure 3A). This instruction is stored in the front of node(*.0)'s instruction queue. In the next phase, the second instruction is transmitted to node(*.0) of the cycle and is placed in the second position of node(*.0)'s instruction queue (see figure 3B). Concurrently, the instruction located in the front of node(*.0)'s instruction queue is transmitted to node(*.1) and is stored in the front of node(*.1)'s instruction queue (see figure 3B). This process repeats for a total of 2^r steps, at which time the first instruction to be executed is transmitted from node(*.2^r-2) to node(*.2^r-1) (see figure 3C). Recall that each cycle contains 2^r nodes labeled (*.0).(*.1).(*.2). · · · .(*.2^r-1). where * denotes the cycle number.

After the first instruction is received at node(*.2^r-1). execution can commence. At step 2^r-1. the instructions are distributed within the cy.cle as shown in figure 3C and instruction execution proceeds as follows. First. each node within the cycle executes the instruction at the front of its instruction queue. Upon completion of this instruction. node(*.p) transmits the instruction at the rear of its instruction queue to node(*.(p+1)mod 2^r) (see figure 3D). Note that node(*.0) receives an instruction from the controller and that node(*.2^r-1) does not have a node above it to transmit to The instruction queue is then shifted forward one position (see figure 3E). overwriting the front of the queue. The front of the queue now contains the next instruction to be executed. The process repeats for all subsequent instructions. requiring only one instruction transfer between adjacent nodes per execution cycle. New instructions enter the cycle during successive phases through node(*.0) of each cycle.

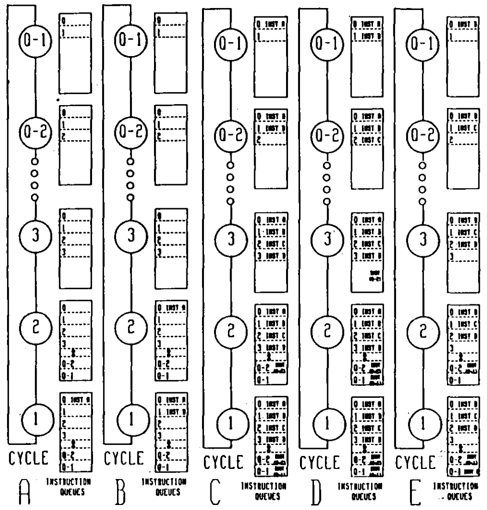


Figure 3: Instruction Broadcasting Example

Programming by Logical Networks

It is known that CCC can operate as a universal parallel machine[11]. It can simulate every reasonable parallel machine with only a small loss of time and with essentially the same number of processors[11]. For example. the CCC can emulate a hypercube with a O[$\log_2 n$] slowdown factor. In fact. it can emulate a large class of hypercube algorithms (Ascend and Descend type) within a constant slowdown factor[32]. Any fixed dimensional cube (eg. 2D mesh. 3D cube) can be emulated efficiently by a hypercube Consider. for example. the mesh network of figure 4. Each PE is identified by an ordered pair of its cartesian coordinates. Note that adjacent PEs differ by only one in any of the two coordinates. We determine the address of the node (to which each mesh PE is mapped) by applying Gray code to the mesh PE's coordinates. In this way adjacent PEs get mapped onto nodes whose addresses differ in a single bit position. These nodes therefore form a hypercube with the required adjacencies being the hypercube connections. These are easily realizable on the CCC. In this manner a mesh network can be

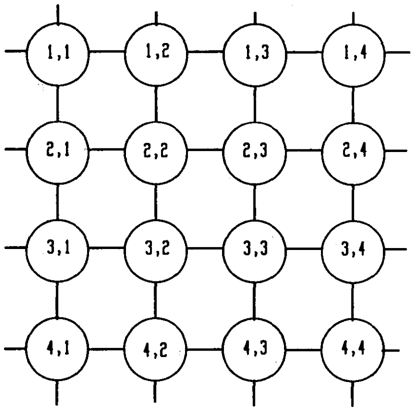


Figure 4: Logical Mesh Connected Network

efficiently realized on the CCC. Also, tree networks can be emulated by CCC within a O[$\log_2 n$] factor. Tree-like computations (such as counting) can be emulated within constant factor[3]. We therefore propose a parallel programming environment for the CCC based on the specification of logical networks of PEs.

A user may specify any logical network to execute algorithms written in a programming language similar to Concurrent Pascal. In the network specification, the user will specify a PE node type, define the interconnection pattern between nodes, describe data and control communications and finally the instruction set to be executed by the node. The instruction set will be restricted to a finite set of inter-node, as well as intra-node data movement and data path instructions. Initially, we intend to support the specification and programming of the CCC, Hypercube, Mesh, Tree, Pyramid, and Shuffle Exchange networks.

For example, consider the logical mesh connected network of figure 4. The network could be specified by defining the interconnections in terms of the cardinal directions such as North[PE(3,2)]←→South[PE(2,2)]. The program segment shown in figure 5 could then be specified for such a network. The program first specifies the network as an n by n mesh of PEs, each containing two integer type records. The next section of the program specifies the type of operations to be performed. If the four connections to a PE are denoted by the cardinal directions, this operation specifies that a PE is to sum and store the values stored in its north and west neighbors. This network specification and program is then mapped onto the CCC and its instruction set

By allowing programming via the specification of a logical network, the details of the physical CCC network remain hidden. This eliminates the need for programmers to provide information concerning the pipelining of instructions and data on the CCC. These details will be extracted from the logical network's specification and its associated instruction sequence. To further aid the user friendliness, we intend to provide a graphical interface for entering the network's connection pattern and program.

A compiler will take the network specification (and its program) and will generate a sequence of "MACRO" instructions that are supported by the underlying CCC. A MACRO instruction specifies the movement and processing of data via the CCC's sheaves, cycles, and nodes. To do such emulations, MACRO instructions such as BSHIFT, FSHIFT (forward and backward cyclic shift), LSHIFT (left lateral shift) RSHIFT (right lateral shift), and NO-OP instructions are necessary. These MACRO instructions are then mapped into sequences of CCC instructions.

```
network A: MESH n by n of

        record  X: integer;
                   Y: integer;
        end

        for i ϵ [1..n], j ϵ [1..m] do

                A[i,j]←A[i,j+1]+A[i-1,j]

end;
```

Figure 5- Sample program for a logical mesh network

If the logical network is not already specified in the compiler, the network and its program must be defined using MACRO instructions of the type listed above (e.g. BSHIFT, FSHIFT, LSHIFT and RSHIFT). This allows a user to emulate an arbitrary network. However, a programmer must fully specify the operations, in terms of the MACRO instructions, necessary to emulate the logical network.

The Processing Element (PE)

In this section, the PE instruction set and architecture are detailed. We start with a brief description of the number representation used. We then present an instruction set based on the RISC concept[14,20,28,35], and relate this instruction set to the instruction execution cycle of a conventional processor. An example of a typical execution sequence is shown. Details of implementation of conditional blocks have also been given. This is followed by a description of the hardware elements in the PE. In conclusion, we briefly give some timing estimates for performance evaluation.

Number Representation

We have adopted a very simple number representation for the processing elements. The word length is 16 bits, with negative numbers in two's complement notation. Integer data can be represented by any number of words, depending on the size of the number. For floating point data, the mantissa is 32 bits long organized as two 16-bit words. The exponent is represented by a separate 16-bit word. This representation enables floating point operations to be implemented as a series of integer operations and avoids the task of field extraction. While this system is simple, it allows much faster execution of floating point operations with our instruction set than would be possible with a standard floating point representation.

The PE Instruction Set

A normal data path cycle on a conventional processor, as shown in figure 6 consists of the following phases: 1) Operand Fetch; 2) ALU Operation; 3) Result Store. The aggregate of these phases is denoted as a major cycle. A processor implementing a major cycle must contain a certain amount of control logic to schedule these events, breaking this major cycle into a series of operations we term minor cycles (see figure 6). Note that a significant amount of information must also be provided in an instruction representing a major cycle (i.e. source operand addresses, opcode, and destination address). Providing such information to a processor efficiently will require a large number of bits even with encoding and will also require a complex control circuit. To have a processor that is as simple as possible, we provide instructions as minor cycles. Providing instructions as minor cycles allows great flexibility in controlling the PE's actions. Such flexibility will be needed if the emulation of other networks is to be done efficiently.

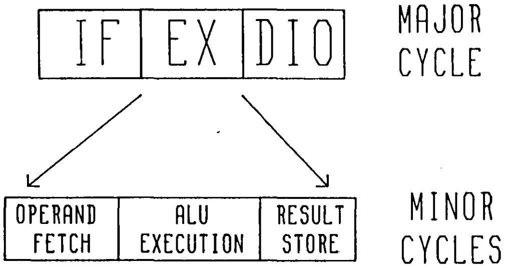


Figure 6- Conventional Processor Cycle

The decoding for each of these minor cycles is going to be simple and can be done in parallel with execution i.e. instruction i+1 is decoded while instruction i is being executed. (Note that this implies an extra queue position per processor for the IQ scheme discussed earlier). Since an SIMD machine does not allow branching within program memory, this decode/ execution pipelining does not introduce any constraints. It is completely transparent to the user in that the user is aware of one minor cycle being executed every clock cycle.

The proposed instruction set is shown in figure 7 (and the associated data path is shown in figure 14). The instruction set can be split into the following three categories: 1) Load-Store; 2) ALU; 3) Data Transfer. The complete instruction set has been encoded into a format 12 bits long. Details of the encoding scheme can be found in [7].

The Load-Store type instructions control the movement of data between the register file and the ALU. To facilitate floating point (32-bit) operations, separate load-store of the high and low words (16-bits) have been provided. An option is also provided to load an 8-bit immediate data into the ALU.inA register using the LLIT (Load LITeral) instruction.

ALU instructions are those that implement data modifications within the PE. One input to the ALU (R1) is always provided by the register ALU.inA. The other ALU input (R2) can be either ALU.inB or ALU.out. For the unary ALU operators, the input (Source) can be either R1 or R2. The result of the ALU operation (Dest) can again be either ALU.inB or ALU.out. The flexibility in choosing the sources and destination of the ALU operations helps in handling of floating point operations. For the same reason, a 16-bit as well as a 32-bit mode has been provided for the ALU. The ALU also has a set of four condition flags- C:Carry, V:Overflow, Z:Zero and S:Sign. These flags are set or reset based upon the result of the ALU operation. The status of these flags can be used to conditionally execute instructions as explained in a later section. It is to be noted that instead of providing a separate Shift instruction, an option is available to cyclically shift (left/right, with/without carry) the results of some ALU instructions (PASS,ADD,SUB). The shift is performed immediately before the results are sent to their destination eg. an add and shift can be done in one cycle.

Data transfer instructions control the PE's three connections (i.e. Forward, Backward, Lateral). These instructions simply schedule data transfers between processors.

A sample of a typical sequence of instructions implementing a data path major cycle is shown in figure 8. The normal data path cycle starts with the loading of the ALU's input buffers with two operands, followed by the execution of the operation within the ALU. The result of this operation is then stored in the register file. Note that we show the two operands being read in sequence rather than in parallel. This is because the PE's register file has only one port, as described in the next section.

The overhead associated with the transfer of instructions between nodes can be eliminated as follows. Since the operations of the instruction decoder, datapath and the I/O circuits are disjoint, the three phases of instruction decode, execution and transfer can be overlapped as shown in figure 9. This pipelining eliminates the delay in instruction distribution in all but the initial phases of the CCC's operation.

The data transfer between nodes by the FWDTR, BWDTR and LATTR instructions can also be pipelined. Note that the data transfer instructions are disjoint in operation from all but the LOAD/STORE type instructions. Provided that the effects of LOAD/STORE instructions remain orthogonal to the effects of the data transfer instructions, the data transfer instructions can be executed concurrently with the remaining instructions. The Controller can ensure that the data transfers and LOAD/STOREs are disjoint by prohibiting access of an I/O register while it is transmitting or receiving data. To implement this pipelining, two additional bits of the instruction field will be used to specify the state of the I/O operations (i.e. 00:FWDTR, 01:BWDTR, 10:LATTR, 11:no transfer). We will therefore have two instructions executing concurrently in one cycle.

Conditional Instructions

When implementing conditional blocks of code in an SIMD machine, as found within an IF-THEN-ELSE statement, the instruction streams must remain identical in all PEs. Standard implementations can not be applied because bypassing blocks of instructions via branching in the instruction memory is not allowed. The processors must be provided with each block of instructions for all possible outcomes of a conditional statement, executing only the block that corresponds to the satisfied value of the condition.

To illustrate, consider the IF-THEN-ELSE block shown in figure 10. When providing a PE in an SIMD network with instructions that implement this program, both the block of instructions for the IF segment (B1 etc) and the ELSE segment (B2) must be provided to the PE. If this is not done, the SIMD network controller will be required to monitor each processor to determine the outcome of the conditional test, and will then have to provide each individual processor with a unique instruction stream. This is not a feasible scheme, especially if there are a large number of PEs (as in our case)

In order to have the same instruction stream for all PEs, the condition codes of an earlier instruction must be able to affect the execution of future instructions. A scheme for implementing this structure is as follows (see figure 11). We have a one-bit wide and N-word deep stack called the Condition Stack (CS). Any instruction is enabled to execute if and only if all the 1-bit words of the CS are set i.e. the execution Enable signal is derived by a logical AND of all the

PROCESSOR ELEMENT INSTRUCTION SET		
Instruction	Operation	Comments
Load-Store		
LDA L Rs	ALU.inA L ← Rs	Load low 16 bits of ALU.inA with RF(Rs)
LDA H Rs	ALU.inA H ← Rs	Load high 16 bits of ALU.inA with RF(Rs)
LDB L Rs	ALU.inB L ← Rs	Load low 16 bits of ALU.inB with RF(Rs)
LDB H Rs	ALU.inB H ← Rs	Load high 16 bits of ALU.inB with RF(Rs)
STR L Rd	RF(Rd) ← ALU.out L	Store low 16 bits of ALU.out in RF(Rd)
STR H Rd	RF(Rd) ← ALU.out H	Store high 16 bits of ALU.out in RF(Rd)
LLIT L	ALU.inA ← Literal	Load ALU.inA bits 0-7 with literal from the IQ.
LLIT H	ALU.inA ← Literal	Load ALU.inA bits 8-15 with literal from the IQ.
ALU ‡		
NOP	none	No operation- nothing happens
PASS†	Dest ← Source	Pass the selected operand to the ALU Destination.
ADD†	Dest ← R1 + R2	Integer addition
ADDC	Dest ← R1 + R2 + Carry	Integer addition with carry
SUB†	Dest ← R1 - R2	2's complement subtraction
SUBC	Dest ← R1 - R2 - $\overline{\text{Carry}}$	2's complement subtraction with borrow
AND	Dest ← R1 ∩ R2	Logical And
OR	Dest ← R1 ∪ R2	Logical Or
XOR	Dest ← R1 ⊕ R2	Logical Exor
COMP	Dest ← $\overline{\text{Source}}$	Logical (1's) complement
INC	Dest ← Source + 1	Increment
DEC	Dest ← Source - 1	Decrement
Data transfer		
FWDTR	BRin[*.(p+1)mod2^r] ←FRout[*.p]	[ForWarD TRansfer] Data contained in the forward output register of node (*.p) is transferred to the backward input register of node (*.(p+1)mod 2^r)
BWDTR	FRin[*.(p-1)mod2^r] ←BRout[*.p]	[BackWarD TRansfer] Data contained in the backward output register of node (*.p) is transferred to the forward input register of node (*.(p-1)mod 2^r).
LATTR	LRin[*+2^ϕ.p]←LRout[*.p] LRin[*.p]←LRout[*+2^ϕ.p]	[LATeral TRansfer] Simultaneous data transfer between PEs connected by a lateral connection.
‡ ALU instructions will set the ALU's condition codes (C:carry, V:overflow, Z:zero, S:sign) † Optional shift of results (shift right,left,with carry, without carry)		

Figure 7 Processing Element Instruction Set

LDA Rs	LDB Rs	ALU op	STR Rd

Figure 8 - Instructions implementing a data path cycle

INST I EXEC	INST I+1 EXEC	INST I+2 EXEC
INST I+1 DEC	INST I+2 DEC	INST I+3 DEC
INST I+2 TRANS	INST I+3 TRANS	INST I+4 TRANS

Figure 9 Instruction Pipelining

```
if (cond1) then          ....(a)
  begin
  B1
  if (cond2) then        ....(b)
    begin
    C1
    if (cond3) then      ....(c)
      D1
    else                 ....(d)
      D2
    endif                ....(e)
    end
  else                   ....(f)
  C2
  endif                  ....(g)
  end
else                     ....(h)
  B2
endif -                  ....(i)
```

Figure 10- Sample Conditional Program

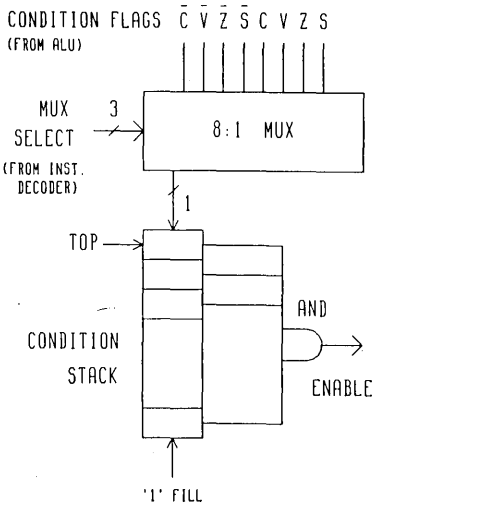


Figure 11 : Hardware for Conditional Evaluation

CS bits. As mentioned earlier, the ALU has 4 condition flags (C.V.Z.S) and their status can be used to execute an instruction conditionally. We provide an 8-to-1 multiplexer to select one of these (or their negated versions) and this selected status can then be pushed on the CS.

Consider the implementation of a simple IF-THEN-ELSE block using the above scheme. Let all CS bits be initially set. Upon encountering the if (condition) part, an ALU instruction is performed to evaluate the condition and the appropriate flag is selected and pushed on the CS. If the Enable signal (derived by ANDing all CS bits) is true, the THEN block is executed, otherwise it is skipped. When we reach the ELSE block, the top of CS is simply complemented. Note that if this bit was initially zero (i.e. the if-condition was false), it will now get set enabling the execution of the ELSE block. Conversely, if the bit was initially set (i.e. the if-condition was true), it will now get reset disabling the ELSE block. In this manner only one of the two- THEN or ELSE blocks - is executed even though both blocks are part of the instruction stream. Upon reaching the end of the conditional block, the top bit is popped off the CS and it no longer affects the instruction execution. To summarize, for every conditional block, we select and push for IF, complement for ELSE and pop for ENDif.

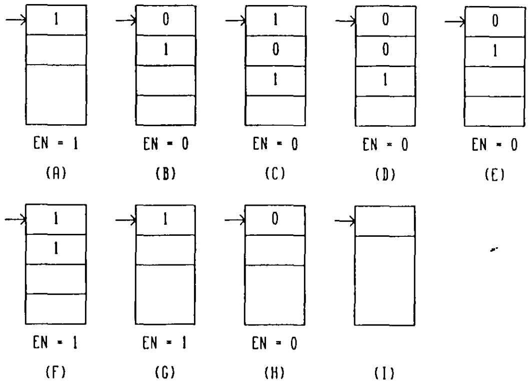


Figure 12 : Snapshots of the CS for sample program

To illustrate use of the above scheme for a Nested conditional structure consider the program given in figure 10. Snapshots of the CS at different points in the program (labelled a, b, etc.) are also given in figure 12. At (a), cond1 is selected and pushed on CS. Suppose it is true. The Enable signal is therefore true and B1 is executed. At (b) cond2 is selected and pushed. Let us assume it is false. The Enable signal is now false and execution of C1 is disabled. At (c) we select and push cond3 (true) on CS. (Note that this selection and push is done in spite of execution being disabled i.e it is not affected by the status of the Enable signal). Whatever be the status of cond3 the Enable stays false because of the false cond2. (For that reason it does not matter whether or not the test for cond3 is even valid). D1 is therefore disabled. At (d), we complement top of CS, the Enable remains false and D2 also is not executed. At (e) the ENDif for cond3 occurs and we pop CS. At (f) we again complement top of CS. This time the Enable becomes true and we execute D2- the ELSE part of cond2. At (g) the CS is popped and we then complement top of CS at (h). The Enable signal now becomes false and execution for the ELSE part of cond1 is disabled. B2 is therefore not executed. Upon reaching (i) CS is popped again and the scope of cond1 ends. Note how pushing/ complementing/ popping for corresponding IF/ELSE/ENDif maintains the correct Enable status inside a nested conditional structure. The maximum allowable nesting depth is determined by the depth of the CS.

A simple format for the conditional control field, which can implement the above operations, is given in figure 13 Note that this control field will be needed only with the ALU instructions. Rest of the instructions will simply use the Enable signal provided by the CS.

The Processor Architecture

The PE architecture is shown in figure 14. It consists of a register file (RF), a 32/16 bit arithmetic logic unit (ALU), a shifter, an exponent register, an address register, instruction queue (IQ), I/O Buffers and a simple instruction

Bit 1=0 No condition selected		Bit 1=1 Condition selected & pushed	
Bits 2.3.4		Bits 2.3.4	
100	: no operation	000 (100)	: $\overline{C}$ (C) selected
101	: complement	001 (101)	: $\overline{V}$ (V) selected
110	: pop	010 (110)	: $\overline{Z}$ (Z) selected
		011 (111)	: $\overline{S}$ (S) selected

Figure 13- Conditional Control Field

decoder. The functions of each of these blocks are summarized in Table 1.

The PE will be implemented as follows. The register file is organized as 256 16-bit words. The lowest 8 bits of the IQ are always input to the RF address decoders. Therefore address decoding for a potential memory instruction always takes place simultaneously with the instruction decoding. If the instruction is indeed decoded to be a memory operation then the appropriate word-line is asserted during the execution phase of the instruction. This is done to save address decoding delay during the actual execution of Load-Store instructions.

For the ALU, there are two 32-bit input registers (ALU.inA & ALU.inB) and one 32-bit accumulator (ALU.out). The ALU is capable of performing arithmetic operations on 16-bit as well as 32-bit data. Even though the PE's datapath is 16-bit wide, it was decided to have the 32-bit option for the ALU to speed up floating point operations. Integer operations (16-bit) are performed by using the lower half of the ALU's datapath.

The Shifter is capable of shifting a 32-bit word right or left by one bit and is used as a post-processor to the ALU. The Exponent register is used to hold the exponent during floating point operations and can be addressed as part of the RF. It is also used as an up/down counter and is automatically incremented (decremented) whenever the Shifter does a shift right (left) operation. The (Φ.p) register consists of 10 bits and contains the PE's address. This register can also be addressed as a special (read-only) RF position with the six high order bits set to zero. This way a processor can be selectively masked out based on its location within the CCC network. For the I/O buffers, two 16-bit buffers will be used for each port (i.e. one for input, one for output). These buffers are addressable as registers in the register file and the input buffers can also be loaded from a port (i.e. F.B or L). Finally, the instruction decoder will be a small finite state machine and will be implemented with a PLA and random logic. The hardware used for conditional evaluation (the Condition Stack and the Enable logic) will form part of the decoder logic.

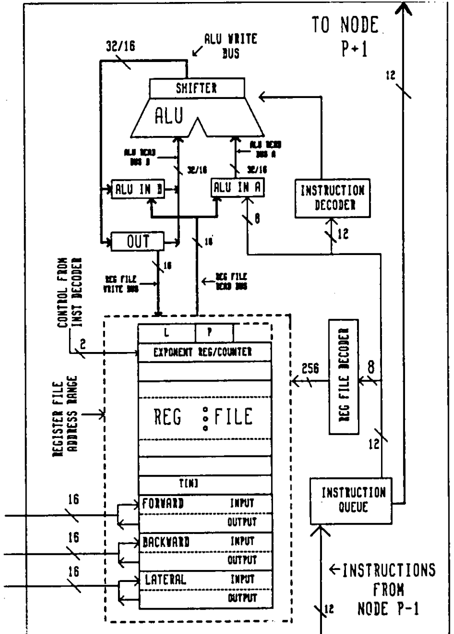


Figure 14- Processor Architecture

<u>Processor Timing</u>

The basic timing scheme for each PE consists of pipelining the three operations of instruction decode, execution and transfer per clock cycle. As discussed earlier, the instructions have been provided in the form of minor cycles leading to simple decode logic. Also, since instruction transfer takes place between adjoining processors, its timing is not critical in determining the clock rate. The determining factor for the maximum achievable clock rate is the longest delay in the datapath during instruction execution. An initial estimate of the datapath timing for the given processor architecture has led to a 50ns. clock cycle. All instructions listed in the instruction set take one cycle time to execute. This gives a performance figure of 20 Mips per processor. Note however that our instructions are minor cycles. Since three to four minor cycles are equivalent to a conventional processor cycle, one-third to a quarter of the above rate would be a reasonable figure of merit for comparison.

The PE architecture does not have any special hardware unit for performing floating point operations. However by using the Exponent register, the Shifter the 32-bit mode of the ALU and the given instruction set, we have been able to implement one floating point operation every

Table 1- PE Function Block Descriptions

Section	Function
ALU	The Arithmetic Logic Unit executes data path instructions of the type add, subtract, not etc. (16 or32 bit ALU)
Shifter	The Shifter shifts the data (16 or 32 bits) right or left by one bit This can be done with or without carry wrap-around.
Exponent Register	The Exponent Register can be accessed as a register file position or it can be used as an up/down counter. A shift right (left) operation by the shifter is accompanied by an auto-increment(decrement).
Register file	The Register File contains the local memory. This is a single port read/write RF, organized as 16 bits by 239 words. Note that its address space (0:255) includes scratch area and special registers.
I-O Buffers	There are three I-O buffers denoted FORWARD, BACKWARD and LATERAL. Each contains two registers, one for input and the other for output. Each I-O register can be accessed as a either a register position or from the port connection.
Instruction Queue	The instruction queue holds the instructions to be executed by the PE.
Instruction decoder	A simple finite state machine, implemented with a PLA.
(Φ.p) Register	This register contains the node's (Φ.p) 10 bit address. This register is set when the network is configured (i.e. it is a ROM register).

5μs i.e. each PE is capable of performing at a rate of 0.2 MFlops. This leads to a figure of 102.4 MFlops for the entire network having 512 PEs.

Implementation

In our implementation, 512 PEs will be connected in a CCC network in wafer scale integration. A floor plan of the PE die is shown in figure 15. Table 2 gives the initial area estimates for each section of the PE's circuits for a 2μm scalable CMOS ($\lambda = 1\mu$m) technology. The PE will contain 256 addressable memory locations (RF). The address space is partioned as follows: 239 data registers, 8 general purpose registers (i.e. scratch memory, constants etc.), 6 positions for the I/O registers, one for the (Φ.p) address (a read only location), one position for the exponent register and one constant register set to 0. To reduce the read/write delay caused by long word lines, the register file has been split into four equal parts of 64 register positions each. Splitting the register file also improves the aspect ratio of the die. Each of the four register file banks will be addressed by a separate decoder. Since the address field is eight bits, the two most significant bits will select one of these four banks while the remaining six bits will select a register from within it. Note that 0.25mm has been added to the perimeter of the die for the I/O pads, routing and random logic. The PE die size will be approximately 3.32mm X 2.2mm.

Following the Discretionary wiring approach [25] discussed earlier, dice will be first fabricated on a wafer and the functioning die sites will be then connected with discretionary wiring. Assuming that a 6 inch diameter ($\approx$15 cm) wafer is used, there will be only 100cm^2 of usable area equal to that of the square inscribed in the wafer. With a die size of 8.964mm^2, there will be 1114 possible die sites. With a 50% yield of dice, there will be approximately 557 functioning dice after fabrication, providing more than the minimum requirement of 512.

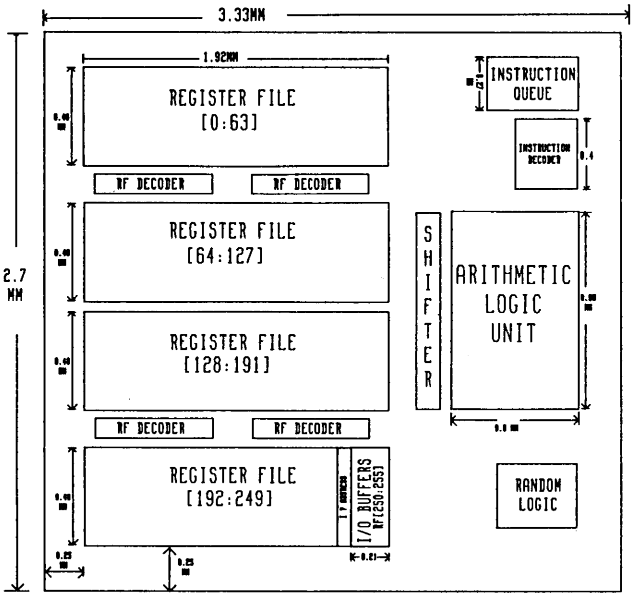


Figure 15- Processor Element Floor Plan

Table 2- Initial PE area estimates

PE Circuit	Unit Cell Dimensions $\lambda x \lambda$	Cell Organization Row x Column	Overall Dimensions $\lambda x \lambda$	Area mm^2	% Total Area
Register File	-	-	-	-	-
RF(0:239)	30x30	16x240	480x7200	3.456	38.55
RF (i.e. 4) Decoders	-	-	75x1000	0.075	0.83
ALU	30x500	32x1	960x500	0.720	8.03
Shifter	30x50	32x1	960x50	0.048	0.54
Exponent Register	30x50	16x1	480x50	0.024	0.27
Φ.p register	30x20	10x1	300x20	0.006	0.07
I-O registers	30x35	16x6	480x210	0.101	1.13
Instruction Queue	30x30	12x9	360x270	0.097	1.08
Instruction Decoder	400x400	1x1	400x400	0.16	1.78
Routing & I-O pads†	-	-	-	2.760	30.79
Die size	-	-	2700x3320	8.964	100

† This refers to the 0.25mm perimeter of the PE die

Alternatively, implementation of this network with the WTM[10,37] is as follows. A custom 6 inch interconnection wafer will be designed for maximum packing. There will be space for 512 processor chip, along with a number of specical I/O chips. The chips will be bonded to the wafer using a solder bump technology as described in [18].

Packages for both the approaches listed above are currently under development at Rensselaer Polytechnic Institute's Center for Integrated Electronics.

Conclusion

The preceeding sections have presented a feasible parallel computer with a large number of processing elements for wafer scale integration. Use of a RISC based architecture for the processing element allows great flexibility in using the computer. The architecture is both simple to construct and flexible enough to permit the emulation of logical networks in an efficient manner. The instruction set is designed for pipelined execution of instructions. Processor allocation and scheduling is done by a dedicated controller. Programming can be done by users unfamiliar with the internal architecture of the computer. Many practical problems can be solved efficiently with this computer and it effective computing power is estimated at approximatly 6MIPS per processor.

Future directions for this project include working on the design details of each PE. The PE is currently being modelled and simulated using the N.2 Simulator. Following verification of the PE architecture and instruction set, each of the modules (ALU, RF etc.) will be designed and simulated at the gate level. The transistor level design and simulations for all circuit elements will be done next. Finally, layouts for the complete processor will be made and verified.

Acknowledgments

The authors wish to thank Randy Steinvorth for the detailed explinations of the wafer transmission module technology and Austin Lobo for his sugestions during the early stages of this project. The comments and financial support

from Digital Equipment Corporation were also greatly appreciated.

REFERENCES

[1] Atallah. Mikhail. S. Rao Kosaraju. "Graph problems on a mesh-connected processor array". 1982 ACM Symposium on Theory of Computing pp. 345-353.

[2] Bergendahl. A.S.. J.F. McDonald. R.H. Steinvorth. G.F. Taylor. "Thick Film Micro-Strip Transmisson Line Interconnect for Wafer Scale Integration". Proceedings of the ECS Meeting on VLSI 1985.

[3] Carlson. David A.. "Parallel Processing of Tree-Like Computations" Proceedings of the 4th International Conference on Distributed Computer Systems. 1984. pp.192-200.

[4] Chapman. G.H.. et al.. "A Laser Linking Process for Restructurable VLSI". 1982 OSA/IEEE Confrence on Lasers and Electro Optics Technical Digest. pp. 60-62.

[5] Chesley. G.D.. "Main Memory Wafer Scale Integration". VLSI Design. March 1985. pp. 54-58.

[6] Chung.M.J.. E.J. Toy. A.A. Lobo. "A parallel Computer Based on Cube Connected Cycles (An Abstract)". Proceedings of the Army Research Office Workshop on Future Directions in Computer Archecture and Software. May 5-7. 1986. Charleston. SC.

[7] Chung.M.J.. et. al. "A parallel Computer Based on Cube Connected Cycles". RPI-CIE Technical Report. March 1986. Rensselaer Polytechnic Institute's Center for Integrated Electronics. Troy. New York.

[8] Daughton. J.M.. B.K. Koeneman. "Cost Trade Offs in Wafer Scale Integration". Proceedings of the IEEE 1984 ICCD. pp. 88-94.

[9] Donlan. Lt. B.J.. J.F. McDonald. G.F. Taylor. R.H. Steinvorth. A.S. Bergendahl. "Computer Aided Design and Fabrication for Wafer Scale Integration". VLSI Design. Vol. VI. No.4. April 1985.

[10] Donlan. Lt. B.J.. J.F. McDonald. R.H. Steinvorth. M.K. Dhodhi. G.F. Taylor. A.S. Bergendahl. "The Wafer Transmission Module". VLSI Systems Design. January 1986. pp. 54-58 & 88-90.

[11] Gail. Zvi. W.J. Paul. "An Efficient General Purpose Parallel Computer". Journal of the ACM. Vol. 30. No.2. April 1983. pp.336-387.

[12] Garverick. S.L.. E.A. Pierce. "A Single Wafer 16-Point 16Mhz FFT Processor". Proceedings of the 1983 IEEE Custom Integrated Circuits Conference. pp. 244-248

[13] Graber. W.S.. et al. "Reconfiguring Semi-custom ICs Using Laser Microchemical Techniques". pp. 453-456.

[14] Hennessay. J.. et al.. "Design of a High Performance VLSI Processor". Proceedings of the Third Caltec Conference on VLSI. 1983.

[15] Hewitt. C.L.. "VLSIC Packaginging for High Reliability Applications". Proceedings of the 1984 IEEE Custom Integrated Circuits Conference. pp. 135-141.

[16] Holems. R.E.. "VLSI Packageing and Interconnect Technologies". Proceedings of the 1983 IEEE Custom Integrated Circuits Conference. pp. 132-134.

[17] Hsia. Y.. G.C.C. Chang. F.D. Erwin. "Adaptive Wafer Scale Integration". Proceedings of the 11th Conference (1979 International) on Solid State Devices. Tokyo. 1979. Japanese Journal of Applied Physics. Vol. 19. Suppleement 19-1. pp. 193-202.

[18] Huang. C. et al.. "Silicon Packaging- A New Packaging Technique". Proceedings of the 1983 IEEE Custom Integrated Circuits Conference. pp. 142-146

[19] Johnson. R.R.. "The Significance of Wafer Scale Integration in Computer Design". Proceedings of the 1984 IEEE Custom Integrated Circuits Conference. pp. 101-105.

[20] Katevenis. M.. "Reduced Instruction Set Computers for VLSI" Doctoral Dissertation. U.C. Berkeley. October 1983.

[21] Kung. H.T.. C.E. Leiserson. "Algorithms for VLSI Processor arrays". in "Introduction to VLSI Systems". sec 8.3 pp 271-292. C.A. Mead & L. Conway. Addison-Wesley. Reading. MA. 1980.

[22] Lathrop. J.W.. R.S. Clark. J.E. Hull. R.M. Jennings. "A Discretionary Wiring System as the Interface Between Design Automation and Semiconductor Array Manufacture". Proceedings of the IEEE. November. 1967

[23] Mangir. T.E.. "Interconnect Technology Issues for testing and Reconfiguration of Wafer Scale Integration". Proceedings of the IEEE ICCD. 1984 pp. 127-131.

[24] Mangir. T.E.. "Sources of Failures and Yield improvement for VLSI and Restructurable Interconnects for RVLSI and WSI: Part II: Restructurable Interconnects for RVLSI and WSI" Proceedings of the IEEE. Vol. 72. No. 12. December 1984. pp. 1687-1694.

[25] McDonald. JF. E.H. Rogers. K. Rose. A.J. Steckel. "The Trials of Wafer Scale Integration" IEEE Spectrum. October. 1984.

[26] Nath. D. S.N. Maheswari. P.C.P Bhatt. "Efficient VLSI Networks for Parallel Processing Based on Orthogonal Trees". IEEE Transactions on Computers. Vol. C-32. No. 6. June 1983. pp. 569-581.

[27] Neugebauer. C.A.. "Comparison of VLSI Packaging Approaches to Wafer Scale Integration". Proceedings of the IEEE 1985 Custom Integrated Circuits Conference. pp. 32-37.

[28] Patterson. D.. C Sequin. "A VLSI RISC". IEEE Computer. September 1982.

[29] Pease. Marshal C.. "An Adaptation of the Fast Fourier Transform for Parallel Processing". J. ACM. Vol. 15. April 1968. pp. 252-264.

[30] Pease. Marshal C.. "The Indirect Binary n-cube Microprocessor Array". IEEE Transactions on Computers. Vol. C-26. No. 5. May 1977. pp. 458-473.

[31] Peltzer. D.L.. "Wafer Sacle Integration: The Limits of VLSI?". VLSI Design. September 1983. pp. 244-248.

[32] Preperata. F.P.. J. Vuillemin. "The Cube Connected Cycles : A Versatile Network For Parallel Computation". Comm. of the ACM. Vol 24. No 5. May 1981

[33] Raffel. J.I.. A.H. Anderson. G.H. Chapman. K.H. Konkle. B Matur. A.M. Soares. P.W. Wyatt. "A Wafer-Scale Digital Integrator". Proceedings of the IEEE ICCD. 1984.

[34] Shaver. D.C.. R.W. Mountain. D.J. Silversmith. "Electron-Beam Programmable 128k-bit Wafer Scale EPROM". IEEE Electron Device Letters. Vol. EDL-4. No. 5. pp.153-155. May 1985.

[35] Sherburne. R.W. Jr. "Processor Design Tradeoffs in VLSI". Doctoral Dissertation. U.C. Berkeley. April 1984.

[36] Stone. H.S.. "Parallel Processing and the Perfect Shuffle". IEEE Transactions on Computers. Vol C-20. Feburary 1971. pp. 153-161.

[37] Taylor. G.F.. Lt. B.J. Donalan. J.F. McDonald. A.S. Bergendahl. R.H. Steinvorth. "The Wafer Transmission Module- Wafer Scale Integration Packaging". Procedings of the IEEE 1985 Custom Integrated Circuits Conference. pp. 55-58.

[38] Taylor. G.F.. "A Two-Dimentional Fast Fourier Transform Processor Suitable for Wafer Scale Integration". Doctor of Engineering Thesis. Rensselaer Polytechnic Institute. Troy. New York. October 1985.

[39] Wagner. David. "The Arcitecture of a Boolean Vector Machine". Depatment of Computer Science Technical Report CS-1982-8. December. 1980. (Revised October. 1981).Duke University. Durham. NC.

MUPPET - A Programming Environment for Message-Based Multiprocessors

H. Muehlenbein, F. Limburger, S. Streitz, S. Warhaut

GMD - Gesellschaft für Mathematik und Datenverarbeitung mbH
P.O. 1240, D-5205 St. Augustin 1

Abstract:

MUPPET is a problem solving environment for scientific computing. It consists of four areas - concurrent languages, programming environments, application environments andman-machine interfaces.
The programming paradigm of MUPPET is based on abstract machines and transformations between them. This paradigm allows the development of programs which are portable among different machines.

In this paper we discuss the programming paradigm and give a brief introduction to the language CONCURRENT MODULA-2 and the simulation system.

MUPPET is being developed as part of the German supercomputer project SUPRENUM.

1. Introduction

The new generation of scientific supercomputers depends more and more on parallel processing. Supercomputers can be divided into two groups

- multiprocessors with shared memory
- multiprocessors with local memory

The latter systems are a more radical departure from the sequential von Neumann type of machines. They can employ more than one thousand individual processors giving peak rates of more than 10 Gigaflops. Most notable is the recently announced T-series from Floating Point Systems. This incredible hardware speed has to be matched by adequate "super" software. This super software should support

- the fast development of programs
- the efficient use of the supercomputer
- the reusability of software

MUPPET (a multiprocessor programming environment) is a long term effort to address these issues in the area of scientific computing. MUPPET is designed for multiprocessors with local memory. It is part of the German supercomputer project SUPRENUM [GiMu86] , the goal forwhich is to build a supercomputer of more than one Gigoflop.

MUPPET is not simply a programming environment, but it is designed as a problem solving environment for scientific computing.

A problem solving environment can be divided into at least four areas ([FOR86])

- CONCURRENT LANGUAGES
- PROGRAMMING ENVIRONMENT
  syntax editor, program templates, library
- APPLICATION ENVIRONMENTS
  e.g. language for the solution of partial differential equations, multigrid expert systems
- MAN-MACHINE INTERFACE
  graphics, algorithm animation

Concurrent languages are necessary for driving the multiprocessors efficiently. The programming environment tries to support the programmer. Its goal is to make programming significantly easier, more reliable and cost effective by reusing previous code and programming experience to the greatest extent possible.

The application environments support the end user who does not want to do low level programming. Application environments should include knowledge about the application domains and application specific technical languages.
Most of the MUPPET tools will run on a high-performance scientific workstation with a graphic oriented man-machine interface.

The standard tools of the programming and application environment (a language-based editor supporting programming in the small, a fragment library supporting incomplete specifications and an interpreter) will be generated by the programming system generator PSG, which is described elsewhere [BSN 85]. This approach makes rapid prototyping of environments possible.

In this paper we concentrate on the specific problems encountered in parallel programming. MUPPET is being developed in close cooperation with the PIE project, whose goal is to develop a programming environment for a shared memory system [SE RU85].

The outline of the paper is as follows:
In chapter two we introduce the programming paradigm of MUPPET, in chapter three we discuss an implementation tradeoff by using the randomization method. The programming paradigm is explained in more detail in chapter four. The examples are downloading and matrix multiplication. The next two chapters describe the language CONCURRENT MODULA-2 and the MUPPET simulation system. In the last chapter we discuss some

CH2345-7/86/0000/0336$01.00 © 1986 IEEE

problems encountered an lessons learned.

2. Parallel programming paradigms and abstract machines

Programming is the conversion of an abstract (machine independent) algorithm into a program that can be run on a particular computer. A programming paradigm can be seen as a way of approaching a programming problem and thereby restricting the solution set.

Following Browne [BROW 86] we define a parallel computation as a graph where each node is the binding of an action to a data object and the edges are the dependency relationships between the computations executed at the nodes. In this paper we use a simpler model to describe a parallel program on a parallel computer ([SNY84]). The model consists of

- a graph G = (P, E) whose node set P represents processes and whose edge set E represents the communication scheme
- a similar graph H = (N, L) representing the parallel computer (nodes N and links L)
- a function π: P$\rightarrow$N mapping processes to processors
- a process set P describing the computational activity
- a communication model

MUPPET is based on this model but it is allowed that processes can be created dynamically. This model can be formulated as an abstract machine, which we call LAM-local memory abstract machine. LAM is defined as follows

(I) LAM consists of disjoint processes (no shared variables)

(II) Processes can create other processes and can terminate

(III) Communication is done by asynchronous message passing between partners (processes which know each other)

In [FIL84 p.344] this abstract machine has been advocated also as a basis for coordinated computing.

LAM can handle arbitrary communication schemes. More specific abstract machines can be defined by adding additional constraints. This will be explained by the ring local memory abstract machine-RLAM. RLAM is defined informally by

(IV) In RLAM every process has exactly two communication partners, called the neighbours. Communication is restricted to neighbours only

Similarly other abstract machines can be defined like the tree, the lattice, the hypercubes and the direct interconnected abstract machines.

Ultimately these abstract machines have to be mapped onto real machines of a given connection scheme, e.g. lattice, hypercube, butterfly. Examples are shown in Figure 1.

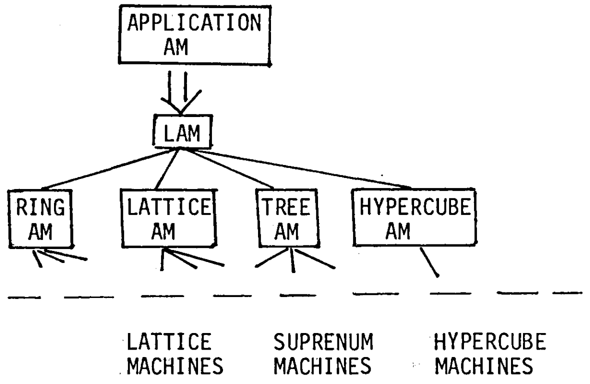


Figure 1: Abstract machines

The programming task can now be characterized as follows:The problem is formulated in an application oriented abstract machine, then it is transformed (manually and/or automatic) to a LAM implemention and may be to a more specific abstract machine implementation. This machine will then be mapped onto the topology of the real machines. We believe that this layering defines clear interfaces, which can be used to create portable and efficient programs.
The programmer has access to every layer. If the given problem has a regular communication scheme, e.g. a 2-D lattice structure, the programmer can use the 2-D lattice abstract machine directly.

The mapping of specific abstract machines to the physical machines can often be done optimally. We take as an example the mapping of a ring onto the SUPRENUM machine.
The SUPRENUM architecture can be characterized as a 2-D lattice of ring buses [GIMU86]. The straight forward mapping uses only the row or the column ring buses. The optimal mapping is the familiar Hilbert curve, which uses the communication network more equally.

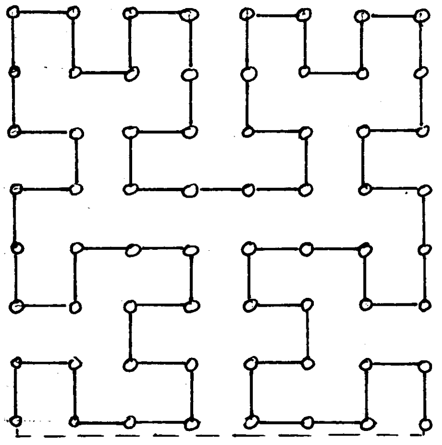


Figure 2: Mapping RLAM onto the 8 x 8 SUPRENUM machine

3. Tradeoffs for the implementation of abstract machines

Transformations of abstract machines should be guided by a goal. In order to define the goal we have to introduce a metric. In this paper we use a very simple one.
Let G1 and G2 be two graphs representing abstract machines. G1 shall be mapped onto G2. We define the Size S (G1, G2) to be

(1) S = max.distance in G2 between communicating partners of G1

The goal of the transformations is to reduce the size. Ultimately the size should be one and the resulting communication scheme should be mapped to the connection scheme of the real machine so that the size remains one in this topology.
If we end up with a size greater than one, the real system has to provide some kind of routing. Otherwise the communication scheme has to be enhanced by intermediate processes which handle the routing of messages and reduce the size to one.

This observation shows the importance of a routing service. The implementation of an efficient routing service depends on the topology of the machine and can be done by different methods.
It has been shown that random routing is a good heuristic for topologies with a rich interconnection structure (i.e. there exist many different routes between nodes). In particular, it has been proven for hypercubes and similar topologies that random routing gives a nearly optimal communication time for many interesting algorithms [ULL84 pp.232].

On these machines therefore it makes sense to provide the LAM abstract machine directly by an operating system. The operating system would use randomization and not transformations of abstract machines. It cannot be shown in general which of the two methods is more efficient. First studies on a 64-processor hypercube machine indicate that regular communication schemes with a small size should be implemented by the first method, other schemes by randomization.

4. Examples

The first example is downloading an abstract hypercube machine from a single source. The problem is to transfer M blocks from the source to all other nodes.
If we use a doubling method, downloading takes time

(2) $T_H = \log_2 N \cdot M \cdot T$

where T is the time for one communication and N is the number of nodes.
If we use the ring abstract machine we get

(3) $T_R = (N + M - 2) \cdot T$

The ring abstract machine can be mapped easily onto the hypercube so that the size is one.
We have $T_H > T_R$, if

(4) $M > \dfrac{N - 2}{\log_2 N - 1}$

This proves the rather astonishing result, that using the most primitive abstract machine gives a more efficient implementation if the problem size (M) is large enough.
The importance of the ring abstract machine has been dicovered recently by different authors (see [KRU84]). This should not really be surprising, since the ring machine is nothing else but a pipeline, a well proven paradigm for achieving high throughput.

The next example is the matrix multiplication. The multiplication of two matrices A and B is given by

(5) $c_{ij} = \sum_k a_{ik}\, b_{kj}$

We use a 2-D lattice abstract machine to implement the multiplication. In order to do this we have to make the following decisions

- how to partition the problem domain
- how to distribute the partitions

A simple distribution binds a_{ij}, b_{ij} and c_{ij} statically to process p_{ij}. The case N = 9 is shown in the next figure

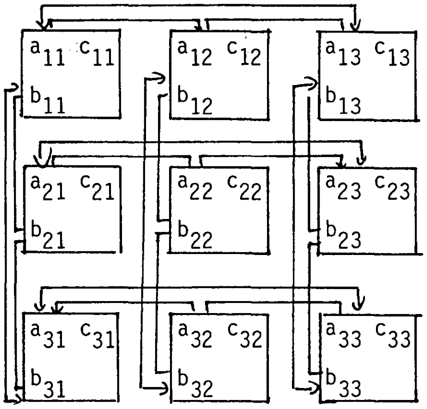


Figure 3: Matrix multiplication

This implementation gives a size of O $(\sqrt{N})$ and requires routing.
We will now try to find a communication scheme with a smaller size by applying transformations. We list only two important transformation rules

Rule 1: (How to start)
The data should be distributed in such a manner that all the processes can start with a computation.

Rule 2: (How to bind data to processes) If processes P_e and p_m request the same data d from process p_n and if the distance (p_e, p_n) = distance (p_e, p_m) + distance (p_m, p_n), we replace request (d, p_n) in p_e by a request (d, p_m) after p_m has got d from p_n.

The second rule describes a dynamic binding of the data d to the processes. These two transformation rules and a very complicated rule taking into account the data dependencies transform the first implementation to an implementation already found by Cannon.

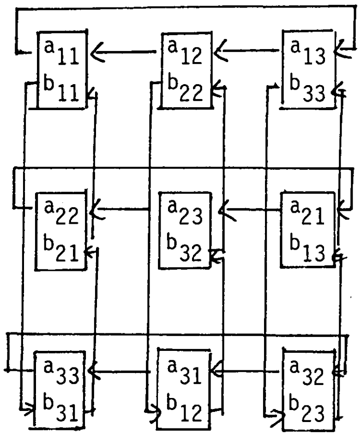


Figure 4: Matrix Multiplication by Cannon

The second implementation uses a modified 2-D lattice machine (torus), a dynamic binding of a and b to the processes and a distribution which obeys rule one. The implementation has a size of 1.

These examples demonstrate the usefulness of the proposed approach. But it should be remarked that the transformation rules are still rather informal We see the mathematics for communication schemes and their transformations slowly emerging. First steps are the metric pattern theory [GRE86] and the automatic synthesis of systolic algorithms [QUI84].

The MUPPET programming environment supports the approach outlined in chapter two and four. Generic abstract machines are provided by programm templates. The transformations will be implemented semi-automatically, this means the programmer directs MUPPET interactively.
We also want to note the similarity of this approach to the implementation assistant of the PIE programming environment [SERU85].

5. CONCURRENT MODULA-2

CONCURRENT MODULA-2 is based on the actor model of Hewitt [HEW77] and is an upward-compatible extension of Modula-2 [WIR83]. We chose Modula-2 as the base language, because it is well-structured, offers a module concept for modular programming and separate compilation of modules, and can be implemented efficiently even on micro-computers.

A CONCURRENT MODULA-2 program is based on LAM. It consists of a set of processes, communicating with acquainted processes by sending and receiving messages. In accordance with SCRIPT [FRA83] we have introduced an abstraction mechanism for communicating processes.

Several modern programming languages [COO80] [DAN81] [OCC83] have constructs to support inter-process communication; some of the constructs are slightly higher-level than others, but most can be considered low-level, because they handle some primitive communication between two partners at a time.

The following description presents the main features of CONCURRENT MODULA-2.
A parallel program in CONCURRENT MODULA-2 comprises of a set of communication patterns and process modules that 'import' some of these pattern to be used in communication steps.
A special module, marked as the initial process module is created as the first process of the parallel application and typically behaves as a supervisor of the computation.

The actions of a process in CONCURRENT MODULA-2 are specified in a program module, a syntactic entity of Modula-2, which is called process module in the following. A process is an instantiation of a process module. These process modules assure that processes operate only on local data objects, because none of the local definitions can be exported and used by other processes. Thereforeno shared data can be provided between different processes.

Processes in CONCURRENT MODULA-2 may be created dynamically during runtime with a create operation called NEWTASK. NEWTASK applied to a name of a process module creates and activates a new process and returns a reference to the active process, executing the corresponding process module.

References to processes are values of type TASK or of a special process type denoted by the name of the process module. In the same way variables can be declared of type TASK or typed with the name of a process module.

Within a process module coroutines of Modula-2 may be used to structure the internal behaviour of the process and to get smaller entities of parallelism.

Processes terminate by executing an explicit terminate statement. They cannot be killed by other processes, except by the initial supervisor process, controlling the computation.

Processes communicate by asynchronous message passing. As indicated above, a process can only communicate with acquainted processes, which means, that the process reference must be known. Each process has its own mailbox, in which the arriving messages are collected.

To participate in a communication, processes import communication patterns, which define the message types for the transaction.
Two processes can communicate successfully, if they know each other and use the same communication pattern. The use of patterns for the communication allows to check the message passing activity. Type checking can be done during compile and link time, provided typed processes are used.

A pattern definition for an ansynchronous communication consists of a pattern name and a list of types, indicating the types of the parameters of the message.

```
   PATTERN
      PatternName ( TypeList ):
```

The basic communication primitive in CONCURRENT MODULA-2 is the asynchronous send statement with no-wait semantic according to the classification of Liskov LIS79 . In a communication transaction, a sending process executes a send statement with an underlying pattern. The send statement specifies the pattern name, actual message parameters and the receiver process.

```
   SEND PatternName ( ActualMessageParms )
        TO Process:
```

A receiver process executes a receive statement, specifying the pattern and a variable list to store the incoming values of the expected message. Optionally the sending process may be denoted. During the execution of a receive statement, the mailbox of the process is checked for a matching message, i.e. a message with the respective pattern name. If no matching message is found, the process will be blocked. Otherwise the actual parameters of a message are bound to the variables of the VariableList. If a DO-END block is specified, the corresponding statements are executed with the variable bindings of the message received.
The variables in the VariableList may either be globally defined or local   to the DO-END block of the receive statement.

```
   RECEIVE PatternName ( VariableList )
       FROM Process    DO StatementSequence END
```

Higher level communication is supported by the following additional features:
A select statement allows a process to wait for the first of several acceptable messages. Every select alternative can be attached with a boolean expression and is a set of CONCURRENT MODULA-2 statements, whereby the first statement is a receive statement.
The semantic of the select statement is wellknown from the ADA DOD80 select.

One of the most important features of CONCURRENT MODULA-2 is the following:
Processes can be combined into sets. Sets can be used in send/receive statements. Therefore messages can be sent to several processes simultaneously. The usual operation on sets like union, exclusion etc. support the creation and transformation of communication schemes.
If a process set is used in a receive statement, it is possible to receive a message from one of the processes without specifying any precedence.

We illustrate the flavour of CONCURRENT MODULA-2 by means of a simple example
a parallel program with a buffer manager, a producer and a consumer process.
The parallel module BoundedBuffer, given below, includes the process descriptions for buffer, producer and consumer processes.

The supervisor module is assumed to be separately compiled and represents the actions of the process created at the beginning of the computation. The supervisor's task is to create buffer, producer and consumer processes and to install the necessary communication scheme, i.e. to send the process reference of the buffer process to both producer and consumer processes using the pattern InitProdCon.

For communication transactions with a buffer process, the pattern put is imported into the producer's module and get is imported into the consumer's module.
InitProdCon is the pattern for the necessary initializations done by the supervisor. The code segments in the producer/consumer modules are trivial: messages of the corresponding patterns are sent/received. Communication is done with the buffer process, referenced by the variable BufferProcess, declared as a Buffer-typed process. To use the name of a process module as type of a process variable the modules name must be declared in a USE-list.

A buffer process with local declarations for the control of the buffer array repeatedly waits for either a put or a get message. Put messages contain a character to be storedin the buffer array: get messages don't contain a message value.
A get transaction returns a character value to the sending consumer process, put transactions return no value.

The select statement within the buffer module has two alternatives. The first alternative handles a put message, which can be accepted whenever the buffer array is not yet filled. The second alternative accepts get messages, when data is available i.e. the buffer array is not empty.
Note that a producer process is not blocked after sending the message even if the buffer array is full. A consumer process executes a send-wait and is blocked until the reply message arrives.

```
PARALLEL MODULE BoundedBuffer;
    USE supervisor;
    PATTERN
       put( CHAR );
       get(): ( CHAR );
       InitProdCon(Buffer);

    MODULE Buffer;
    IMPORT get, put;
    CONST MaxSize=1000;
    VAR in, out, n : CARDINAL;
        buf: ARRAY [0..MaxSize-1] OF CHAR;
    BEGIN
       n:=0; in:=0; out:=0;
       LOOP
          SELECT
             WHEN n < MaxSize :
                RECEIVE put(buf[in]);
                in:=(in+1) MOD MaxSize;
                n:=n+1; |
             WHEN n > 0 :
                RECEIVE get REPLY (buf[out]);
                out:=(out+1) MOD MaxSize;
                n:=n-1;
          END (* SELECT *)
       END (* LOOP *)
    END Buffer.

    MODULE Producer;
    USE Buffer;
    IMPORT InitProdCon, put;
    VAR BufferProcess: Buffer; data: CHAR;
    BEGIN
       data:="any char";
       RECEIVE InitProdCon(BufferProcess);
       LOOP
          SEND put(data) TO BufferProcess;
       END (* LOOP *)
    END Producer.

    MODULE Consumer;
    USE Buffer;
    IMPORT InitProdCon, get;
    VAR BufferProcess: Buffer;
        data: CHAR;
    BEGIN
       RECEIVE InitProdCon(BufferProcess);
       LOOP
          SEND get TO BufferProcess WAIT (data);
       END (* LOOP *)
    END Consumer.

    INITIAL supervisor;

END BoundedBuffer.
```

Figure 5: Bounded buffer in CONCURRENT-MODULA-2

6. Tools of the MUPPET programming environment

MUPPET tries to help the programmer as much as possible before the program is run on the target machine. MUPPET relies on the synergy effect between scientific workstations and supercomputers. Some of the tools are shown in the next figure.

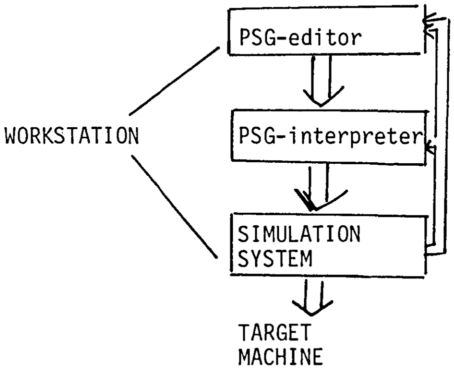


Figure 6: The MUPPET programming scenario

We will describe the simulation system in more detail. The goals of the simulation system are as follows

- the simulator should run concurrent programs unmodified
- the simulator should support different hardware topologies
- the simulator should allow interactive use

The simulator is written in MODULA-2. It is based on a discrete time, event driven and objected oriented simulation model. Some of the problems of simulators are broadly discussed in [FUJ85]. Very important is the method to determine the time which would elapse on the target machine while the program executes.
The MUPPET simulator currently supports CONCURRENT MODULA-2 and MIMD FORTRAN.

For a specific language only a preprocessor and a run-time system has to be provided. Necessary operating system services are implemented in the kernel of the simulator.
The simulator supports the MUPPET programming paradigm, Therefore it can handle different abstract machines.

The hardware machine is specified by a simple hardware description language. Predefined hardware objects like processors and buses are instantiated at runtime and connected according to the specification. In a later version the hardware specification will be assisted graphically. Each hardware object acts as a coroutine in MODULA-2. The coding of the hardware objects is hidden from the user. The programmer may also use predefined hardware machines like the 2-D lattice machine, a hypercube machine and the SUPRENUM machine. The user can specify and evaluate any process to processor mapping strategy. The runtime interface of the simulator supports interactive use. The user is able to intervene in a running experiment at any time e.g. to display some informations or to start debug actions. For that purpose a special interface language is provided.

Special emphasis has been given to the debugging features. We devide debugging into two parts

- debugging of the communication part
- debugging of sequential process

The first part will supervise and influence the flow of messages between interacting processes. The debugging features will be similar to the ones discussed in [SMI85].
Experiences with former simulation projects ([MUH81], [BMS83]) greatly influenced the design of the simulation system.

7. Experiences and Conclusions

A number of studies have been made so far, mostly with numerical applications [HOMU86]. We have found that parallel algorithms for local memory multiprocessors can be formulated either data-structure or macro data-flow oriented.

The data structure oriented approach emphasizes the global aspects of the algorithm, whereas the macro data-flow solution concentrates upon the dynamic interaction of parallel processes by the flow of data. The LAM machine can be thought of as a macro data flow machine if the processes do not have information about internals of other processes.

SIMD machines support the data structure approach. It is obvious, that a SIMD machine can easily be implemented by LAM while the reverse is not true.
The macro data flow approach leads to very simple programs for the numerical parts, but new problems arise with the control part-initialization and termination of processes and their mapping to processors.

Every "real life" algorithm seems to need at least two different communication schemes: one scheme for the numerical algorithm and one for the control part. The best way seems to implement the two communication schemes by two different process nets, one process net supervising the other net. This separation of concerns makes it possible to implement different control strategies without changing the numerical process net. We have shown that distributed process initialization and termination can be implemented efficiently with a quadtree abstract machine [HOMU86].
More research has to be done to test the reliability of the macro data flow aaproach. Of particular importance is the definition of well-structured communication schemes for a broad class of applications, the integration of these schemes into the programming languages and into the programming environment.
Another important observation is that local memory multiprocessor systems have most ot he problems of distributed systems. Fundamental problems are distributed termination, distributed snap shots and the management of time. Without solving these problems by using newest results of computer science research, the above systems have only limited applicability.

References

BMS83 C.Beilken, F.Mattern, M.Spenke: Entwurf und Implementierung von CSSA - Teil C: CSSA - Systembenutzung, Interner Bericht 67/83, Universität Kaiserslautern, Feb. 1983

BSN85 R.Bahlke, G.Snelting: The PSG - Programming System Generator, Proc. ACM SIGPLAN 85 Symp. on Language Issues in Programming Environments, Seattle, Washington, 25-28. Juni 1985

BRO86 J.C.Browne: Framework for formulation and analysis of parallel computation structures, Parallel Computing 3 (1986)

COO80 R.P.Cook:*MOD - A Language for Distributed Programming, IEEE-Transaction on SE, Vol.6 No.6, pp 563-371, 1980

DAN81 R.B. Dannenberg: AMPL - Design, Implementation and Evaluation of a Multiprocessing Language, Tech. Report, Department of Computer Science, Carnegie Mellon University, 1981

DOD80 Reference Manual for the ADA Programming Language US DOD-Report, July 1980

FIL84 R.E.Filman, D.P.Friedmann: Coordinated Computing, McGraw-Hill, New York 1984

FOR86 B.Ford: The What and Why of Problem Solving Environments in: Problem Solving Environments for Scientific Computing; B.Ford ed. to be published 1986

FRA83 N.Francez, B. Hailpern: A Communication Abstraction Mechanism. ACM Operating Systems Review, Vol.19, No.2, 1985

FUJ85 R.M.Fuijmoto: The Simon Simulation and Development System, Proc. Summer Computer Simulation Conference, July 1985

GIMU86 W.Giloi, H.Muehlenbein: Design and Rationale for the SUPRENUM architecture, Proc. 1986 Int. Conf. on Parallel Processing, 1986

GRE85 U.Grenander: Pictures as complex systems in: Complexity, Language and Life: Mathematical approches; J.L. Casti, A. Karlquist eds. Springer Berlin 1985

HEW77 C.Hewitt: Viewing Control Structures as Patterns of Passing Messages, Artifical Intelligence, Vol.8, pp 323-364, 1977

HOMU86 H.C.Hoppe, H.Muehlenbein: Parallel adaptive full-multigrid methods on message-based multiprocessors; to be published in Parallel Computing, 1986

KRU85 C.P.Kruskal: Performance Bounds on Parallel Processors: An Optimistic View; in: Control Flow and Data Flow: Concepts of Distributed Programming, M. Broy ed. Springer, Berlin 1985

LIS79 B.Lishkov: Primitives for Distributed Computing. Proc. of the 7th Symp. on Operating System Principies, pp 33-42, 1979

MUH81 H.Muehlenbein: "TOCS - A SIMULA-based Simulator for the Analysis of Mainframe-oriented Distributed Systems", LNCS 123, Springer-Verlag, Berlin 1981

OCC83 INMOS, Occam Programming Manual 1983

QUI84 P.Quinton: Automatic Sythesis of Systolic Arrays form Uniform Recurrent Equations, Proc. of the 1984 Int. Conf. on Parallel Processing 1984 pp 208-214

SERU85 Z.Segall, L.Rudolph: PIE: A Programming and Instrumentation Environment for Parallel Processing, IEEE Software, Nov. 1985 pp 22-37

SMI85 E.T.Smith: A Debugger for Message-based Processes, Software - Practice and Experience, Vol.15 (11), Nov. 1985

SNY84 L.Synder: Parallel Programming and the Poker Programming Environment, IEEE Computer July 1984 pp 27-55

ULL84 J.D.Ullmann: Computational Aspects of VLSI, Computer Science Press 1984

WIR83 N.Wirth: Programming in Modula-2 Springer-Verlag, Berlin, 1983

DISTRIBUTED FUNCTIONS ALLOCATION FOR RELIABILITY AND DELAY OPTIMIZATION

S. Hariri*, C. S. Raghavendra**

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089.

ABSTRACT

In this paper, we present two optimization algorithms for allocating the functions of a given distributed task so that the reliability is maximized and the communication delay is minimized. A distributed task reliability measures the probability of executing successfully a task that is composed of a set of functions running on remote processing elements, while task delay describes the average delay incurred during the processing of a task. The allocation of resources, such as computing functions and files influences the reliability and the delay. Two different approaches are developed to allocate the resources of a given task. In the first method, the problem is reduced to a 0-1 integer linear programming problem and can be solved as two sub-problems. First, the set of allocations that maximize reliability is found using an approach based on a *branch-and bound* technique. Then, standard delay analysis is adopted to assess the average packet delay associated with each of these allocations to choose the optimal solution. In the second approach, a compound objective function is constructed to measure both reliability and delay. Then, it is used in an optimization algorithm to obtain the optimal distribution of a task's functions. A detailed example is presented to illustrate the optimization methodology.

1. INTRODUCTION

A Distributed System (*DS*) can be viewed as a collection of Processing Elements (*PE*'s) that are connected via a communications network and are controlled by a distributed operating system. A distributed task (*T*) is defined as a set of functions such as, updating a database, requesting a system service routine, and calling a remote procedure. The processing elements cooperatively execute tasks for improved performance. The number of *PE*'s involved and the degree of cooperation among them may vary considerably with the tasks. The static distribution of the resources (functions) associated with tasks influences their reliabilities, the recovery from failures, and the communication delay; the delay becomes significantly long when many intermediate computers must be visited before required resources can be accessed.

Several optimization problems have been addressed in the context of computer communication networks, for example, the capacity assignment problem, and the topology optimization problem [FRAN 72, KLEI 76, GERL 77, KLEI 80]. These problems minimize the total cost of the system to satisfy certain performance constraints such as average delay. Topology optimization with reliability constraint is a hard problem, and a heuristic solution is usually used which is to provide two or more node-disjoint paths between every pair of *PE*'s. The optimal allocations of the resources of a distributed database for minimizing storage cost and response times have been widely investigated [CHU 69, MARI 79, LANI 83, RAMA 83]. Researchers have also investigated the problem of combined allocation and network design to minimize storage and communication costs [MAHM 76, MARI 79, CHEN 80, IGNI 82, IRAN 82].

Algorithms that assist designers in determining the optimal allocations of the resources are important especially in the early stages of designing reliable distributed systems; they provide the initial estimation of the best preformance measures that can be obtained with a given set of resources. If they do not satisfy the system's requirements, alternative designs or more resources can be added. In this paper, the topology of a distributed system is assumed to be fixed, and it is required to distribute the resources of a task so that reliability is maximized and delay is minimized. In distributed systems, we are more interested in the allocation of multiple tasks than that of one task. However, our approach is still applicable to this case because these tasks can be combined together to form one task whose functions are the union of those associated with the given set of tasks. The optimization problem to be solved here is formulated as:

GIVEN	: Fixed topology. n processing elements. a task T of n_f functions.
MAXIMIZE	: Distributed task reliability (R).
MINIMIZE	: Task Packet Delay (TPD).
OVER DESIGN VARIABLES	: Allocation variables.
SUBJECT TO	: Redundancy used is less than an upper bound.

* Supported by Agency for International Development (AID) Fellowship.

** Supported by NSF Presidential Young Investigator Award No. ECS-8452003 and a Grant from AT&T Information Systems.

CH2345-7/86/0000/0344$01.00 © 1986 IEEE

Two different approaches are used to solve this complex optimization problem. In the first method, it is decomposed into two sub-problems: 1) find the set of allocations that maximize the reliability, and 2) find the task packet delay associated with each of these allocations and choose the solution that minimizes the delay. In the second approach, a compound objective function is constructed to measure the effects of resource allocations on both reliability and delay. Then, this objective function is maximized using an algorithm similar to the one developed in the first approach. Such an optimization procedure can be used during the design phase and when there are changes in the distributed system. The optimization can be performed periodically to account for the time varying reliabilities.

The organization of this paper is as follows: in the next section, reliability analysis of distributed systems is presented. A method for estimating the average task packet delay is discussed in Section 3. Optimization algorithms, which are based on the branch-and-bound technique, to solve different variations of the problem formulated previously are developed in Sections 4 and 5. An illustrative example is presented in Section 6. Concluding remarks and summary of the paper are discussed in the final section.

2. RELIABILITY ANALYSIS

In our analysis, a distributed system is modeled as an undirected graph in which the nodes and the edges represent the processing elements and the communication links, respectively. The conventional reliability measures used in communications networks such as network connectivity and terminal reliability [WILK 72, GRNA 80, HARI 86a] are not sufficient to describe accurately the reliability in distributed systems. This has led researchers to develop new reliability measures, such as source-to-multiple-terminal reliability [SATY 81], survivability index [MERW 80], and distributed program and system reliability measures [PRAS 86, HARI 86b].

A distributed task (T) which consists of n_f functions can run successfully when all these functions are processed correctly and are accessible for information exchange. The reliability of a task can therefore be measured by the probability of processing all its functions successfully. This depends on the reliability of the processing elements and the communication channels, and the allocation of the interacting functions. The reliability p_i of a component x_i (it can be either a computer or a communication channel) represents the probability of being operating.

Redundancy in the functions (f_i's) of a distributed task is introduced to increase reliability and fault tolerance. The level of redundancy (number of redundant copies of functions or files) can be increased up to the total number of processing elements in the system. However, a large number of redundant copies of resources (such as replicated files) introduces additional traffic to maintain consistency. Typically, the number of redundant copies will be small (2 or 3 copies).

In what follows, we briefly discuss an approach for evaluating the task reliability associated with a given distribution of its functions. This is done by first determining the set of all trees that provide the required accessibility to the cooperating functions. These trees are referred to as Minimal Task Spanning Trees (*MTST*'s). An *MTST* is minimal when there is no other *MTST* of smaller size (smaller number of nodes). Then, the probability of executing a given distributed task successfully is evaluated as the probability that there exists at least one *MTST* in the operational state (all its nodes and links are fault-free) in spite of channel and node failures, i.e.,

$$R = P_r(\text{ at least one } MTST \text{ is operational}) \qquad (1)$$

Let N_t denote the total number of *MTST*'s for a given allocation of the functions. The distributed task reliability can be written as:

$$R = P_r(\bigcup_{i=1}^{N_t} MTST_i) \qquad (2)$$

The evaluation of Equation 2 can be explained through an example. Figure 1 shows an allocation of the duplicated functions (f_1, f_2) of a distributed task T.

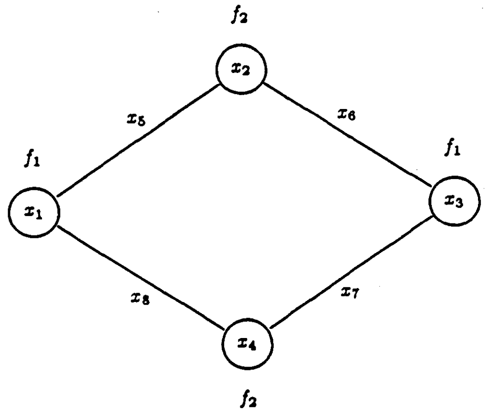


Figure 1: A maximal reliability allocation of 4 functions.

For example, in $MTST_1 = x_1x_2x_5$ of Figure 1, the functions (f_1, f_2) are processed on nodes x_1 and x_2, respectively, while channel x_5 is used for data exchange. The other trees that run the task under consideration are:

$$MTST_2 = x_1x_4x_8$$
$$MTST_3 = x_2x_3x_6$$
$$MTST_4 = x_3x_4x_7$$

Substituting these trees in Equation 2, yields the following:

$$R = P_r(\bigcup_{i=1}^{4} MTST_i)$$

The reliability expression of Equation 2 can be obtained using any terminal reliability algorithm based on path enumeration when the set of *MTST*'s are used instead of the set of the simple paths. A detailed description of algorithms to enumerate efficiently these trees and then to determine the corresponding reliability expression can be found in [PRAS 86, HARI 86b]. If we apply the SYREL algorithm in [HARI 86a] to the above example, the following reliability expression is obtained:

$$\begin{aligned} R = {} & p_1p_2p_5 + p_1p_4p_8(1-p_2p_5) \\ & + p_2p_3p_6[q_1+p_1q_5(1-p_4p_8)] \\ & +p_3p_4p_7[q_1(1-p_2p_6)+p_1q_8(q_2+p_2q_5q_6)] \end{aligned}$$

where p_i and q_i denote the reliability and the unreliability ($1-p_i$) of an element x_i, respectively.

The reliability expression can be modified by replacing the reliability of the components (p_i's) with the corresponding time-related functions to derive other reliability measures, such as availability, mean time to first failure, and mean time between failures [RAGH 83]. To measure the amount of reliability improvement achieved from adding redundant copies of the functions, we introduce a measure that relates the increase in reliability resulted from redundancy level r to the maximum obtainable improvement. This maximal reliability increase occurs when the non-redundant reliability $R(0)$ is improved up to 1 (the maximal reliability). The Reliability Improvement Factor (*RIF*) is used to measure this relative improvement, and is defined as:

$$RIF = \frac{R(r) - R(0)}{1 - R(0)} \qquad (3)$$

where $R(r)$ and $R(0)$ denote the reliability with r redundancy and with no redundancy, respectively. If we assume all the processing elements and the channels have a reliability of 0.9, the *RIF* of the functions allocation shown in Figure 1 equals 90.37%.

The evaluation of task's reliability can not be achieved in polynomial execution time [BALL 80]; therefore, it is not possible to evaluate it at each iteration of an optimization algorithm that deals with large distributed systems. Furthermore, the purpose of reliability evaluation is to direct the search of an optimization algorithm toward the allocations that maximize the reliability. Hence, the approximation of task reliability can be used effectively in this type of application. Most of the components' reliabilities are generally within a close range. Therefore, with a good approximation, we can assume that all the computers and communication links have the same reliabilities and are equal to p_c and p_l, respectively. Let n_t denote the number of nodes in an *MTST* of a task T. The reliability and unreliability of that tree are:

$$r_t = p_c^{n_t} \cdot p_l^{n_t - 1}$$
$$q_t = 1 - r_t$$

If we assume that there are only two disjoint trees (trees with no common nodes or links) for a given distribution of functions, then the reliability of executing that task is:

$$R = 2\, r_t\, q_t + r_t^2$$

If we assume that all the N_t trees of Equation 2 are all disjoint, then its evaluation can be generalized as shown in Equation 4.

$$R = \sum_{k=1}^{N_t} \binom{N_t}{k} r_t^k \cdot q_t^{N_t - k} \qquad (4)$$

However, the sets of trees are not usually disjoint and therefore there will be some common components among them. The failure of any one of them will contribute to the failure of more than one tree. As a result, the reliability given in Equation 4 can be viewed as an upper bound and can be used to approximate the reliability in the optimization algorithm presented in Section 5.

3. DELAY ANALYSIS

In addition to studying the task reliability associated with an allocation of its functions, one can also analyze the corresponding packet delay, throughput, or any other useful performance measure. In this section, we present an approach for estimating a packet delay incurred during its movement in a store-forward message switching network. It is based on queueing network models. Queues on a single communication link are characterized by the arrival rate of packets flowing through it and the distribution of transmission times. The major difficulty of the queueing network model is that the flow of traffic throughout the network is not independent. To simplify the analysis of these queueing models, the independence assumption is introduced in which the length of a message (packet) is considered as an independent random variable as it moves from one node to another. If we also assume Poisson processes for all packet arrivals, exponential distribution of packet length with $\frac{1}{\mu}$ bits per packet, fixed routing, and no nodal delay, then the average delay T_k on channel k is given by [KLEI 76] as:

$$T_k = \frac{1}{\mu C_k - \lambda_k} \qquad (5)$$

where C_k *and* λ_k denote the capacity of channel x_k and the traffic flow on it (both in bits per second), respectively, and $\frac{1}{\mu}$ represents the average packet length.

Equation 5 is an approximation of the delay associated with a channel. The consequent analysis is not restricted to that formula or to the assumptions used in its derivation. However, any other equation derived based on more realistic assumptions can be used to replace Equation 5. A Task Packet Delay (*TPD*) is defined as the average packet delay incurred during the information exchange between the cooperating computers. The links of an *MTST* can be used to dictate the routes of information exchange among the cooperating *PE*'s. One criterion for choosing an appropriate tree is identifying the one that leads to a better load balancing of the distributed system.

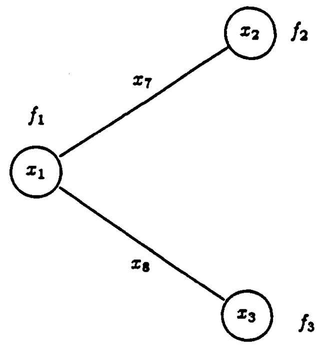


Figure 2: An *MTST* with the allocation of a T's functions.

For example, Figure 2 shows an *MTST* and the allocations of three functions (f_1, f_2, f_3) that compose a distributed task. In that tree, the following routes are used for information exchange among the functions: 1) x_7 for transferring packets between f_1 and f_2, 2) x_8 for transferring information between f_1 and f_3, and 3) $x_7 x_8$ for transferring packets between f_2 and f_3. Since there are many trees for each allocation of the functions, the TPD_p of an Allocation a_p can be computed in term of the Packet Delay (PD_i) associated with each $MTST_i$. This can be expressed as follows:

$$TPD_p = \frac{1}{N_t} \sum_{i=1}^{N_t} PD_i \tag{6}$$

where N_t denote the number of trees that can run a given task.

Let $tr(i,j)$ denote the number of packets transferred between functions (f_i, f_j) and $e_l(i,j)$ be a binary variable that takes on value 1 when x_l of an $MTST_i$ is included in the route between f_i and f_j. The total amount of traffic (γ) flowing on an $MTST_i$ is the summation of all the traffic between all the pairs of the functions, i.e.,

$$\gamma = \sum_{i=1}^{n_f} \sum_{j=1}^{n_f} tr(i,j) \tag{7}$$

The packet delay PD_i of an $MTST_i$ can be found by applying the following five steps [TANE 81]:

1. Find the traffic flow on each channel of $MTST_i$.

$$\lambda_l = \sum_{i=1}^{n_f} \sum_{j=1}^{n_f} e_l(i,j)\, tr(i,j) \quad , \forall\, x_l \in MTST_i \tag{8}$$

2. Find the total traffic on all channels.

$$\lambda = \sum_{\forall x_l \in MTST_i} \lambda_l \tag{9}$$

3. Find the mean number of hops per packet.

$$\bar{n} = \frac{\lambda}{\gamma} \tag{10}$$

4. Find the Mean Channel Delay (MCD_i).

$$MCD_i = \sum_{\forall x_l \in MTST_i} \frac{\lambda_l T_l}{\lambda} \tag{11}$$

5. Find the Packet Delay of $MTST_i$ (PD_i).

$$PD_i = \bar{n}\ MCD_i \tag{12}$$

The objective of the optimization algorithms presented here is to find the set of allocations that optimize both reliability and delay. The comparison among allocations is not straightforward because of their different effects on reliability and delay. Hence, in order to simplify the comparison, the TPD is normalized with respect to the Worst Task Delay (WTD). This delay is computed in similar steps to the one used in computing the TPD after considering the following changes:

1. The mean channel delay (MCD_i) associated with an $MTST_i$ is considered the worst possible delay. This occurs when the maximal traffic flows through the minimal capacity channel, i.e.,

$$\frac{1}{\mu C_{\min} - \gamma} \tag{13}$$

where $C_{\min}$ denotes the minimal capacity channel of $MTST_i$.

2. The mean number of hops per packet is equal to the number of links in $MTST_i$, that is:

$$\bar{n} = n_t - 1$$

where n_t denotes the number of nodes in $MTST_i$.

The ratio ($\frac{TPD}{WTD}$) is unitless and therefore can be used with the reliability function to form a compound objective function that measures both reliability and delay.

4. DECOMPOSING THE OPTIMIZATION PROBLEM

This approach is suitable for large distributed systems because it does not require the evaluation of a reliability expression to determine the allocations that maximize it. It has been shown in the literature that the time complexity of reliability algorithms is not polynomial [BALL 80]. A property of the allocation that maximizes the reliability is identified to guide the search for this type of allocation.

Let T be a distributed task of n_f functions and also require the cooperation of n_f computers. The function f_i that is assigned to run on a computer does not necessarily need to be only one function; it could be a set of small functions grouped together to form one entity in order to satisfy some other performance constraints. For a given distributed system, let $n_{\max}$ be the maximum number of trees of n_f PE's that run T and n be the total number of PE's.

Definition 1: A tree of n_f PE's is said to **cover** a distributed task T if its nodes can run all the functions such that each one runs on a distinct PE.

Theorem 1: In a distributed system with only one function allocated to each node, the reliability of a distributed task T requiring n_f functions is maximized if each of the $n_{\max}$ trees covers T.

Proof: Let us assume that there exists an allocation of the functions ($a_{\max}$) such that each tree of n_f nodes covers T. The number of trees covering T in any other allocation is either less than or equal to $n_{\max}$ because any allocation that has number of trees larger than $n_{\max}$ will have some trees with larger than n_f nodes. These trees do not contribute to the reliability of a task since they are supersets of smaller size trees of n_f nodes. If it is equal to $n_{\max}$, the reliability is the same because they both have the same set of trees that cover T. Let a_j be an allocation in which the number of trees covering T ($n_c(a_j)$) is less than that of $a_{\max}$. This set of trees is a subset of that corresponding to $a_{\max}$. Hence, the task reliability corresponding to a_j, which is the probability that there exists at least one tree in the operational state, is less than that associated with $a_{\max}$.

□

Most of the optimization algorithms of computer networks and distributed systems that address reliability as a performance requirement to be met do not use the reliability expression in their formulations. Instead, they use a simpler criterion such as the connectivity of the network or maintaining a lower bound on the number of node-disjoint paths between node-pairs. We also do not use the reliability expression to guide the search toward maximal reliability allocations, but the result of Theorem 1 is used instead to identify those solutions. That is, the number of trees executing the given task is maximized. This allows us to formulate the problem as a 0-1 integer linear programming problem.

Let $x_{i,j}$ denote a binary variable that takes on value 1 when function f_i runs on node x_j and 0 otherwise. Also, let $n_c(a_p)$ represent the number of trees of n_f *PE*'s covering a distributed task T for an allocation a_p. The allocation procedure of a task's functions can be formulated as follows:

Formulation 1

GIVEN : Fixed topology.
n processing elements.

MINIMIZE : $\sum_{i=1}^{n_f} \sum_{j=1}^{n} x_{i,j}$
; minimize the total number of allocated functions.

SUBJECT TO : $n_c(a_p) = n_{\max}$
; all trees of n_f nodes cover T.
and $\sum_{j=1}^{n} x_{i,j} \leq r_i \quad i=1, \ldots, n_f$
; r_i is an upper bound of redundancy in f_i.

In large distributed systems where the number of *PE*'s is much larger than the number of functions to be distributed, it is not practical to allocate them so that all the trees of n_f *PE*'s cover the task under consideration (T); to cover all trees, the upper bound on redundancy could be increased up to $\binom{n}{n_f}$ which is not feasible because of the difficulty in maintaining consistency and the overhead traffic. As a result, the previous formulation is modified so that the number of trees that cover a given task is maximized. We introduce a control variable c_k to refer whether or not tree t_k covers T. It is a binary variable which takes on a value of 1 only when each *PE* in that tree processes a distinct function of T and 0 otherwise.

The formulation of the problem becomes as follows:

Formulation 2

GIVEN : Fixed topology.
n processing elements.

MAXIMIZE : $n_c(a_p) = \sum_{\forall t_k} c_k$.
; where $n_c(a_p)$ denotes the number of trees covering T for allocation a_p.

SUBJECT TO : $\sum_{j=1}^{n} x_{i,j} \leq r_i \quad , i=1, \ldots, n_f$
; r_i is an upper bound of redundancy in f_i.

Formulation 2 is a standard 0-1 integer linear programming problem and therefore any efficient algorithm can be used to solve this allocation problem. An algorithm based on a branch-and-bound technique is used to solve Formulation 2 [HILL 80]. The algorithm consists of four steps: branch, bound, fathoming, and stopping. In the branch step, one set is selected according to a criterion which could be the one that gives the better bound or the most recently created set. In the bound step, the number of trees covering T $(n_c(a_p))$ is determined for a_p. In the fathoming step, the selected set is checked and then dropped if it violates the set of constraints. Once it is removed, the algorithm proceeds from the branch step. Otherwise, the following steps are performed before it proceeds to the stopping step: 1) the current lower bound (Z_l) is reset to $n_c(a_p)$ whenever it is larger than Z_l; 2) the selected allocation a_p is partitioned into two subsets about an allocation variable $x_{i,j}$. In the first subset, $x_{i,j}$ is set to 1 while in the second subset, it is set to 0. In the stopping step, the algorithm is stopped when all the allocation variables of the remaining subsets have been considered; otherwise, it proceeds from the branch step and so on. The optimal allocations are those whose $n_c(a_p)$'s are equal to Z_l.

If all the computers and the communication links have the same reliability, then any allocation a that maximizes the number of trees covering a task T will result in a maximal reliability. This is true because in this case all the trees will have the same reliability and therefore the highest value is achieved when the number of trees is maximized. However, if the elements have different reliability values, it is possible for an allocation a' to exist, for which $n_c(a') < n_c(a)$, and its reliability is larger than that of a. In general, the reliability values of computers as well as communication channels are in a close range, and therefore algorithms for solving Formulation 2 will identify optimal or near optimal solutions. Now we present the optimal file allocation procedure algorithm.

Function Allocation Procedure (FAP)

1. Initialization.
   1.1 $Z_l = 0$; the current lower bound is zero.
   1.2 $A = a_0$; list A has allocation a_0 in which no function has been assigned to any node.

2. Branch step.
   2.1 select the newest partitioned subset, say a_j,

3. Bound step.
   3.1 find $n_c(a_j)$; the number of trees covering T.
   3.2 the number of trees that can not cover T $(n_c(a_j)')$.

4. Fathoming step.
   4.1 a_j is fathomed, and is therefore removed from list A, when one or more of the following constraints are violated:
   $\sum_{j=1}^{n} x_{i,j} \leq r_i \quad , i=1 .,.,., n_f$
   $n_{\max} - n_c(a_j)' > Z_l$; the number of trees that could cover T is greater than Z_l.
   4.2 if a_j violates the constraints, go to branch step; otherwise, do

```
4.2.1 if $n_c(a_j) > Z_l$, then $Z_l = n_c(a_j)$
4.2.2 partition $a_j$ into two subsets $(a_{j_1}, a_{j_2})$ about the unconsidered variable $x_{p,k}$ such that $x_{p,k} = 1$ in $a_{j_1}$ and $x_{p,k} = 0$ in $a_{j_2}$.
5. Stopping step.
If all the allocation variables $x_{i,j}$ have been considered, the procedure stops; otherwise, go to the branch step.
```

Algorithm 1

```
1. Enumerate all the trees of $n_f$ nodes.
2. Apply FAP to obtain the set of allocations that maximize the number of trees covering $T$.
3. For each allocation obtained in 2, evaluate the task reliability using the approach discussed in Section 2. Then, discard from the set of allocations the ones that have lesser reliability than other allocations with maximal reliability.
4. Find the $TPD$ associated with each allocation obtained in 3.
5. Choose the optimal allocation such that $TPD_p$ is minimized, i.e.,
   $TPD_{opt} = \underset{\forall p}{MIN}(TPD_p)$
```

5. CONSTRUCTING A COMPOUND OBJECTIVE FUNCTION

Instead of solving the optimization problem sequentially as done in Algorithm 1, one could form a function that measures both reliability and delay and then find the allocation that maximizes that function.

$$F = \alpha R - \beta \frac{TPD}{WTD} \tag{14}$$

where α and β are weight factors assigned to reliability and delay functions, respectively.

This function (Equation 14) can further be simplified if we normalize their summation, i.e., $\alpha + \beta = 1$

Substituting β with $1 - \alpha$ in Equation 14 yields to:

$$F = \alpha\left(R + \frac{TPD}{WTD}\right) - \frac{TPD}{WTD} \tag{15}$$

The value of the parameter α can be used to reflect the relative importance of the reliability with respect to delay. For example, maximizing the function F can either lead to optimizing separately the reliability or delay or both combined. If α=0, this results in minimizing the delay function while setting α=1 leads to maximizing the reliability. However, if α=0.5, both functions will receive equal weight. The optimization problem can be solved as shown in Formulation 3.

Formulation 3

GIVEN	: Fixed topology. n processing elements. parameter α.
MAXIMIZE	: $F = \alpha\left(R + \frac{TPD}{WTD}\right) - \frac{TPD}{WTD}$
SUBJECT TO	: $\sum_{j=1}^{n} x_{i,j} \le r_i \quad , i = 1, \ldots, n_f$; r_i is an upper bound of redundancy of f_i.

In some applications, it is required to reduce the cost of running the functions of a given task on the computers and also satisfy some reliability and delay constraints. Let us assume that $C_{i,j}$ denotes the cost of running function f_i on computer x_j. If the cost is a major objective in the functions allocation problem, one could solved it as shown in Formulation 4.

Formulation 4

GIVEN	: Fixed topology. n processing elements. parameter α.
MINIMIZE	: $COST = \sum_{i=1}^{n_f} \sum_{j=1}^{n} C_{i,j}\, x_{i,j}$
SUBJECT TO	: $\sum_{j=1}^{n} x_{i,j} \le r_i \quad , i = 1, \ldots, n_f$; r_i is an upper bound of redundancy of f_i. $R \ge R_{\min}$; reliability must be larger than a lower bound. $TPD \le TPD_{\max}$; delay must be smaller than an upper bound.

The Formulations 3 and 4 can be solved using a procedure similar to the one described in FAP. Algorithm 2 describes the steps of a procedure to solve Formulation 3, and also it can be used to solve Formulation 4 with a slight modification to the objective function and the set of constraints.

Algorithm 2

```
1. Initialization.
  1.1 $F_l = 0$ , $A = a_0$
2. Branch step.
  2.1 select the newest created allocation $a_j$.
3. Bound step.
  3.1 evaluate $n_c(a_j)$ and and $n_c(a_j)'$ (the number of trees covering $T$ and those do not cover it, respectively).
  3.2 evaluate $TPD$.
```

3.3 evaluate the task reliability using either the approach discussed in Equation 2 or 4.
3.4 evaluate the objective function F .

4. Fathoming step.
4.1 a_j is fathomed when one or more of the following constraints are violated:

$$\sum_{j=1}^{n} x_{i,j} \leq r_i \quad , i=1 .,.,., n_f$$

$$n_{\max} - n_c (a_j)' > Z_l$$

4.2 if the constraints are not violated then
4.2.1 if $F > F_l$ then $F_l = F$.
4.2.2 partition a_j into two allocations about $x_{p,k}$ such that
a) in the first allocation, node x_k has function f_p .
b) in the second allocation, node x_k does not have f_p .

5. Stopping step.
If all the allocation variables have been considered, the algorithm stops; otherwise, go to the branch step.

6. AN ILLUSTRATIVE EXAMPLE

Let us assume that we have a distributed task T consisting of three functions (f_1,f_2,f_3) which are to be distributed among the PE 's of the distributed system shown in Figure 3 so that the reliability is maximized and the delay is minimized. The capacity of the communication channels, the traffic rate between functions (f_1,f_2,f_3), and the redundancy level in this example are assumed as follows:

$C_7 = C_8 = C_9 = C_{14} = 20$ Kbits/sec.
$C_{10} = C_{11} = 30$ Kbits/sec , $\frac{1}{\mu} = 1000$ bits/packet
$C_{12} = C_{13} = C_{15} = 15$ Kbits/sec.
$tr\ (1,2) = 1$ packet/sec.
$tr\ (1,3) = 3$ packets/sec.
$tr\ (2,3) = 4$ packets/sec.
$r_1 = r_2 = r_2 = 2$

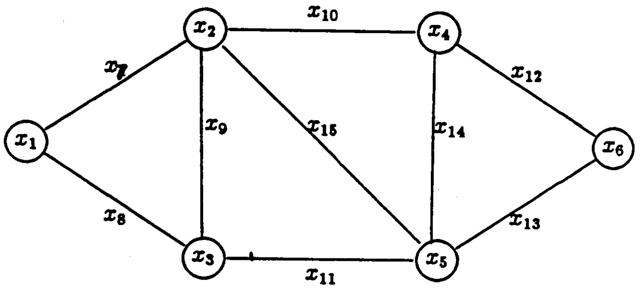


Figure 3. A six node distributed system.

In the following discussion, we use the two algorithms developed in the previous section to obtain the allocation that optimizes both reliability and delay. For simplicity, we assume that only one function could be allocated to a computer. The first step of Algorithm 1 is to find the number of trees of 3 computers that can be connected with trees of size 3. By doing so, the following 12 trees are obtained:

(x_1,x_2,x_3) , (x_1,x_2,x_4) , (x_1,x_2,x_5) , (x_1,x_3,x_5)
$(x_2,x_3,x_4,)$, (x_2,x_3,x_5) , (x_2,x_4,x_5) , (x_2,x_4,x_6)
(x_2,x_5,x_6) , (x_3,x_4,x_5) , (x_3,x_5,x_6) , (x_4,x_5,x_6)

Applying FAP leads to 18 allocations where each one of them covers only six trees of the 12 listed above. Although, these allocations cover the same number of trees, the corresponding reliability for each allocation is different because the number of $MTST$'s spanning the nodes of each tree depends on the degree of the nodes. For example, in Figure 3, there are three trees of size 3 that span nodes (x_1,x_2,x_3), while there is only one tree spanning nodes (x_1,x_2,x_4). Hence, the distributed task reliability (R) must be computed for each allocation to identify the one that maximizes it. By doing so, the number of allocations decreases to six. The distribution of the functions for each allocation of these six identified allocations is shown in Table 1.

Once the set of allocations is obtained, the next step is to use the equations in Section 3 to find the average packet delay associated with each allocation. The resulting delays and reliability of each allocation are also given in Table 1. The optimal allocation is a_2 since it has the minimal TPD_2 (107.6 msec), while allocation a_5 has the worst delay (154.6 msec) as shown in Figures 4 and 5, respectively. This can be explained as follows: in a_2, the functions that interact with each other heavily (f_2,f_3) are allocated such that they use highest capacity links (x_{10},x_{11}), while the opposite is done in a_5, in which light interacting functions (f_1,f_2) are assigned to use those links of high capacity.

If we assume all the nodes and links have the same reliability of p=0.9, the reliability of executing that task without any redundancy is

$$R\ (0) = p^5$$

and the reliability improvement factor for any one of the six allocations is

$$RIF = \frac{R\ (2) - R\ (0)}{1 - R\ (0)} = \frac{0.9597 - 0.5905}{1 - 0.5905} = 90.17\%$$

The reliability R(2) is obtained by applying the MFST algorithm given in [PRAS 86] and SYREL algorithm [HARI 86a].

Applying Algorithm 2 identifies the same optimal distribution of the functions obtained by allocation a_2.

Maximal Reliability Allocations

Maximal Task Reliability =0.9597

Allocation	x_1	x_2	x_3	x_4	x_5	x_6	TPD_i (in ms)
a_1	f_1	f_2	f_3	f_3	f_2	f_1	126.6
a_2	f_1	f_3	f_2	f_2	f_3	f_1	107.6
a_3	f_2	f_1	f_3	f_3	f_1	f_2	139.3
a_4	f_2	f_3	f_1	f_1	f_3	f_2	112.9
a_5	f_3	f_1	f_2	f_2	f_1	f_3	154.6
a_6	f_3	f_2	f_1	f_1	f_2	f_3	143.6

Table 1.
TPD 's of the allocations resulted from applying FAP.

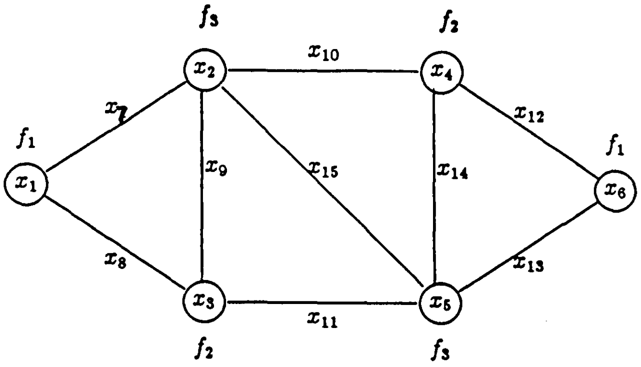


Figure 4. Distribution of functions according to a_2.

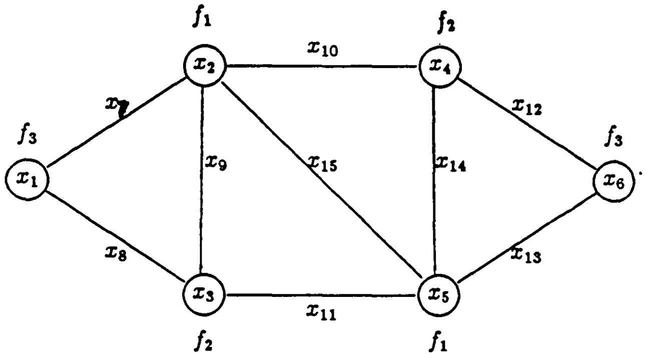


Figure 5. Distribution of functions according to a_5.

7. SUMMARY AND CONCLUSIONS

In this paper, we addressed the problem of resource allocation for combined reliability and delay optimization and present two algorithms. Algorithm 1 decomposes the optimization problem into two subproblems. First, the allocations of the functions that maximize the task reliability are determined using a method based on the branch-and-bound technique. Next, channel delays are analyzed using queueing network model to estimate the task packet delay for each allocation resulted with maximal reliability. The optimal allocation solution is then chosen to be the one with minimal delay. This algorithm is useful in handling large distributed systems because it does not require the evaluation of the task reliability at each iteration. It is fairly efficient due to the 0-1 integer linear programming formulation. Algorithm 2 solves the reliability and delay optimization simultaneously by maximizing a compound objective function that captures both reliability and delay using a similar technique to Algorithm 1.

The proposed algorithms can also be applied to optimize other performance measures along with the reliability of a given set of distributed tasks.

ACKNOWLEDGMENT

The authors wish to thank Dr. F. Parisi-Presicce for several useful discussions.

REFERENCES

[BALL 80] M. O. Ball, "The Complexity of Network Reliability Computations," *Networks,* Vol. 10, 1980, pp. 153-165.

[CHEN 80] P. P. S. Chen, J. Akoka, "Optimal Design of Distributed Information Systems," *IEEE Trans. Computers,* Vol. C-29, December 1980, pp 1068-1080.

[CHU 69] W. W. Chu, "Multiple File Allocation in Multiple Computer System," *IEEE Trans. Computers,* Vol. C-18, October 1969, pp 885-889.

[FRAN 72] H. Frank, W. Chou, "Topological Optimization of Computer Networks," *Proceedings of the IEEE,* Vol. 60, No. 11, November 1972, pp 1385-1397.

[GERL 77] M. Gerla, L. Kleinrock, "On the Topological Design of Distributed Computer Networks," *IEEE Trans. Communications,* Vol. COM-25, No. 1, January 1977, pp 48-60.

[GRNA 80] A. Grnarov, L. Kleinrock, M. Gerla, "A New Algorithm for Symbolic Reliability Analysis of Computer Communication Networks," Pacific Telecommunication Conference, June 1980.

[HARI 86a] S. Hariri, C. S. Raghavendra, "SYREL: A Symbolic Reliability Algorithm based on Path and Cutset Methods," Proceedings of the IEEE INFOCOM 86, April 1986, pp 293-301.

[HARI 86b] S. Hariri, C. S. Raghavendra, V. K. Prasanna Kumar, "Reliability Analysis in Distributed Systems," Proceedings of the 6th International Conference on Distributed Computing Systems, May 1986, pp 564-571.

[HILL 80] F. S. Hillier, G. J. Lieberman, *Introductions to Operations Research,* San Francisco: Holden-Day, 1980.

[IGNI 82] J. P. Ignizio, D. F. Palmer, C. M. Murphy, "A Multicriteria Approach to Supersystem Architecture Definition," *IEEE Trans. Computers,* Vol. C-31, No. 5, May 1982, pp 410-418.

[IRAN 82] K. B. Irani, N. G. Khabbaz, "A Methodology for the Design of Communication Networks and the Distribution of Data in Distributed Supercomputer Systems," *IEEE Trans. Computers,* Vol. C-31, No. 5, May 1982, pp 420-434.

[KLEI 76] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications,* Wiley, New York, 1976.

[KLEI 80] L. Kleinrock, F. Kamoun, "Optimal Clustering Structures for Hierarchical Topological Design of Large Computer Networks," *Networks,* Vol. 10, 1980, pp 221-248.

[LANI 83] L. J. Laning, M. S. Leonard, "File Allocation in a Distributed Computer Communication Network," *IEEE Trans. Computers,* Vol. C-32, No. 3, March 1983, pp 232-244.

[MAHM 76] S. Mahmoud, J. S. Riordon, "Optimal Allocation of Resources in Distributed Information Networks," *ACM Trans. Data Base Systems,* Vol. 1, March 1976, pp 66-78.

[MARC 81] R. Marcogliese, R. Novarese, "Module and Data Allocation Methods in Distributed Systems," Proceedings of the 2nd. International Conference on Distributed Computing Systems, 1981, pp 50-59.

[MERW 80] R. E. Merwin, M. Mirhakak, "Derivation and Use of a Survivability Criterion for DDP systems," Proceedings of the 1980 National Computer Conference, May 1980, pp 139-146.

[PRAS 86] V. K. Prasanna Kumar, S. Hariri, C. S. Raghavendra, "Distributed Program Reliability Analysis," *IEEE Trans. Software Engg.,* Vol. SE-12, No. 1, January 1986, pp 42-50.

[RAGH 83] C. S. Raghavendra, S. V. Makam, "Dynamic Reliability Modeling and Analysis of Computer Networks," Proceedings of the International Conference on Parallel Processing, Augest 1983.

[RAMA 83] C. V. Ramamoorthy, B. W. Wah, "The Isomorphism of Simple File Allocation," *IEEE Trans. Computers,* Vol. C-32, No. 3, March 1983, pp 221-231.

[SATY 81] A. Satyanarayana, J. N. Hagstrom, "A New Algorithm for Reliability Analysis of Multi-Terminal Networks", *IEEE Trans. Reliability,* Vol. R-30, No. 4, October 1981, pp 325-333.

[TANE 81] A. S. Tanenbaum, *Computer Networks,* Prentice Hall, New Jersey, 1981.

[WILK 72] R. S. Wilkov, "Analysis and Design of Reliable Computer Networks," *IEEE Trans. Communications,* Vol. COM-20, No. 3, June 1972, pp 660-678.

DCBL: DATAFLOW COMPUTING BASE LANGUAGE WITH n-VALUE LOGIC

Jayantha Herath*, Nobuo Saito*, Kenji Toda, Yoshinori Yamaguchi, Toshitsugu Yuba

Electrotechnical Laboratory, 1-1-4, Umezono, Sakuramura, Ibaraki 305, Japan.

* Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama 223, Japan.

Abstract

An abstract model of parallel computation, specification of a base language for the abstract model, design of a high-level language and design of an architecture are steps to constructing a computing system. This paper introduces and describes the n-value logic, the NOT(OPERATION) abstract computing model based on the dataflow computing concept, functional programming and n-value logic, its effectiveness over the traditional dataflow computing models and defines DCBL, a Dataflow Computing Base Language based on NOT(OPERATION) model as a base for an efficient new class of dataflow computing languages. The major features of DCBL expressions, the expression - graph translation process, and formal and informal operational semantics are discussed in detail.

1 Introduction

Dataflow computing [1], a radical departure from von Neumann computing, supports multiple processing on a massive scale and has the potential to play a major role in next generation computing machines. In dataflow, an instruction is enabled immediately after the arrival of operands. The computations are free of side effects, and independent computations proceed naturally in parallel. The dataflow approach has the potential of exploiting large scale concurrency efficiently, maximum utilisation of VLSI in computer design, compatibility with distributed networks, and compatibility with functional high-level programming. The dataflow concept can be easily implemented in both major computing application areas: numerical computations such as scientific and technological computations and non-numerical computations such as symbolic manipulation in knowledge processing. Several dataflow computing models have been proposed for implementing dataflow computing concept in practical machines [5], [10] All these models are based on the Dennis's initiative work [1], [2]. Dynamic computing models, proposed separately by Arvind at MIT [3] and Gurd at Manchester, [4] are more efficient than the basic model.

The language is very important in representing parallel algorithms efficiently. Side-effect-free functional languages such as Pure Lisp and FP can be used effectively to execute computations in dataflow computing machines since they do not reflect the properties of von Neumann computing. Dataflow languages [1],[2],[3] should be designed with the dataflow concept in mind. Users should not consider the explicit control of memory allocations in using machines, but deal only with data values. Low-level languages for dataflow computing machines should describe dataflow computing efficiently.

In section 2, the n-value logic and NOT(OPERATION) computing model is introduced and its effectiveness over the traditional model in some important issues in dataflow computing is discussed. Section 3 introduces and defines the dataflow computing base language, DCBL (pronounced decibel) and discusses its specifications. Section 4 outlines a dataflow graph specification language since the operational semantics of DCBL expressions consist of expression translation to dataflow graphs and specifying the functionality of DCBL operators. Here, a simple specification language for parallel system specifications is introduced. Section 5 details the translation process of expressions to dataflow graphs which specify informal operational semantics. Section 6 specifies the functionality of the DCBL operators which give the formal operational semantics of DCBL expressions.

2 NOT(OPERATION) computing model

The NOT(OPERATION) model (see Fig. 1) is based on the dataflow computing concept, functional programming and n-value logic. This model provides very high performance with multi-dimensional parallelism, for parallel computing. Here, the control structures in parallel computations are composed of three primitive functions: sequential computations, parallel computations and decision making computations. Strict rules govern the way each control structure is used, guaranteeing the consistency and completeness of the specifications. Sequencing parallel computations is necessary to ensure logical correctness of the

CH2345-7/86/0000/0353$01.00 © 1986 IEEE

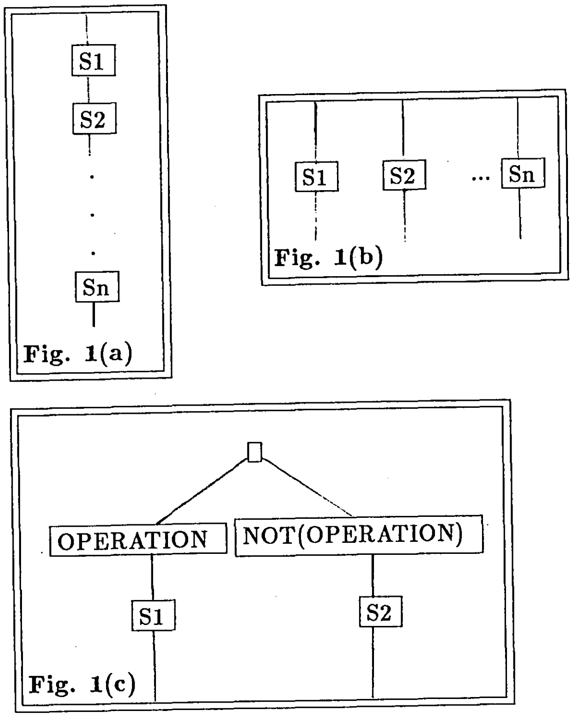


Fig. 1 NOT(OPERATION) abstract model

computations. Sequential computation control functions composed of functions that depend on the previous result of some other function. Fig. 1(a) represents the execution of ordered sequential computations S1, S2, ... Sn. Each successive computation depends on the result of its predecessor. Here, input to the S2 is the output of the predecessor S1. Parallel computations are composed of functions independent of the other for data. They are executed in parallel. Fig. 1(b) represents the execution of independent parallel computations S1, S2, ... Sn. Decision making computations use n-value logic.

- **n-value logic:** All dataflow computing languages developed so far [1] - [12] represent the class of operational semantics with two-value logic proposed by the traditional model. Executing conditional computations result in two values, TRUE (1) and FALSE (0) boolean values. These values trigger remaining computations.

  In n-value, logic the execution of a conditional computation does not give strictly TRUE or FALSE values but gives any value, including no value, instead. The translation of two-value logic conditional computations to n-value logic is performed in two steps. First, the traditional conditional computation is disintegrated into two complementary basic operations. Both operations must be executed for deadlock free computation. The next step of the translation defines the semantics of the execution. One of the two operations executed will give an output value if the operation is satisfied. 1-value logic generates TRUE boolean value and n-value logic generates any value, including no value, instead of boolean values.

Proposition 1: The traditional conditional computation can be divided into two OR parallel computations. One represents the positive state and the other represents the negative state.

The positive state of a conditional computation is denoted by OPERATION and the negative state by NOT (OPERATION). Two input tokens which will satisfy only one of the operations are required to enable the conditional computation. The parallel computing of conditional computation can be represented as **IF (OPERATION) THEN S1 OR IF (NOT(OPERATION)) THEN S2**. It is shown in Fig. 1(c).

Proposition 2: NOT(OPERATION) provides an efficient facility to implement conditional operations and hence iterative and recursive computations in parallel computing systems.

In 1-value logic the execution of both OR combined operations gives a TRUE boolean value output when the operation is TRUE. No FALSE output is given. Successive computations receiving the TRUE boolean value are executed. Boolean value generation complicates dataflow computing. n-value logic greatly simplifies the conditional computations, iterative and recursive computations in dataflow computing. Here, when the OPERATION is satisfied, input data is given as the result of execution, and the output of the NOT(OPERATION) is frozen. Otherwise, the NOT(OPERATION) gives the data value output while the OPERATION output is frozen. Boolean values can be obtained by executing CONSTANT operations when necessary.

Proposition 3: Boolean value generation in the execution of conditional computations in a dataflow computing environment is not a requirement.

There are three types of conditional computations:

- **IF A(x) THEN B(x) ELSE C(x)** Here, the value of A(x) is used in the successive B(x) and C(x) computations.

- **IF A(x) THEN B(x) ELSE C(z)** Here, the value of A(x) is used partially in the successive B(x) computation.

- **IF A(x) THEN B(y) ELSE C(z)** Here, the value of A(x) is not used in the successive B(y) or C(z) computations.

In the first and second cases, the value of conditional opertion execution is used in the following executions. Therefore, n-value logic is more efficient than 2-value logic. In the third case, B(y) and C(z) are invariants. These computations can be performed independent of A(x) whenever y and z become available and the result of A(x) can be used to trigger the final result. Fig. 2 shows the implementa-

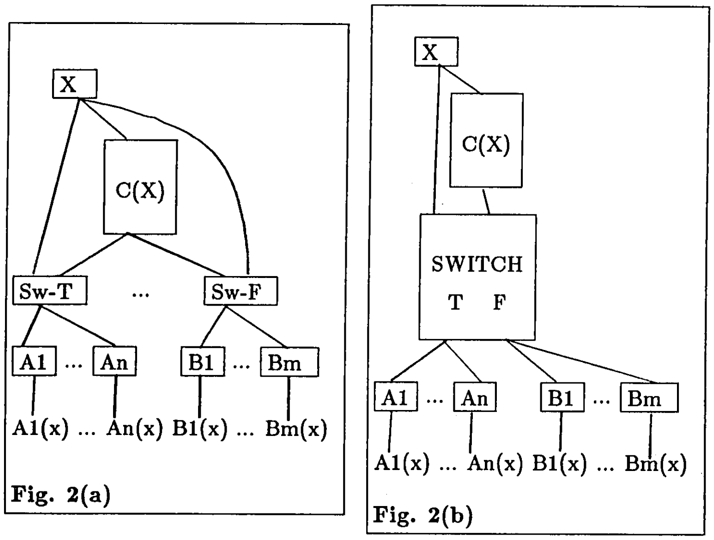


Fig. 2(a)

Fig. 2(b)

Fig. 2 Conditional computation - Traditional

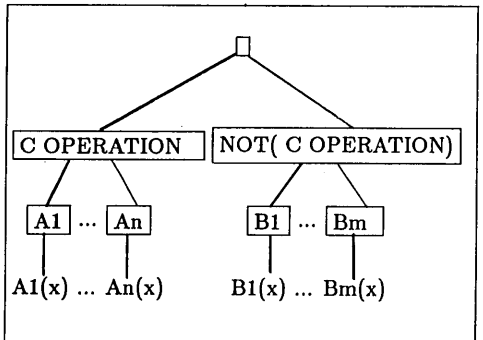


Fig. 3 Conditional computation-NOT(OPERATION)

tion of conditional computation, IF C(x) THEN A1, A2, ... An ELSE B1, B2, ... Bn, with 2-value logic. It is also possible to use a complex, macro version of switch-t and switch-f operations as shown in fig. 2(b) [3]. Fig. 2(c) shows the execution of IF C(x) THEN A1, A2, ..., An IF NOTC(x) THEN B1, B2, ... Bn. Here, the data value 'x' flows to the parallel operations, 'C OPERATION' and NOT'C OPERATION'. If it satisfies the 'C OPERATION' the same 'x' data value is given as the output value and sent to the defined destinations (thin lines in Fig. 3). The output of the NOT'C OPERATION' is frozen (dark lines in Fig. 3). If the data value 'x' satisfies the NOT'C OPERATION' the same data 'x' is given as the output and sent to the defined destinations (dark lines in Fig. 3). Here, the output of 'C OPERATION' is frozen (thin lines in Fig. 3). It is also possible to use a complex, macro version of OPERATION and NOT(OPERATION). Iterative and recursive computations are extensions of conditional computations. Hence, the representation of conditional computations has a large impact on computations. The boolean expression in the general iterative and recursive computations, can be substituted by two OR PARALLEL operations. Hence, the general while loop computation **While(boolean expression) (loop S1); S2, can be represented as (While (OPERATION) (Loop S1)) OR (NOT(OPERATION)(S2)).** and the general recursive computation **f(x) = IF c(x) then f(B(x) else x in f(z)** can be represented as **f(x) = IF c(x) then f(B(x) OR IF NOT c(x) then x in f(z).**

2.1 Effectiveness

It is easy to prove that the number of tokens necessary for the conditional computation and the time taken to enable successive computations of the conditional operation are always less in n-value logic based models than the 2-value logic based models. Several non-numerical and numerical computations were performed in the EM-3 [8],[9],[12],[13]. and the corresponding frequency of operations executed in each benchmark program was measured. These results show that SWITCH operations share a large percentage of all operations executed. The implementation of n-value logic in the EM-3 solved some problems in data-flow computing [13]. These results are summarized below.

1. **Matching bottleneck:** The basic computing cycle of double operand operations consist of: 1. matching operands 2. fetching instructions 3. executing instructions 4. dispatching results to destinations. The basic computing cycle of single operand operations consist of 2, 3, and 4 steps. A dataflow processing element basically consists of a matching unit, execution unit and a destination unit. (See Fig. 4.) The execution unit performs the execution and structure handling. The matching unit sequentially matches and synchronises the operands for execution. The larger the number of double operand operations the more matching to be performed in the matching unit. This narrows the pipeline between the matching unit and execution unit and reduces the computing speed. This is the matching bottleneck. Double operand switch operation is the major source of this. The traditional dataflow computing model reduces the computing speed described by the concept at the very first level of implementation. The implementation of n-value logic decreased significantly the number of single operand packets, double operand packets and operations executed. This results in a proportionate reduction of computing and communication cost and increase in computing speed. Fig. 5 shows machine cycles needed for single operand and double operand conditional computations with 2-value logic and n-value logic. Performance evaluation measurements in the EM-3 support these conclusions.

2. **Remaining packet garbage:** In dataflow computing, a large number of unexecuted packets wait in the matching unit until they are removed by some process. This is remaining packet garbage, RPG.

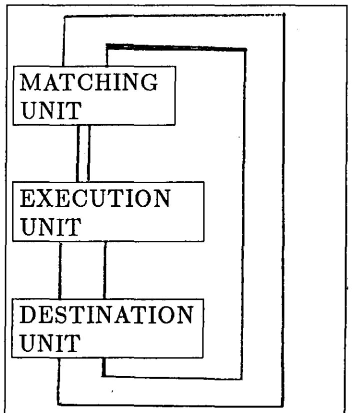


Fig. 4 Data-flow PE

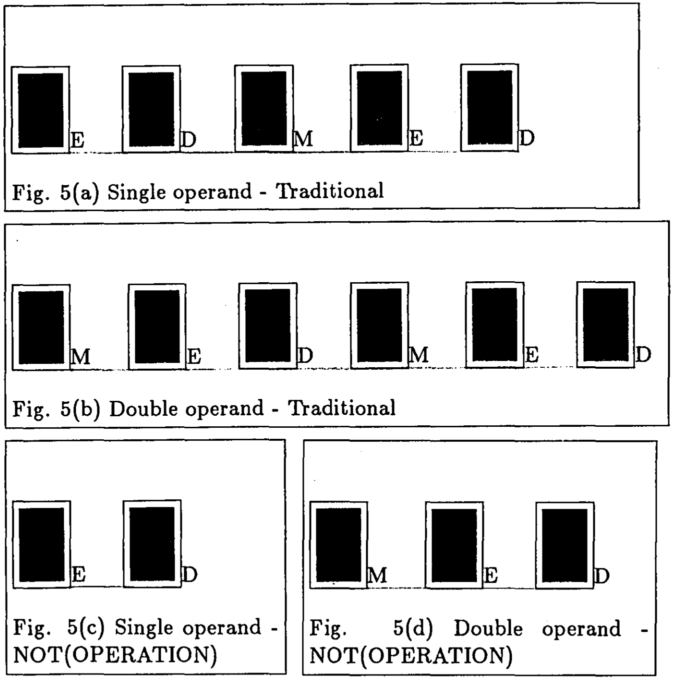


Fig. 5(a) Single operand - Traditional

Fig. 5(b) Double operand - Traditional

Fig. 5(c) Single operand - NOT(OPERATION)

Fig. 5(d) Double operand - NOT(OPERATION)

Fig. 5 Machine cycles

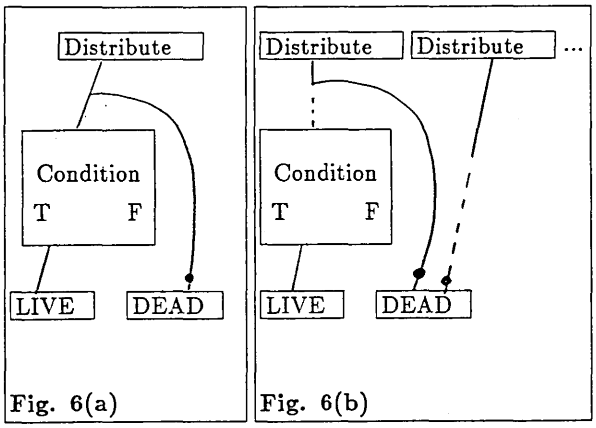


Fig. 6(a)

Fig. 6(b)

Fig. 6 Remaining packet garbage

Vertical branches created by conditional computations and multi-argument functions generate RPG. Fig. 6(a) shows RPG generation with single argument functions. A conditional computation divides the dataflow computing into two vertical branches. Execution of the conditional computation makes one branch LIVE and the other DEAD. Data flow to the operations ignores the liveness of the branch. Loading only one operand of a double operand operation in the DEAD branch results in RPG. Fig. 6(b) shows RPG generation with multi-argument functions. The multiple arguments in a function divide the dataflow computation into live vertical branches. Here, RPG is created when a DEAD branch of a conditional computation in one vertical branch receives the data values from the same vertical branch and/or from some other vertical branch. The n-value logic completely stops the flow of data from just above the conditional computation to the DEAD branch of the conditional computation. A control operation to stop the data flow into the DEAD branch, setting life time to double operand packets, dispatching a special COMMANDO packet to execute all the operations in the DEAD branch or other efficient garbage collecting mechanism must be implemented to completely remove RPG.

3. **Control of parallelism:**. Dataflow computations have huge parallelism, many times larger than the parallelism available in the hardware. Such computations tend to use excessive amounts of storage, since many partial results are created long before resources are available to process them. Therefore, it is necessary to restrict excess program parallelism to approximately match machine parallelism. A large number of unnecessary computations with pseudo parallelism in the traditional model is removed by n-value logic. Dataflow computing requires very large, complex instruction sets. n-value logic provides the facility to design an efficient, refined and reduced instruction set.

4. **Sequential computing segments:** A sequential computing segment is a program segment in which the maximum parallelism is less than the number of processors and/or pipeline stages. Such computations are involved in conditional computations, recursive procedure calls and iterative computations which uses previous computing results to continue computation. The performance of the sequential computing segments in a program is important in parallel computing. Dynamic dataflow computing models unfold loops to compute iterative computations dynamically. n-value logic eliminate sequential computing segments implemented in loop unfolding and gives high computing speeds.

3 Specification of DCBL

This section discusses the specifications of a base language for the NOT(OPERATION) model. The main objective of DCBL design is to introduce and define a new class of operational semantics with n-value logic for dataflow computing languages. The other objective is the development of high-level dataflow computing languages which are comfortable for machines and users to express many forms of concurrency. The language considered here is a preliminary version of the DCBL, therefore, it does not include features such as static arrays or dynamic arrays such as streams, records or unions. Such extensions are considered later. DCBL, a VAL like language [1], [2] allows user selected, highly concurrent algorithms to be expressed as a collection of expressions. Therefore, the execution of a DCBL program consists of a sequence of parallel executions of side effect free expressions. All individual operations in a DCBL expression are executed simultaneously. An expression execution may generate zero (no value expression), one (uni-value expression), or more than two (multi-value expression) values. Tuple expressions, multi-value function expressions, conditional expressions and parallel expressions are multi-value expressions. The specification of BNF syntax for iterative computations of DCBL is given below.

```
exp ::= function(exp)
| exp, exp, ... exp
| IF exp THEN exp IFNOT exp THEN exp
| identifiers
| constants
| LET idlist = exp IN exp
| IF exp THEN exp
| FOR idlist = exp DO iteration

iteration ::= ITER exp NOTITER exp
| LET idlist = exp IN iteration
| IF exp THEN iteration
| IF exp THEN iteration
IFNOT exp THEN iteration
idlist ::= id
| idlist id
```

The application of a function to an expression, **funct(exp)**, is used to represent sequential computations. The elementary functions are operators. The operations performed on expressions can be characterised by mathematical functions. The application of function F to the imports, x, y, and z, produces export F(x,y,z). The expression tuple |**exp, exp, ... exp** | is used to represent parallel computations. **Identifiers** and **constants** are the most elementary expressions in DCBL. Values can be bound to identifiers. Identifiers can be bound to simple types (integer and real), structured types and function calls. The **LET ... IN ...** expression provides local binding to extend the execution environment.

Decision making computations, conditional expressions and the **FOR ... DO ...** expressions in DCBL, sequence the parallel computation to ensure logical correctness and avoid initiating computations whose results can never be used. Huge collections of IF-THEN rules are used in numerical and non-numerical computations, especially in expert systems. Therefore, in the DCBL design, special attention is given to the implementation of conditional expressions. DCBL has a special conditional expression, representing the features described in the n-value logic. Therefore, DCBL has new operational semantics for conditional, iterative and recursive computations. In DCBL, the general IF THEN ELSE expressions and case expressions are represented by the **IF exp THEN exp** conditional expression. All predicates in **IF exp THEN** are supported by expressions, and depending on the imports, produce and seize exports, instead of producing boolean values. **IF exp THEN exp** provides the single branch conditional expression. **IF exp1 THEN exp2 IFNOT exp1 THEN exp3** is a two branch conditional expression with two complementary conditional computations. These expressions are executed in parallel. Combining n **IF exp THEN exp** expressions, n parallel computations with a live branch and n-1 dead branches, provides guarded command feature.

Sequential expressions of the type of iterative computations that depend on the previous iterative computation result are executed sequentially. However, the hidden parallelism of the iteration expression is exploited and the computing speed is increased by the NOT(OPERATION). The **FOR idlist = exp DO iteration** expression implements loops that cannot execute in parallel because values produced in one iteration must be used in the next. The **FOR** expression has loop initiation and a loop body. Loop initiation is performed by the **FOR idlist = exp** part and the loop body appears in **DO iteration.** Side effect free conditional expressions are included in the decision making for loop iteration. The iterative expression **FOR idlist = exp DO iteration** is evaluated by binding the iterative identifiers, the elements of **idlist**, to the values of **exp**. The evaluation of the iteration body results in a **NOTITER** expression and an **ITER** expression. Both these expressions are evaluated concurrently in each iteration. If the **NOTITER** expression satisfies the condition, this terminates the iteration and gives the computation result. Otherwise, the output is given by **ITER** expression. Here, the **ITER** expression is satisfied and continues iteration. The iteration is terminated when the evaluation of the **ITER** body results in an ordinary **NOTITER** expression. The value of this expression is the value of the

ITER expression. Parallel expressions for the computations of the type **For I := 1 to n do C[i] := A[i] * B[i]**, which represents iterations that do not depend on the previous computation result, are not discussed here. The following section describes the dataflow graph specification language.

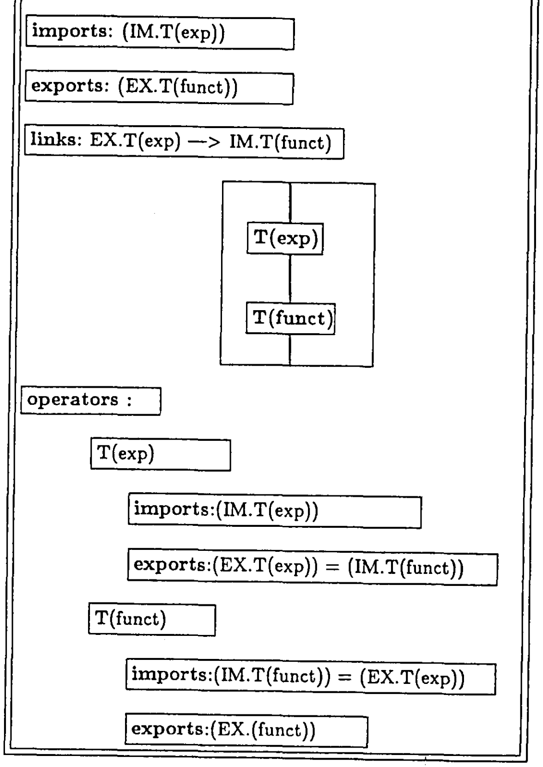
4 Dataflow graph specification language

Expressions and compiler help identify concurrency in algorithms and their program and map that concurrency into graphs. The translation of DCBL expressions gives new dataflow graphs presenting the NOT(OPERATION) abstract computing model. The graph, which connects sub-graphs composed of operators, is an explicit representation of the concurrency available in evaluating expressions. An element of a dataflow computation consists of import ports, imports, export ports, exports, import links, export links and operators. Specifications of a dataflow graph include imports, exports, data links and operators. Specification of operators is defined recursively using local imports and exports. Imports to the operator embark at import ports. Exports of the operators disembark at export ports. The number of imports or exports in a link is unlimited. This gives the dynamic computing features [3] [4]. The restriction of values to one gives the static computing feature [1] [9] [11]. The operators communicate values through their import and export ports. The graph has an import port for each free variable of the expression and an export port for each value returned by the expression. The following notations are used to specify the dataflow graphs. **IM.T(exp) and EX.T(exp)** are the set of imports to the operators of the translation of (exp) and the exports at the export ports of the operators of the translation of (exp). These imports and exports have defined import ports and export ports. Links are represented by **EX.T(expl)—>, IM.T(exp2)** which means that the exports of translated expression expl are linked to the defined import ports of the translated expression exp2 as imports. The import ports of all parallel sub-graphs are assigned the set of import values. The graph export ports are formed by concatenating the export ports of the component sub-graphs. An extension of this language will provide facilities to specify parallel systems.

5 Translation of DCBL expressions to graphs

Transfer function T maps DCBL expressions to dataflow graphs and functionality of the DCBL operator, F, map imports onto exports. The operational semantics is defined and derived by the application of F(T(exp)). This section discusses the expression specification translation to graphs and gives the informal operational semantics of the dataflow graphs.

Figure 7 **[T(funct(exp))]**

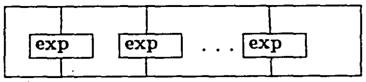


The translation of funct(exp), T(funct(exp)), is shown in Fig. 7. This shows sequentially connected dataflow sub-graphs. The translation is made by connecting the export ports of T(exp) to the import ports of T(funct). Fig. 8 shows the translation of [T(expl, exp2, ... expn)] which consists of n sub-graphs, [T(expl)], [T(exp2)], ... and [T(expn)], that can be executed in parallel. The T(let idlist = expl IN exp2) is shown in Fig. 9. Fig. 10 shows the implementation of the simplest conditional expression **IF expl THEN exp2**. Two sub-graphs are connected sequentially in this graph. Predicate expl controls the evaluation of exp2. No special gates are used in this translation. The import data value is the export of T(expl) if this data satisfies the condition expressed by **expl**; if not, the data value is simply absorbed. This expression provides the facility to evaluate n parallel conditional expressions. A simple extension of the conditional expression is the parallel execution of complementary expressions, **IF expl THEN exp2 IFNOT expl THEN exp3**. Fig. 11 shows the translation of this expression. The translation of identifier, [T(id)], gives a graph with no operators. The translation of a constant expression gives the **const** operator with import export links. A trigger-value import produces the value **const** as the export.

Figure 8 [T(exp,exp, ... exp)]

imports: (IM.T(exp1) + IM.T(exp2) ... + IM.T(exp))

exports: (EX.T(exp1) + EX.T(exp2) ... + EX.T(expn))



operators:

T(exp1)

imports:(IM.T(exp1))

exports:(EX.T(exp1))

T(exp2)

imports:(IM.T(exp2))

exports:(EX.T(exp2))

T(expn)

imports:(IM.T(expn))

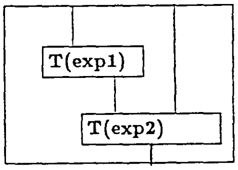
exports:(EX.T(expn))

Figure 9 [T(LET idlist = exp1 IN exp2)]

imports: (IM.T(exp1) + (IM.T(exp2) - EX.T(exp1))

exports: (EX.T(exp2))

links: (EX.T(exp1) —> (IM.T(exp2))



operators:

T(exp1)

imports:(IM.T(exp1))

exports: (EX.T(exp1))

T(exp2)

imports:(EX.T(exp1)) + (IM.T(exp2))

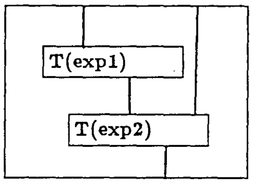
exports:(EX.T(exp2))

Figure 10 [T(IF exp1 THEN exp2)]

imports: (IM.T(exp1) + (IM.T(exp2) - EX.T(exp1))

exports: (EX.T(exp2))

links: (EX.T(exp1) —> IM.T(exp2)



operators:

T(exp1)

imports:(IM.T(exp1))

exports:(EX.T(exp1)

T(exp2)

imports:(IM.T(exp2))

exports:(EX.T(exp2))

The iteration expression **FOR idlist = exp DO iteration** is translated as shown in Fig. 12. DCBL binds identifiers locally. In evaluating **FOR idlist = exp DO iteration,** the elements of idlist are bound to the values of exp, and iteration is terminated when the iteration results in an ordinary expression. Fig. 12.1 shows the translation of **ITERexp NOTITERexp. ITER(exp)** supports iteration if the imports satisfy the expression exp. **NOTITERexp** gives the result of the computation. The iteration body **LET idlist = exp IN iteration** is implemented in the same way as the expression **LET idlist =exp1 IN exp2** is implemented. The dataflow graph implementation of the conditional iteration body **IF exp THEN iteration** is similar to that of the conditional expression. See fig. 12.2. Both subgraphs, IF exp and IFNOTexp, provide a complete set of exports. [T(exp)] and [T(notexp)] are placed on the import paths of the iteration body sub-graphs, [T1(iteration1)] and [T1(iteration2)]. Exports of [T(exp)] or [T(notexp)] enable the evaluation of a selected iteration body.

6 Functionality

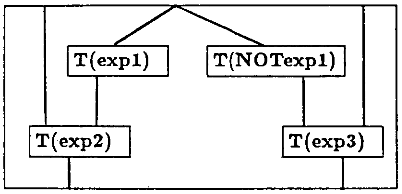
The operational semantics of DCBL expressions represent the formal simulation of dataflow graph execution. The operational semantics of the expression is the functionality of its graph. The graph is mapped onto its semantic characteristics using the functionality. The functionality of a graph is characterised by the functionality of its operators. There are two types of operators; one produces exports in the execution with the arrival of imports, and the other produces and seizes or freezes the exports, depending on the arrival of imports. In the first type, the availability of all imports enables the execution of an operator and produces exports and export via export ports to defined destinations. The destination of an export value is specified by the number of the import port of the destination operator. The export port of an operator is connected by a link to the import port of another operator; therefore, the export value of one operator is the import value to another operator. In the second type, an operator receives import values at each import port and produces or seizes export values at the export port during the execution. Producing an export value enables successive computations. These operators are used in the implementation of conditional, iterative, and recursive computations.

Figure 11 [T(IF exp1 THEN exp2 IFNOT exp1 THEN exp3)]

imports: (IM.T(exp1) + (IM.T(NOTexp1)= IM.T(exp1)) (IM.T(exp2) - (EX.T(exp1)) + (IM.T(exp3) - (EX.T(NOT(exp1))

exports: (EX.T(exp2)) = (EX.T(exp3))

links:(EX.T(exp1) —> IM.T(exp2)) + EX.T(NOTexp1) —> IM.T(exp3))



operators:

T(exp1)

imports:(IM.T(exp1) = EX.T(NOTexp1))

exports:(EX.T(exp1))

T(NOTexp1)

imports:(IM.T(NOTexp1))

exports:(EX.T(NOTexp1) = IM.T(exp1))

T(exp2)

imports:(IM.T(exp2))

exports:(EX.T(exp2)) 8

T(exp3)

imports:(IM.T(exp3))

exports:(EX.T(exp2))

The operational semantics of a dataflow operator is given by its functionality, which maps its imports onto exports. The specifications of the functionality of various dataflow operators are discussed in this section. The functionality of an operator is the usual arithmetic or boolean function associated with it. For example, F+(x,y) = x+y and Fconst(x) = const. Here, x triggers the operator, const, to give the export defined by the const operator. The functionality can be extended for an ordered set of dataflow imports. The operator, plus, can be performed to the ordered sets x.X and y.Y where x represents the first value of one ordered set, X represents the rest of that ordered set, y represents the first value of the other ordered set, and Y represents the rest of that ordered set. Hence, Fplus(x.X, y.Y) = Fplus(x,y). Fplus(X,Y)
= (x+y). Fplus(X,Y).

In executing conditional operators, the data value imported is exported when it satisfies the conditional operator; if it does not satisfy the conditional operator, no export values are produced. The pair Fcond(x) and Fnotcond(x) is the complementary set of conditional operators that can execute concurrently. Here N implies frozen or seized exports.

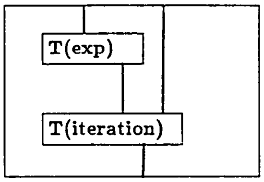
Fcond(x) = x if x satisfies the condition
Fcond(x) = N if x does not satisfy condition
Fnotcond(x) = x if x not satisfies the condition
Fnotcond(x) = N if x satisfies condition

Figure 12 [T(FOR id1...idn = exp DO iteration)]

imports:(IM.T(exp) + IM.T(iteration) - EX.T(exp))

exports:(EX.T(iteration))

links:(EX.T(exp) —> (IM.T(iteration))



operators:

T(exp)

imports:(IM.T(exp))

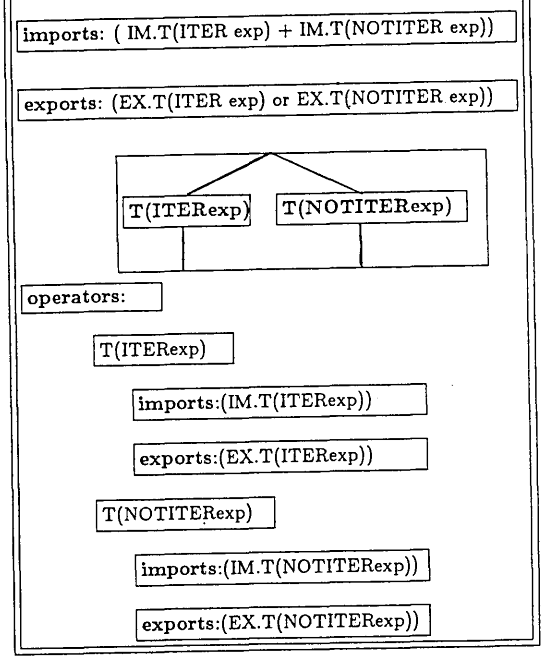
exports:(EX.T(exp))

T(iteration)

imports:(IM.T(iteration) + EX.T(exp))

exports:(EX.T(iteration))

Figure 12.1 [T(NOTITER exp, ITER exp)]



imports: (IM.T(ITER exp) + IM.T(NOTITER exp))

exports: (EX.T(ITER exp) or EX.T(NOTITER exp))

operators:

T(ITERexp)

imports:(IM.T(ITERexp))

exports:(EX.T(ITERexp))

T(NOTITERexp)

imports:(IM.T(NOTITERexp))

exports:(EX.T(NOTITERexp))

The operator will not execute without its complete set of imports.

F(X, Y, ...) = e if X or Y ... = e = empty

Foper(e) = e

Fconst(e) = e

Fcondop(e) = e

Fcondop(e,x) = e

Fcondop(x,y) = x or y (predefined)

The functionality of DCBL operators for list operations, numerical operations and conditional operations are shown below Here, E implies error value export.

List operations

```
Ffirst((x1 x2 ...)) = x1            Ffirst(()) = E
Frest((x1 x2 ...))  = (x2 x3 ...)   Frest(())  = E
Fcons(x1 (x2 ...)   = (x1 x2 ...)
```

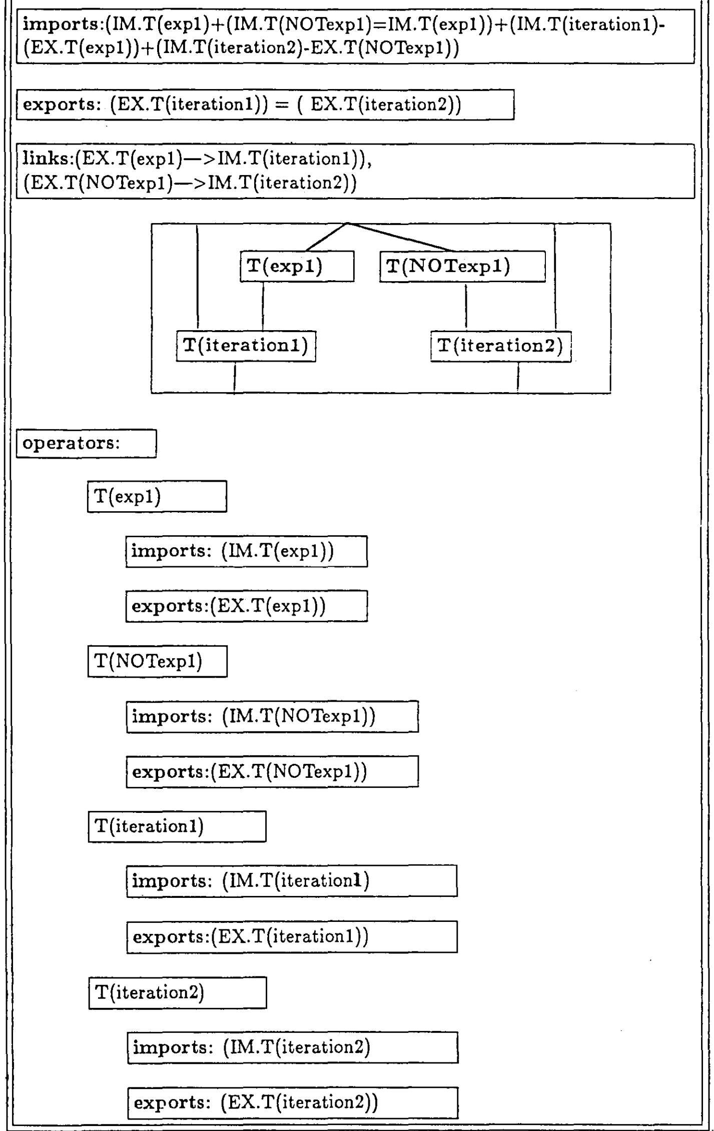
Numerical operations

```
Fplus(x y)       = x+y   Fdifference(x y) = x-y
Fquotient(x y)   = x/y   Fremainder(x y)  = rem x/y
Ftimes(x y)      = x * y
```

Conditional operations

```
Fnull(())      = ()  Fnotnull(x1 ...)  = (x1 ...)
Fnull(x1 ...) = N    Fnotnull(())      = N

Fatom((x))      = (x)   Fatom((x1 ...) = (x1 ...)
Fatom((x1...)) = N      Fnotatom(x)    = N
```

Figure 12.2 [T(IF exp1 THEN iteration1 IFNOT exp1 THEN iteration2)]



imports:(IM.T(exp1)+(IM.T(NOTexp1)=IM.T(exp1))+(IM.T(iteration1)-(EX.T(exp1))+(IM.T(iteration2)-EX.T(NOTexp1))

exports: (EX.T(iteration1)) = (EX.T(iteration2))

links:(EX.T(exp1)—>IM.T(iteration1)),
(EX.T(NOTexp1)—>IM.T(iteration2))

operators:

T(exp1)

imports: (IM.T(exp1))

exports:(EX.T(exp1))

T(NOTexp1)

imports: (IM.T(NOTexp1))

exports:(EX.T(NOTexp1))

T(iteration1)

imports: (IM.T(iteration1)

exports:(EX.T(iteration1))

T(iteration2)

imports: (IM.T(iteration2)

exports: (EX.T(iteration2))

```
Fnumberp(1)      = 1   Fnotnumberp(1 ...) = 1 ...
Fnumberp(1 ...) = N    Fnotnumberp(1)     = N

Fequal(x x)      = x   Fnotequal(x x)     = N
Fequal(x y)      = N   Fnotequal(x y)     = x

for integers x greaterthan y
Fgreater(x y)     = y   Fnotgreater(x y)    = x
Fgreater(y x)     = N   Fnotgreater(x y)    = N
```

7 Conclusions

Parallel processing systems give ultra-high computing speeds. The dataflow computing concept seems to be the most effective, promising computing method to implement in machine architecture for high speed computing. In this paper, n-value logic and the NOT(OPERATION) abstract computing model were introduced and their effectiveness over traditional models was discussed. In general, the NOT(OPERATION) model increases the speed of the dataflow computing by decreasing machine cycles for conditional computations, reduces the number of operations, especially double operand operations in the computation, reduces the computing and communication costs, and remove an RPG source.

All dataflow computing languages proposed so far are based on the 2-value logic. The preliminary version of the dataflow computing base language using the n-value logic based NOT(OPERATION) model, the DCBL, was introduced and defined. The BNF specifications of DCBL and a dataflow computing graph specification language were presented. The new class of operational semantics was described in detail by translating the DCBL expressions to dataflow graphs and defining the functionality of the dataflow graphs.

Acknowledgements

We wish to thank Dr. Shimada, Dr. Hiraki and Dr. Uchibori of Electrotechnical Laboratory for their constant support and helpful discussions. We also extend our thanks to Dr. Nakamura, Mr. Kitagawa, Mr. Yamaoka, Mr. Sakurai and other members of Saito lab. Special thanks to Susantha and Rosemary for helping in many ways to produce this paper.

References

[1] J. B. Dennis, W. Y. P. Lim, W. A. Ackerman, "The MIT Dataflow Engineering Model", *Proc. of IFIP* (1983), 553-560.

[2] J. D. Brock, "Operational Semantics of a Data flow Language", *MIT/LCS/TM-120*

[3] P. Arvind, V. Kathail, K. Pingaley, "A Dataflow Architecture with Tagged Tokens", TM-174, Lab. Comp. Sci., MIT (Sept, 1980).

[4] J. Gurd and I. Watson, "Preliminary Evaluation of a Prototype Dataflow Computer", *IFIP* (1983), 545-551.

[5] T. Yuba, "Research and Development Efforts on Data-flow Computer Architecture in Japan", *SICOB*, (1984).

[6] K. Hiraki, K. Nishida, S. Sekiguchi, T. Shimada "Maintenance Architecture and LSI Implementation of a Dataflow Computer with a Large Number of Processors", *Proc. of Int. Conf. on Parallel processing'86* (1986) 486-490.

[7] T. Shimada, K. Hiraki, K. Nishida, "An Architecture of a Dataflow Machine and its Evaluation", *Proc. of COMPCON '84 Spring* (1984) 486-490.

[8] Y. Yamaguchi, K. Toda, T. Yuba, "A Performance Evaluation of a Lisp Based Data-driven Machine (EM-3)", *Proc. 10th Ann. Int. Symp. Comp. Arch.*, (1983) 363-369

[9] Y. Yamaguchi, K. Toda, J. Herath, T. Yuba, "EM-3: A Lisp Based Data-driven Machine", *Proc. of Int. Conf. on Fifth Generation Computing Systems*, (Nov. 1984) 524-532.

[10] Vason P. Srini " An Architectural Comparison of Dataflow Systems", *IEEE COMPUTER*, (March 1986) 68-88.

[11] A. A. Faustini, S. G. Matthews, A. AG. Yaghi, "The PLUCID Programming Manual", *Technical Report, Arizona State University, TR-83-004* (October 1983).

[12] J. Herath, "Performance Evaluation of a Data-driven Machine Using a Software Simulator", *Masters Thesis, Univ. of Electrocommunications, Tokyo, Japan*, (March 1984).

[13] J. Herath, N. Saito, K. Toda, Y. Yamaguchi, T. Yuba, "NOT(OPERATION) for High Speed Dataflow Computing Systems", *Proc. of Int. Conf. on Super Computing Systems*, (Dec. 1985) 524-532.

Evon: an Extended von Neumann Model for Parallel Processing

Wai-Mee Ching

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

Abstract

We propose an extended von Neumann model called Evon for parallel processing. It is centrally controlled, and is capable of exploiting instruction level as well as expression level parallelism inherent in high level language programs. Its effectiveness in exploiting parallelism depends crucially on powerful primitives, and we illustrate this point by several short programming examples written in APL. We then report our work on the design of an instruction set embodying the Evon model, and a portable compiler for that instruction set aimed at extracting parallelism automatically. Finally, we compare Evon with other computational models and architectures proposed for parallel processing with pragmatic considerations and programming concerns.

1. Introduction

Different approaches to parallel processing can be discussed around several distinct axes. The first axis is whether the machine is of a special-purpose or a general purpose nature. For special-purpose parallel machines we have seen examples like the Yorktown Simulation Engine which is about 5000 times faster than what is available on sequential machines [9],[19]. For general-purpose machines, on the other hand, only moderate parallelism has been achieved in actual use such as on the Cray X-MP. In the search for general-purpose highly parallel architectures, many computational models and accompanying architectures have been proposed. This constitutes the dominant axis in the discussion of parallel processing. ' Among the contending models of computation, the following are the most actively pursued in research on parallel processing: control-driven or multiprocessing von Neumann style, data-driven or data flow, and demand- driven or reduction (see [1],[14]). Furthermore, there are also hybrid models like Rediflow [18] and Eazyflow [16]. Different computational models emphasize the exploitation of parallelism at different levels of granularity. Generally speaking, multiprocessing von Neumann architectures aim at procedure-level parallelism while data flow aims at expression-level. Among multiprocessing von Neumann models, there is the distinction of SIMD and MIMD, differences in switching network interconnection topologies and whether packet-switching or circuit switching is used.

Treleaven and Hopkins [24] pointed out that the success of the von Neumann model is related to its general-purpose nature. The flourishing of contending models for parallel processing in contrast to the limited parallelism available for general-purpose applications so far bespeaks the great difficulty in identifying a truly general-purpose model of parallel computation. We believe the success of the von Neumann model also has to do with its conceptual simplicity, and practicality in terms of hardware implementation and programmability. Thus, another important axis which is often neglected in discussing parallel processing is the set of issues involving programming and languages. Should parallelism be explicitly specified or implicitly programmed and extracted by a high-level language compiler? Should we use an established language or a brand new language for parallel processing (see [21],[20],[15]). Compilers have traditionally played second fiddle to hardware projects in parallel processing. Two exceptions are the VLIW architecture and the ELI-512 machine of Fisher [10] with a compiler doing trace scheduling, and University of Illinois's Cedar Project with Parafrase [11].

Our approach to parallel processing is also compiler-oriented. We propose an extended von Neumann model, called **Evon**, for parallel processing. Evon is centrally controlled globally but functional and data independent instructions can be executed simultaneously, similar to the ELI-512 machine of VLIW architecture which has a single instruction stream but is not a traditional SIMD machine. Like the very long instruction word in ELI-512, each instruction in Evon can potentially command a substantial amount of hardware to operate in parallel. Like in Cedar, Evon uses powerful (compound) functions to organize computations to be carried out in parallel, and also applies the principle of data flow at certain levels. As both in VLIW and Cedar, Evon is accompanied by a sophisticated compiler responsible for extracting parallelism from programs written in a high-level language, so programmers are free of the job of parallel decomposition and subtasks synchronization. Evon differs from both VLIW architecture and Cedar in the implicit assumption of the high-level language most likely, or most suitable, to be used on proposed parallel machines. Both ELI-512 and Cedar assume that programs are written in a traditional scalar von Neumann language like Fortran, while Evon assumes that programs are written in a vector von Neumann language like APL. We shall see from programming examples later that this change of language does have great significance on previous assumptions on what are the main impediments to execution in parallel of a high-level language program. Although Evon is very much like a vector

CH2345-7/86/0000/0363$01.00 © 1986 IEEE

supercomputer, it will become clear in the following sections that Evon can afford a much higher degree of parallelism than that offered by traditional vector processors.

We shall report our research work on defining an instruction-set E-code as a particular embodiment of the Evon model, and the work of a compiler (called E-compiler) which translates a substantial subset of present APL into E-code. The work on E-compiler is but a first step in our effort to achieve the goal of automatic extraction of inherent parallelism in high-level language programs. The successful completion of its first phase can provide us a valuable tool in studying inherent parallelism in various programming tasks much like Parafrase did for scientific programs written in Fortran. But we will be able to cover a wider area of applications including scientific and engineering computations, design automation algorithms and financial data processing.

2. The Evon Instructions and Instruction-level Parallelism

Like the classical von Neumann model, Evon has a central processing unit (CPU) and a linearly addressed global memory connected to the CPU by a bus of sufficient bandwidth. The memory is interleaved so as to increase its effective bandwidth. For example, if it is organized into 32 banks, then the famous von Neumann bottleneck is still there but its capacity has been increased by 32-fold. Obviously, an increase in the memory bandwidth has to be matched by an increase in the concurrency of the CPU operations to boost overall performance.

In the von Neumann model each instruction either moves one scalar datum from memory to CPU, manipulates a scalar datum in the CPU's execution unit or stores a datum in CPU to memory. In Evon each instruction can manipulate scalar as well as vector data, move a segment of vector data or a scalar datum from CPU to memory, or vice versa. Evon manipulates four primitive types of data: boolean, fixed-point number (integer), floating-point number (real) and character, both in scalar and vector. An Evon can have all the scalar instructions of some popular von Neumann machine, but most of all an Evon has vector instructions. The vector instruction-set repertoire of Evon certainly contains all those arithmetic operations found in a typical vector processor. But it must further include the following

1. calculation of all partial results of a binary arithmetic operation on a vector which corresponds to the scan operator \ in APL and includes the special case of reduction;
2. selection and expansion of vectors according to a boolean vector;
3. finding membership and indexes of a vector of elements with respect to another vector.
4. rotation and shift of vectors.
5. shuffle exchange of elements of a vector (This one has no corresponding APL primitive).

However, it shall not include functions of a special nature like matrix multiplication, inversion, or a random number generator. We also note that in the basic model the length of a vector is not restricted. But in any particular hardware realization there must be some limit mvl put on the length of a vector in cpu (we envision that 128≤mvl≤1024 for a machine not geared towards any particular scientific application). In other words, when a vector in memory is longer than the limit mvl imposed by CPU, the vector has to be wrapped around a real cpu vector register with masked tail.

In order to pursue our research in parallel processing using the Evon model and to make our case more concrete, we have designed an instruction-set, E-code, for a hypothetical simplified Evon machine. A sketch of the machine organization is given in Figure 1. The machine has four vector accumulators of a certain maximal length: BV, CV, IV, EV, one for each data type: boolean, character, integer and floating-point. The effective length of a vector residing in one of the vector accumulator is controlled by a (internal) length register BL, CL, IL, or EL according to its type. It also has several scalar registers R0-R15 and F0, F2, F4 and F6 as in System 370. E-code, like System 370, is a two-address code, i.e. one of the operands serves as the destination, which can be a scalar register, a vector accumulator, or a memory location in case of store operations. Further description of E-code is in [4]. A basic implementation of E-code on System/370 has already been accomplished. We remark that E-code is introduced here only as a convenient tool for discussing machine-code translation of some programming examples. Its underlying machine organization is not intended to represent an ideal implementation of the Evon model.

In general, an Evon CPU has three parts: (1). the I-box for instruction decoding, scheduling and issuing. (2) the M-box for memory address calculation, register file and memory accessing and data transfer. (3) the E-box for the functional execution of arithmetic and logical operations. The E-box is a collection of functional elements including operational units for +, ×, =, etc. For each arithmetic function, there are separate scalar and vector execution elements. For each vector arithmetic functional element, there are duplicate hardware execution units to carry out the actual calculation in segments of a vector. For example, there can be 32 floating-point multipliers. However, the number of duplicates can vary among functional elements. This has to do with the relative spread or the number of pipeline stages of each function's execution unit. For example, there should be more fixed-point adders than multipliers. Duplicating hardware units would overcome the limitation on effective parallelism by the pipeline stages of traditional vector processors.

It is well known that typical vector instruction can save the repeated instruction fetches and decodes in a vector loop on a conventional machine. But Evon incorporate further parallelism with the following two features which are absent in all vector machines:

First, Evon hardware supports the parallel execution of some important compound operations listed in 1) - 4) above. In particular, operand registers and the array of execution units of a functional element can be hardwired to achieve log n time for calculating all partial results of a particular binary operation on a vector. We illustrate the basic idea for a vector of 8 elements in Figure 2. Let us say the particular functional element is addition. At the first row, 8 elements v0..v7 of a vector v are sitting in 8 registers r0..r7. After first round of parallel operation by 4 adders, the even numbered registers' contents are not changed while the odd numbered registers all contain the sum of the original content and it left-neighbor. We note that a joint of two lines in Figure 2 represents the employment of an operational unit, like an adder, of a functional element. After third round of operation, the total sum is in r7. At the end of 4(=1+log8)-th round of operation, all partial sums are calculated and reside in r0-r7 (See [5] and [6] for detail). The instructions in group 4) can also be implemented by circuits doing a parallel associative search on a vector register with respect to another (However, Evon is not an associative processor because it does not employ associative memory).

Second, Evon expects each basic block of instructions (defined in the next section) to be grouped into streams by a compiler. A **v-stream** (of instructions) is an instruction sequence such that 1) all involve vectors of the same length (or all scalars) and 2) the result of one instruction is used as an operand of the next. The grouping of instructions into v-streams by a compiler represents static chaining. The chaining technique of the Cray-1 which permits successive (vector) operations to be issued as soon as the first result becomes available as an operand is well known. And we note that the chaining in Cray-1 is done at run-time while in an Evon it is going to be done at compile time. The effect of (static) chaining is that of loop fusion of the controlling micro-code. For example, the following line of code in APL

```
E←D+A←C×B
```

translated into E-code:

```
LIA Rl,100
LDI Rl,aB
MPI Rl,aC
STI Rl,aA
ADI Rl,aD
ADI Rl,aE
```

where Rl is a vector length register indicating vector length with respect to an implicit vector accumulator. The particular meaning of these instructions is not important. The point we want to make is that the last five instructions, which either move data between integer vector accumulator and memory or operate on the integer vector accumulator, can be chained together. That is, these five instructions will share the same control structure of micro-instructions. An operation on a vector element of instruction is carried out as soon as the operation of the previous instruction on that element has been completed. At the micro-code level, there is only one length checking for the five loops.

Another example for the first point above is the following: Calculate the population distribution when given B as the base population and A[0], A[1], ..., A[99] as the death rate for age 1 to 99. In PASCAL we have,

```
P[0]:= B;
FOR I=0 TO 99 DO
  P[I+1]:= P[I]*(1-A[I])
END;
```

while in APL it is

```
P←B××\1-A
```

translated to E-code:

```
LIA  Rl,100
LDE  Rl,aA
SBEM 0
ADEM 1
MPEP
MPEM aB
```

where MPEP computes all partial product of the elements in the vector and can be implemented by (n/2)-execution units with a (n/log n) speed-up. We note that this is basically the same example discussed in sec.3 of [15] in a multiprocessing von Neumann style with explicitly programmed synchronization.

Basically, there are two kind of loops (in programs written in scalar von Neumann languages): independent loops, i.e. the data of one iteration is independent of data in the previous iteration, and dependent loops, i.e. the data of one iteration is intrinsically dependent on previous iterations (this means the dependency can not be removed by renaming). The first example of chained vector instructions given above is a loop of the first kind. A vector machine can easily handle a speedup of loops of the first kind by pipelining, and machines of array architecture can also ideally handle this kind of loop. The data dependent loop presents difficulties for parallelization even if it can be recognized by a vectorizer. In general, nothing can be done to speed up such loops besides detecting certain independent portions to be executed in an overlapped fashion. The Evon approach is to use the scan operator of APL as a centerpiece to express such dependency and provide a machine primitive, i.e.instructions of class 1), which controls a large amount of hardware in parallel to carry out the execution in a theoretically optimal time. It is amazing that a large amount of seemingly intrinsically sequential code can have a speedup factor of $(n/\log n)$ where $n/2$ is the number of available execution units in the functional element. For example, the first order linear recurrence equation

```
X[0]:= X0;
FOR I=0 TO 99 DO
  X[I+1]:= X[I]*A[I]+D[I]
END;
```

can also be coded in one line of APL:

```
  Z←(RT⌽((⌽×\(0,L)↓(RT←-⍳L)⌽(SQ⍴⌽A),
(SQ←2⍴L)⍴0),(L←⍴A)⍴1))+.×X0,D
```

and benefits from the hardware supported primitives for fast execution. It is not the virtuosity of one-line coding we want to show. Rather, if a language (and machine) contains a sufficient amount of high-level primitives then we can trade storage for speed in dealing with some important classes of data dependent loops. One line APL code is functional programming, i.e. there is no change of state in the computation. The high-level primitives absorb the control structures originally in the corresponding PASCAL version, and turn the data dependent loop into an expression tree. Besides parallelism exploited in executing each node which roughly corresponds to an Evon instruction, many parts of the tree can be executed in parallel as we shall show in the next section.

From the perspective of scalar von Neumann languages, very little parallelism can be gained from instruction-level parallelism. But in the extended von Neumann model Evon where many hardware supported high-level primitives are available, the situation becomes quite different. We can profitably utilize instruction-level parallelism to speedup computation considerably.

3. Basic Block Scheduling and Local Data Flow

Compiled programs are executed on Evon one (maximal) basic block at a time. A (maximal) basic block is a (maximal) group of instructions without a branch instruction except the last one and without a branch into it except to the first one. We shall simple use 'basic block' to mean 'maximal basic block'. If we adopt an APL-like high-level language as a source language, then a basic block consists of a string of expression trees (each tree corresponds to a line of source code) produced by the parser of a compiler. Each of these expression trees can be executed in parallel within the constrains of data dependency and the availability of hardware execution units. Furthermore, the expression trees in a basic block need not be executed in the sequential order of their corresponding source code lines. In other words, Evon is globally centrally controlled in that it executes one basic block of code at a time according to a program flow graph, but locally executes its block of code according to the data flow principle.

Before we discuss the instruction scheduling scheme, let us describe Evon's instruction processing unit, the **I-box**, in general terms. The I-box has an instruction buffer which is large enough to hold all machine instructions in a basic block of average size. The front-end of the I-box is the instruction decoding unit which can decode several instructions simultaneously. For sake of concreteness, we can assume that each instruction is 32 bits wide, and 4 to 8 instructions can be decoded at the same time. This aspect of Evon is very similar to ELI-512's having very long instruction words. The instruction decoding unit has its own adders to calculate addresses. There is a functional elements' usage register, called f-register, to indicate at any machine cycle which function execution elements are idle and which are executing. The back-end of the I-box is a group of independent instruction-stacks. We can assume that there are anywhere from 3 to 8 instruction stacks in a Evon. The top of each stack holds an instruction ready to be issued for execution as soon as a hardware functional unit is available.

We have already introduced the concept of v-streams (of instructions) in the previous section. An expression tree can be decomposed into groups of v-streams of instructions. However, the compiler first calculates internal data dependency, i.e. the dependency created between instructions assigning variables in the basic block of our concern and instructions using these variables as operands. The reason we ignore data inter-block data dependency is simple: compiled programs execute one basic block at a time on Evon. When the code block of a basic block is loaded into the instruction buffer, all instances of exposed-used variables, i.e. variables assumed to be assigned outside of the basic block, should already have correct values. V-streams are then broken into pieces, called **i-sequences** (of instructions) so that only the head instruction of an i-sequence is possibly data dependent on any other instruction in the same basic block.

The basic instruction scheduling scheme for a basic block of code can now be described: Independent i-sequences of instructions in a basic block are decoded and loaded into separate instruction stacks waiting to be executed. In the beginning of the execution a group of i-sequences chosen at compile-time, whose first instructions are all independent of intra-block data and execute on disjoint functional elements, are loaded on instruction stacks and simultaneous execution of i-sequences starts. If there is competition, due to the limited number of instruction stacks and functional elements, between i-sequences of instructions to be in that first group the compiler chooses one by one according to who has the most number of i-sequences data dependent on it. Each instruction will set the occupied bit of the functional element it is going to use and each assignment instructions will set the enable bit on all instructions depending on the variable getting written into that instructions. Decoded and enabled instructions then check the f-register to see if the required functional element(s) is free. If it is, the instruction on the top of an instruction stack is issued for execution. In case of a vector i-sequence, if not all functional elements can be secured at one time, the machine breaks the stream of instructions. That is, an Evon machine would not insist on executing vector loops in a fused fashion. It can execute the loops separately to utilize available functional units. Hence it is very likely that an adder and a multiplier, or a scalar multiplier and a vector one, operate concurrently in executing a compiled program. We note that this type of functional parallelism is available on the CDC Cyber series and on the CRAY-1 where this is carried out by a scoreboard involving registers as well as functional units. Hence the run-time checking in these traditional vector machines for overlapping execution is considerably more elaborate than in Evon.

We shall briefly illustrate the scheduling and intra-block multi-functional parallelism based on compile-time dependency calculation by the following example: Find all the prime number from 2 to N. Unlike in FORTRAN or PASCAL where loops are most likely to be used, we have

```
∇Z←PRIME N;V
[1] Z←2,(~V∊(2↓~⌊N*.5)∘.×2↓⍳⌈N÷3)/
    V←1+2⍳⌊←(N-1)÷2
∇
```

and its parse tree is shown in Figure 3. Basically, it sets up a multiplication table and has the vector of odd numbers up to N be checked against that table. The ones not in the table are prime numbers. The instruction stream corresponding to the right corner subtree calculating V is mutually independent with respect to the instruction stream corresponding to the subtree (2↓..)∘.×2↓..3 rooted at the outer product node which set up a multiplication table. In short, the two subtrees delineated by the two dotted semi-circles in Figure 3 are data independent. They are the main parts of the computation and can be executed in separate hardware components if such components are available. This example shows that Evon can successfully support implicit parallelism at the expression level.

We remark that there are some broad similarities between our approach to the exploitation of expression-level parallelism and that of the expression processor [23], the block-driven data-flow processor [3] and the works on parallel execution of sequential programs [20], [21] and [24]. Our scheduling is much more specific and quite different in detail from that discussed in [3]. In [23], the execution is ultimately carried out in a tree complex of processing elements, each of which has instruction decoding capability. Our functional elements only have the capability to execute specific functions. Most works on the parallel execution of sequential programs assume that the program is already serialized which we do not assume. And we restrict our attention to a basic block. Our v-streams and i-sequences bear resemblance to the parallel execution strings of [24]. But they differ in many aspects. In particular, each instruction is not assumed to take an equal amount of time here, and we have taken the limitation of hardware execution units into consideration.

Both Evon and typical vector machines are aiming at fast execution of vector instructions. But the Evon comes with duplicate hardware execution units, not just the pipelining of one functional unit for each operation, to speedup execution on multiple data, and supports some compound operations like scan as a primitive whereas most vector machines only support reduction at most. More importantly, the Evon model relies on compile-time detection of data dependency among instructions in a basic block and appropriate hardware support to systematically achieve any possible parallelism at expression level which traditional vector machines can hardly handle.

4. Languages and Compilers

Most research in parallel processing of multiprocessing von Neumann style implicitly assumes that the source language is a scalar von Neumann language like FORTRAN or PASCAL. While data flow or reduction architectures usually explicitly require a new data flow language like VAL or Id, or a functional programming language of Backus style. We can also design a new language without blemish for parallel processing but with all desirable features. The problem with a new language is that there usually will be a scarcity of written real-life programs for experimentation (on the effectiveness of any parallel processing model). We can also use some old language like FORTRAN and then develop a sophisticated vectorizer like Parafrase to extract vector operations and piece together compound functions for Evon. This is beyond the scope of the present project. Fortunately, a well established language naturally suited for our purpose is readily available, namely APL. This will save us the demanding job of vectorization because programs in APL are not serialized as in FORTRAN. In fact, programming style in APL emphasizes the importance to turn a problem solution in a vector form whenever possible. Even though APL's usage is not as extensive as that of FORTRAN in scientific and engineering fields, it has been used for long time in wide range of applications: scientific and engineering computations, design automation algorithms and financial calculations. It is very important to have a variety of real-life programs for experimentation in research on parallel processing models for general purpose applications. The fact we choose APL as a source language for experimentation should not confuse our effort with what people usually call an APL machine. As APL's implementation at least so far is by interpreters, a so called APL-machine is mostly aimed at fast interpretation, i.e. it typically has an interpreter as its model of execution. This also includes fast recognition of syntactic units from source code which is a totally irrelevant issue and a waste of time for us. Past efforts in this direction include work on microprogramming a model of 360 to aid interpretation (see [11]).

The problem with using APL is that a compiler is not yet generally available. This hinders our research since we have no compiled code to study, to rearrange and to use for scheduling experiments. Hence our effort after designing E-code is to implement a compiler for a subset of APL to E-code. The subset excludes features which make APL not compilable. It covers about 95% of the language features of APL so most APL programs in scientific and engineering applications can be compiled without any change. In particular, it does not require declaration of variables and their types. Such an extra requirement would defeat our intention to have a ready set of programs for experimentation. The front end of the compiler is basically complete [4]. It contains a component to analyze type and shape (dimensions) of variables based on local type-shape inference and global data flow analysis. Local data dependency calculation can be easily inserted in to the part implemented. We can also add a component to group i-streams and cut them into i-sequences. A major portion of the compiler back end has also been implemented. In fact, we can already compile some simple functions and (together with the 370 E-code simulator mentioned above) run compiled code on System 370. However, due to the extremely large number of primitive functions in APL (66), not all code-generation functions (for corresponding APL primitive functions) have been implemented as of this writing.

A preliminary result of our compiler effort is the confirmation that APL programs do indeed have large basic blocks. The front end of our compiler has successfully processed three APL programs: i) a workspace called SIMPLE which is for impact printer simulation and involves finite element differential equation calculations, ii) a workspace called CUT which is for PLA minimization (in the area of design automation) and involves a large amount of boolean calculations and topological algorithms and iii) part of a workspace called GRAFSTAT which is for statistical analysis and graph drawing (the part not processed by the compiler has to do with so called auxiliary processors in the APL system controlling graphic terminals). We have the following table

Program	Fns	Lines	Blocks	Nodes	Node/Block
CUT	4	241(104-17)	57(22-2)	2014	35.33
SIMPLE	9	64(19-1)	18(5-1)	629	34.94
GRAFSTAT	22	488(71-1)	188(28-1)	6136	32.63

indicating the number of functions in the program, total number of source code lines excluding comments with the maximum and minimum number of lines of functions in brackets and similarly with blocks, the total number of parsing nodes and the average nodes per basic block. We note that the Node/Block number for a typical FORTRAN program is likely to be about 10. Also we note that in APL, operations are more likely to be coalesced, so it actually would use less nodes for the same computation than FORTRAN. Note that the average number of the large blocks, namely those likely to dominate computation time, is likely to be at least twice at large as the over all average. The large size of a basic block is important because it means rich opportunities to apply the local data flow principle to schedule instructions in parallel.

Finally, we remark that these three programs are all used without a corresponding FORTRAN or PASCAL version. That means that there must be sufficient vector operations using APL primitives to compensate for the ineffiency of an APL interpreter so that overall computation times become tolerable in comparison with possible FORTRAN versions. This indicates good opportunities for the instruction level parallelism in these programs. We hope to process and study more existing programs from more areas of applications. We also intend to write new APL programs for some computation intensive jobs for which only FORTRAN versions exist at present.

5. Comparison with Other Parallel Architectures

The multiprocessing von Neumann model and the data flow model are two of the most actively pursued directions in parallel processing research. There are critiques of each model offered by proponents of the other (see [2] and the article 'A Second Opinion on Data Flow Machines and Languages' by Gajski, Padua, Kuck and Kuhn in [1]) . We shall compare the Evon model with both the multiprocessing von Neumann model and the data flow model, and also with the VLIW architecture of Fisher. (We shall ignore the systolic array approach here because we feel that it is more towards the special purpose end of the spectrum of parallel processing.) It is not that we are necessary critical of the ambitious attempts of the two prevalent approaches or the innovation in VLIW. Rather, we simply argue that Evon is a pragmatic approach to parallel processing for general purpose applications, and a new direction worth pursuing.

Both Evon and the data flow model attempt to find parallelism at the expression level, or parallelism of fine granularity. In both cases parallelism is implicit, not explicitly programmed. The basic idea of data flow computation is to have an instruction executed as soon as its arguments are available, giving a high degree of parallelism. This does away the program counter of the von Neumann model to break the so called 'von Neumann bottleneck'. However, instructions may waste time waiting for unneeded arguments. Data values pass through data flow graph as tokens and an operation is triggered whenever all input tokens are available. Since side-effects are disallowed by using single-assignment data flow languages, any two enabled operations can be executed in either order or concurrently. For data dependent loop iteration, there is nothing to stop further iteration from proceeding, even though one is not yet completed. This causes tokens to accumulate on certain arcs of the data flow graph, hence the need to mark tokens for different rounds of computation. Complicated internal control schemes have been developed in attempts to solve this problem. The difficulty in managing input-output token queues is largely due to an abstraction at too low a level; in other words, the granularity of parallelism is too small.

In contrast, Evon explicitly employs vector operations and supports some important high-level primitives to deal with vector loops and frequently occurring data dependent loops. The overall control is centralized but computations in a basic block are distributed while the control of subtasks is diffused. In other words, by first organizing computations into some pre-specified computation chunks which absorb frequently used microprogram controls and then limiting the application of data flow principles to a basic block we greatly simplify the control structure of our computational model and make its machine realization more practical.

We can look at the multiprocessing von Neumann model as a horizontal extension of the classical von Neumann model, while Evon is a vertical extension. Strictly speaking, Evon is neither SIMD nor MIMD, but conceptually it is more like a SIMD machine. In a SIMD machine, each processing element has instruction decoding capability as instructions are broadcast to them. In Evon the functional execution elements can only execute specific operations. Centralized instruction decoding and scheduling saves hardware circuits. Even in a MIMD multiprocessing von Neumann machine, functional parallelism in terms of operating an adder and a multiplier of one processing element concurrently is almost impossible. Hence from a hardware utilization point of view, Evon is more efficient. On the other hand, the processor array configuration of multiprocessing von Neumann model has two advantages. First, as microprocessors are now widely available, aside from the complication of an interconnection network, an assemblage of processors can readily be made. Second, such a scheme is scalable, i.e. we can talk of thousands of (identical) processors connected together with potential speedup in the thousands. In Evon, if on the average we can have 1.5 of the 4 functional elements busy (for example, if floating-point adds and multiplications balance to within 50 percent), then 64 execution units in each functional elements will give us a speedup of approximately 100. Naturally, pipelining techniques can further increase the speedup rate, but this is not unique to Evon.

In a multiprocessing von Neumann model, the programmer needs to explicitly manage synchronization unless a very sophisticated compiler like Parafrase is employed to automatically transform sequential codes into parallel forms. In Evon, the programmer is freed from the concern of synchronization as this is done by the compiler during dependency calculation and partly managed by the instruction scheduling hardware.

In theory a MIMD machine like the NYU Ultracomputer, which has a very ingenious synchronization primitive, can also exploit expression level parallelism. As Gottlieb and Schwartz pointed out in 'Networks and Algorithms for Very-Large-Scale Parallel Computation' in [1 2] the Ultracomputer is fully capable of simulating a data flow machine. In reality, to explicitly program synchronization to systematically exploit parallelism at the expression level in the manner of the data flow model would be a formidable job. Also, network traffic for such fine grain parallelism on a MIMD machine is likely to be congested. Hence, a multiprocessing von Neumann MIMD machine is best suited to exploit parallelism at the procedure or task level. This requires a certain amount of job regularity, and the programmer has to decompose a programming job for parallel execution. On the other hand Evon offers ease of programming and requires less regularity within its jobs. For jobs with regularity, like physical simulation on lattice points, we can most likely turn them into vector forms to be efficiently processed by Evon. However, on those jobs an Ultracomputer-like MIMD machine can deliver much higher speedup than Evon. In a sense, Evon trades high potential parallel speedup with that of general applicability.

The VLIW architecture also requires less regularity of its jobs than do typical multiprocessing von Neumann machines. The reason the sophisticated trace-scheduling techniques has been developed for the VLIW architecture is the underlying assumption that basic blocks on a typical scientific or engineering program are quite small. We have seen that this assumption need not be true if we use a vector von Neumann language with enough high-level primitives as a source language. Indeed, many instances where the trace-scheduling techniques can successfully predict the direction of branches in the VLIW approach are the vector loops. The Evon model can apply to a much wider range of application areas for parallel processing, because when programs are written in a vector von Neumann language with sufficient high level primitives, the compiler can do a more effective job of parallelization.

6. Conclusion

We have presented the Evon model for parallel processing. It is a multifunction machine organization with global central control and local distributed computations. It executes vector instructions very efficiently and has hardware supported compound primitives to speedup some typical data dependent loops. The parallelism comes not only from parallel execution on vector instructions but also from the simultaneous execution of different hardware functional units on independent data. Parallelism is not explicitly programmed, but rather automatically extracted by a compiler from what is inherent in a high level language program. We report our research on defining an instruction set and implementation of a compiler. We believe that for many scientific, engineering as well as commercial applications, vector loop operations are most likely to dominate a large portion of computation time. If we can find an efficient and cost-effective solution to that particular form of computation, we would have made a great progress toward a practical exploitation of parallelism in computations.

We compare the Evon model with other models of parallel processing, and we see some advantages of our approach. But different models designed to exploit different levels of parallelism can be complementary to each other. We need more experimentation with existing programs as well as rewriting existing programs in languages of different style in research on parallel processing to better understand the effectiveness of various approaches.

Acknowledgements

My thanks to George Almasi for his support and encouragement of this research.

References

[1] T. Agerwala and Arvind, ed., Data Flow System, IEEE Computer, Feb., 1982.

[2] Arvind and R.A. Iannucci, A Critique of Multiprocessing von Neumann Style, Proc. 10th Annual Int'l Symposium on Computer Architecture, 426-436, 1983.

[3] T.L. Chang and F.D. Fisher, A Block-driven Data-flow Processor, Proc. Int'l Conf. on Parallel Processing, 151-155, 1982.

[4] W.M. Ching, A Portable Compiler for Parallel Machines, Proc. IEEE Int'l Conf. on Computer Design, 592-596, 1984.

[5] W.M. Ching and D. L. Ostapko, Hardware and an Algorithm for Performing Parallel Prefix Calculation, IBM Technical Disclosure Bulletin, No.Y0884-0007, May, 1984.

[6] W.M. Ching and D. L. Ostapko, Regular and Fast Hardware Interconnection for a Group of Execution Units to Calculate All Partial Results of an Associative Operation, IBM Technical Disclosure Bulletin, No.Y0884-0694, February, 1985.

[7] M. Denneau, The Yorktown Simulation Engine, Proc. ACM- IEEE 19th Design Automation Conf., 55-59, 1982.

[8] J. Fisher, Very Long Instruction Word Architectures and the ELI-512, Proc. 10th Annual Int'l Symposium on Computer Architecture, 140-150, 1983.

[9] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, Cedar-A Large Scale Multiprocessor, Proc. Int'l Conf. on Parallel Processing, 524-529, 1983.

[10] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rodulph and M. Snir, The NUY Ultracomputer-Designing a MIMD Shared Memory Parallel Computer, IEEE Trans. on Computers, vol.32, 175-189, 1983.

[11] A. Hassitt, J. Lageshulte and L. Lyon, Implementation of a High Level Language Machine, ACM Comm., 16(1973), 199-212.

[12] L Haynes, ed., Highly Parallel Computing, IEEE Computer, Jan., 1982.

[13] H.A. Hartung, FORTRAN: "for the birds", 11, Physics Today, September, 1984.

[14] R. Jaganathan and E.A. Ashcroft, Eazyflow: A Hybrid Model for Parallel Processing, Proc. Int'l Conf. on Parallel Processing, 514-523, 1984.

[15] A. Kapauan, K.Y. Wang, D. Cannon, J. Cuny and L. Snyder, The Pringle: An Experimental System for Parallel Algorithm and Software Testing, Proc. 1984 Int'l Conf. on Parallel Processing, 1-6.

[16] R.M. Keller, F.C.H. Lin and J. Tanaka, Rediflow Multiprocessing, Proc. IEEE COMPCON, 1984.

[17] E. Kronstadt and G. Pfister, Software sopport for the Yorktown Simulation Engine, Proc. ACM-IEEE 19th Design Automa- tion Conf., 60-64, 1982.

[18] D. Kuck, A debate: Retire FORTRAN? No, Physics Today, May, 1984.

[19] J.R. McGraw, A debate: Retire FORTRAN? Yes, Physics Today, May, 1984.

[20] C.V. Ramomoorthy and W.H. Leung, A Scheme for the Parallel Execution of Sequential Programs, Proc. Int'l Conf. on Parallel Processing, 312-316, 1976.

[21] H.D. Shapiro, A Comparison of Various Methods for Detection and Utilizing Parallelism in a Single Instruction Stream, Proc. Int'l Conf. on Parallel Processing, 67-70, 1977.

[22] P.C. Treleveaven and R.P. Hopkins, Decentralized Computation, Proc. 8th Int'l Symposium on Computer Architecture, 279-290, 1981.

[23] J.R. Vanaken and G.L. Zick, The Expression Processor: A Pipelined, Multiple-Processor Architecture, IEEE Trans. on Computers, vol.30, 525-536, 1981.

[24] P.S. Wang and M.T. Liu, Parallel Processing of High-level Language Programs, Proc. Int'l Conf. on Parallel Processing, 17-25, 1981.

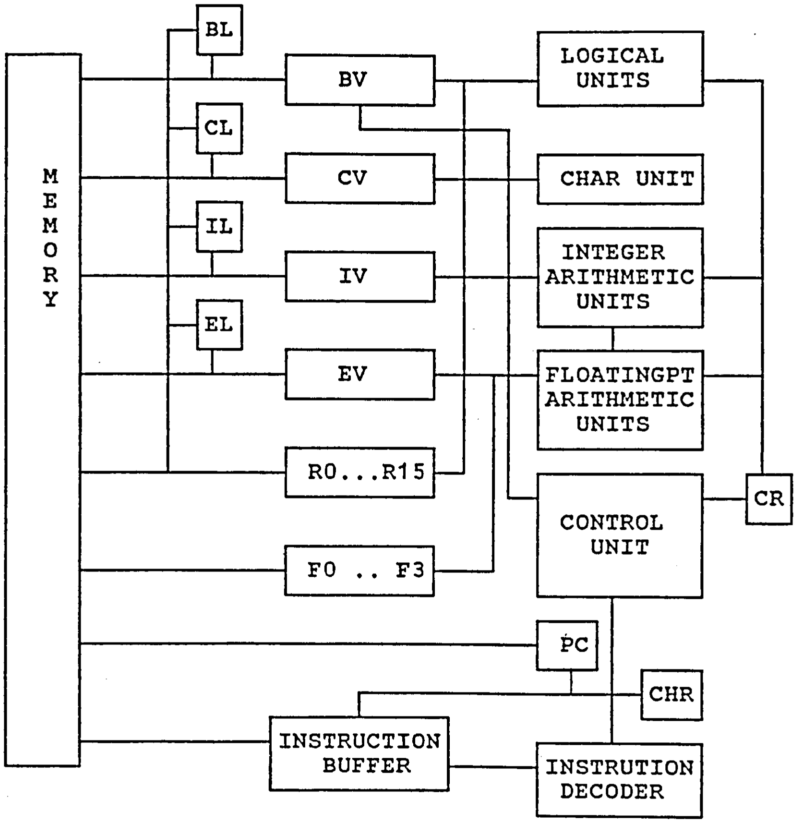


Figure 1 . Logical Structure of a Evon machine

EV: floating-point vector accumulator   CR: condition-code reg.
EL: floating-point vector length reg.   CHR: chaining register

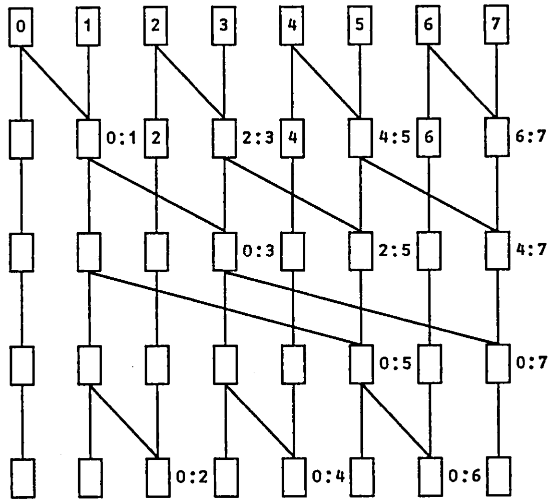


Figure 2 . Interconnection Scheme for the Execution of Scan operation (the boxes represent the operand registers and the array of execution units of a functional element).

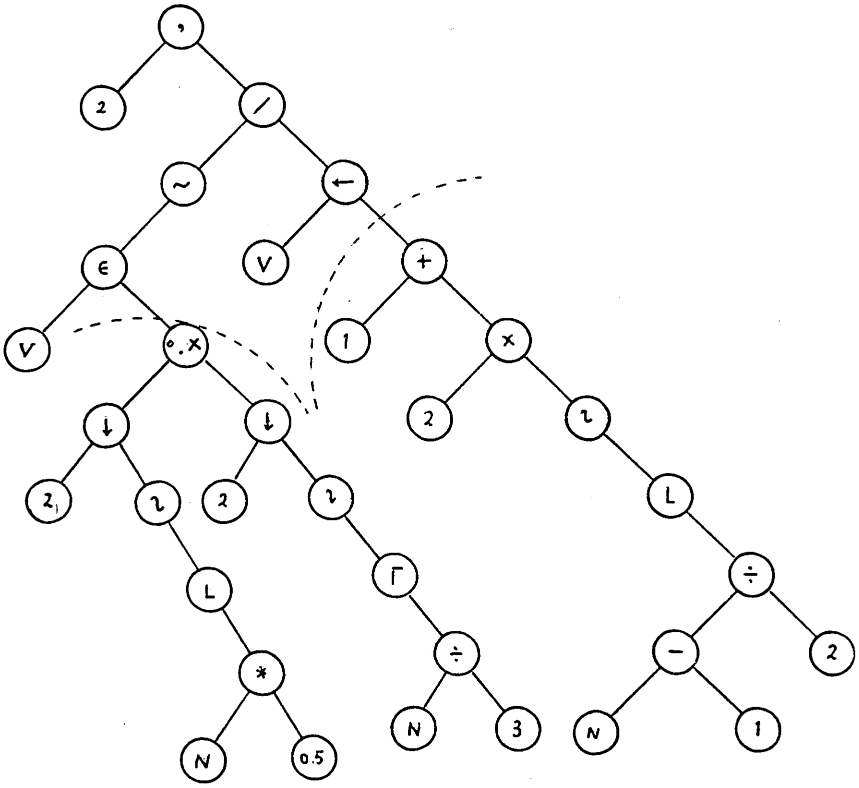


Figure 3.

Parse tree for the prime-number finding example, illustrating expression-level parallelism within a basic block. The leaf nodes are variables or constants, the internal nodes are operations. Two dashed circles indicate two intra-block data independent subtrees.

OPTIMAL CODE GENERATION FOR EXPRESSIONS ON SUPER SCALAR MACHINES

Pradip Bose

IBM T.J. Watson Research Center, H2-B48
P.O. Box 218, Yorktown Heights, NY 10598

Abstract

The problem of generating optimal code to evaluate expression trees under varied assumptions of the underlying execution model is considered. A RISC-style load/store instruction set architecture is assumed throughout. Initially, a simple non-pipelined, serial execution model, with N registers is considered. The machine model is then modified to allow pipelined execution of a single instruction stream. Finally, a decoupled access/execute (DAE) model is considered, in which each (decoupled) unit is pipelined. Concepts of optimum register allocation and code scheduling are combined into a single, efficient, tree-walk algorithm.

I. INTRODUCTION

Sethi et al [1] and Aho et al [2] consider the problem of minimizing the number of instructions and/or the number of storage references in the evaluation of an expression, given a fixed number of registers. The implicit model of the underlying execution process in such a study, is that of a strictly sequential, non-pipelined machine with infinite memory (see Figure 1). In [1], a specific architecture (instruction set) is assumed, whereas the results in [2] are applicable to a wider range of architectures; nonetheless, the underlying execution model in each case is as depicted in Figure 1. Under this assumption, code length is a valid metric for judging the optimality of a given code generation algorithm; in this sense, the algorithms presented in [1,2] do generate "optimal" code.

In this paper, we consider the task of speeding up execution of expression trees by progressively complicating the execution model without changing the basic idea of single instruction stream flow (fetch) from memory. Pipelined execution [10] is one of the principal features of modern day supercomputing. In addition, various forms of overlapped/decoupled processing combined with pipelining [3,4,6,7] have been used and/or suggested in order to enhance performance. These approaches, in effect, incorporate more and more parallel/pipelined execution modes, without diverging from the basic philosophy of "single instruction counter" processing. Our intent in this paper is to generalize the classical results of [1,2] by considering models of execution used in single instruction counter super scalar machines. In [11], we presented a brief outline of the basic structure of algorithms implemented by us, with such target machines in mind. In this paper, we present detailed algorithms, along with formal justifications of our claims and results.

Two distinct (albeit related) issues in code generation are register allocation and instruction (code) scheduling. Efficient register allocation is necessary to minimize so-called "spill-code" [12], which has the desired effect of reducing execution time, since certain unnecessary references to data memory are eliminated. Instruction scheduling as a code-generation-time optimization is also known to be beneficial [13] and has been implemented in some compilers, such as the IBM PL.8 compiler [14]. The results presented in this paper effectively illustrate that at least for straight-line tree code (i.e., no common sub-expressions, single root or result node) optimum register allocation and code scheduling can be achieved via a single tree-walk code generation algorithm.

An important parameter in generating efficient code is of course the set of commands (or instruction set) available to the compiler writer. Since our ultimate objective is to study the optimal compilation problem for decoupled access/execute (DAE) type organizations [3, 4], our assumed kernel instruction set is a basic, load/store or RISC (Reduced Instruction Set Computer) architecture, in which all arithmetic operations are performed in register-to-register (RR) style only. The only instructions which involve data transfers to and from memory, are the LOAD and STORE instructions. Thus, access/execute decoupling is reflected naturally in the instruction set architecture. Decoupling considerations aside, there are other basic justifications for RISC architectures which have been pointed out by proponents of this concept of "simple" instruction sets; -- the concept and its benefits, for both compilers and hardware implementations, are presented in [15,16].

Another important parameter in enhancement of supercomputer performance is the "instruction issue logic." By adopting more and more sophisticated issue mechanisms, out of order executions may be initiated to ease bottlenecks and hence improve performance [5,6]. Such sophistication in the instruction issue logic can be seen to ease the burden on the compiler writer, since sophisticated code scheduling algorithms are rendered almost unnecessary. Since this paper is mainly geared toward the compiler aspect of performance enhancement, sophistication in the instruction issue logic will not be considered; in other words, a simple, serial instruction fetch and issue mechansim is assumed throughout.

Note that, this paper deals only with execution of expression trees; detailed consideration of the general problem of executing loops of basic blocks (dags) is beyond the scope of this paper. Our attempt here is to describe a natural generalization of the classical results in [1], under super scalar models of execution.

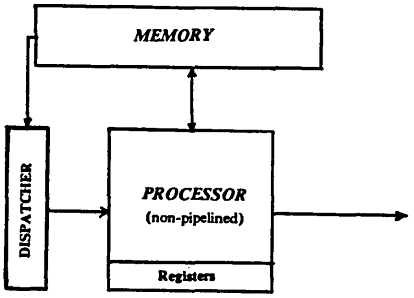


Figure 1. Machine Model A.

II. INITIAL ASSUMPTIONS

We state below, the initial set of assumptions regarding the structure of expressions, and the basic architecture and organization of our machines.

Expressions

As in [1], we assume that there are no nontrivial relations between operators and elements. For example, we assume that all data elements are distinct, and that laws such as $a*b + a*c = a*(b + c)$ are not applicable. All operations are binary; all operations are taken to be noncommutative. Thus, effectively, no restructuring of the expression tree is allowed.

Architecture

The commands permitted are of the following kinds:

1. C(mem) --> C(reg); i.e., LOAD reg, mem-addr.
2. C(reg) --> C(mem); i.e., STORE reg, mem-addr.
3. OP[C(reg1),C(reg2)] --> C(reg3); i.e., OP reg3, reg1, reg2.

Note that the exact address computation mechanism is left unspecified; it is not of concern to us in this paper. We assume a fixed number of cycles needed to decode, generate effective address and send request to memory, and receive data back from memory. Implicit in this is the possible use of a separate set of (index) registers for address computation. We are not concerned here with the number or availability of such registers: for all practical purposes we assume unlimited resources (and hence no conflicts) in that regard. We are concerned here with doing the required arithmetic computation, as fast as possible, given a fixed number of (arithmetic) registers, N.

Memory and Instruction Fetch/Dispatch

We effectively assume unlimited (infinite) storage for both instructions and data. We assume that instruction prefetch mechanisms are available, so that instruction(s) are ready for dispatch on demand. (Note, once again, that we are not dealing with branches, merely straight-line code resulting from expressions). When decoupled organizations are considered, we assume split (infinite) I-cache and D-cache, so that conflicts resulting from trying to access memory at the same time, are absent.

Hardware, Organization and Timing

With pipelined execution, a LOAD or STORE (access) instruction is assumed to take A (≥ 1) cycles (stages) to complete. Each arithmetic (execute) instruction is assumed to take E (≥ 1) cycles (stages) to complete.

The register file is assumed to have two i/o ports; a data "read" (for a computation) and a data "write" (from the putaway bus) can be initiated in the same cycle. (Where the same register is involved in the read and write, a read-before-write interlock mechanism is assumed). The register load/store mechanism uses a separate bus. Attempting to write into the same register from the two buses at the same cycle is not possible in any of our schemes; i.e., the situation does not arise.

No buffers are assumed in front of the decode unit(s); instruction dispatch is just assumed to block (to a given unit) once an instruction is blocked in the decode stage due to an interlock.

III. CODE GENERATION ALGORITHMS

In this section, we present specific algorithms to generate time-optimal code (under the assumptions made) for various models of execution. First, we state the modified tree labeling algorithm [1] needed for our load/store architecture.

The Tree Labeling Algorithm

Each node of the expression tree is labeled with a number that turns out to be the minimum number of registers needed to evaluate the node without stores. To each node n is assigned a label L(n), from the bottom up.

1. If n is a leaf then $L(n) = 1$.

2. If n has descendants with labels l_1 and l_2, then for $l_1 \neq l_2$, $L(n) = \max(l_1, l_2)$, and for $l_1 = l_2$, $L(n) = l_1 + 1$.

The first model we consider is that of a serial, non-pipelined machine: this is just for the sake of completeness and eventual comparison with the pipelined machine models.

A. Serial, Non-Pipelined, Single Dispatch Execution

Figure 1 shows the assumed model of computation, where the dispatcher (I-unit) issues a single instruction to the E-unit as soon as the E-unit finsishes a given computation fully. In this case, of course, minimizing the total number of LOADs and STOREs results in a time-optimal code sequence. It is easily seen that Algorithm 1 of [1], suitably modified by the constraint of our load/store architecture, can be used to generate the optimal code in this case. We call this Algorithm A.

ALGORITHM A

After generating the expression tree and labeling the nodes:
(1) Apply (2) to the root with registers $R_1, R_2, ..., R_N$ available. Routine (2) will evaluate the expression represented by the subtree extending from the node to which it is applied. The result will appear in the lowest numbered register available.

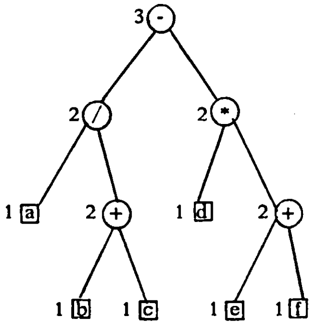
(2) Let n be a node, $L(n) = l$. Suppose, $R_m, R_{m+1}, ..., R_N$ are available, $1 \leq m \leq N$.

Case 1: $l = 1$.
(A) n must be a leaf. Enter (LOAD) the value of n, which is an input datum, into R_m.

Case 2: $l > 1$; let the descendants of n have labels l_1 and l_2.
(B) If $l_1, l_2 \geq N$, apply (2) to the right descendant with registers R_m, $R_{m+1}, ..., R_N$ available. STORE the value of the right descendant into a temporary memory location T. Apply (2) to the left descendant with registers $R_m, R_{m+1}, ..., R_N$ available, leaving the result in R_m. LOAD the value of the right descendant from T into R_{m+1}. Evaluate n using the values in R_m and R_{m+1} leaving the result in R_m.
(C) If $l_1 = l_2$, and at least one of l_1, l_2 is less than N, apply (2) to the descendant with the higher label with registers $R_m, R_{m+1}, ..., R_N$ available. Leave the result in R_m. Apply (2) to the other descendant, with registers, $R_{m+1}, R_{m+2}, ..., R_N$ available. The result will appear in R_{m+1}. Evaluate n using the values in R_m and R_{m+1} and leave the result in R_m.
(D) If $l_1 = l_2 < N$, treat the right descendant as the node with the higher label and proceed as above.

Example 1, taken from [1]:

Consider the expression a/(b+c) - d*(e+f), whose tree is:



The integers at each node are the labels assigned by the labeling algorithm. Algorithm A produces the following code for N = 2:

1. LOAD R_1, addr(f)
2. LOAD R_2, addr(e)
3. ADD R_1, R_2, R_1
4. LOAD R_2, addr(d)
5. MPY R_1, R_2, R_1
6. STORE R_1, addr(T)
7. LOAD R_1, addr(c)
8. LOAD R_2, addr(b)
9. ADD R_1, R_2, R_1
10. LOAD R_2, addr(a)
11. DVD R_1, R_2, R_1
12. LOAD R_2, addr(T)
13. SUB R_1, R_1, R_2

The proof that Algorithm A produces optimal code for the execution model under consideration and the stated assumptions, is along the lines stated in [1] and is being omitted in this paper; a couple of relevant lemmas and a theorem are stated here for completeness.

Lemma 1: Algorithm A evaluates a tree with no STORES when at least as many registers as the label of the root are available.

■

Lemma 2: The label of a node is a lower bound on the minimum number of registers required to evaluate the node without any stores. (Lemma 1 is a special case of Lemma 2, where the node being considered is the root node).

■

Theorem 1: Algorithm A produces an optimal code sequence for the stated machine model and assumptions.

■

B. Serial, Pipelined, Single Dispatch Execution

Figure 2 shows the new machine model under consideration. The dispatcher dispatches a single instruction to the decode unit on a given cycle, if it is free to do so, i.e., if the decoder is not blocked. Assumptions regarding hardware, organization and timing are as stated in section II.

Interlock Mechanism

A hardware interlock mechanism is assumed to ensure that (1) a LOAD is not decoded until the previous data in the register has been used; (2) an arithmetic instruction is not decoded until its source registers have been written with the right data and until the previous data in the target register has been used (or stored); (3) a STORE is not decoded before data in the register is ready. Essentially, we assume the existence of two 'interlock bits' or flags (READY and BUSY) associated with each register. The bits are all OFF initially. The BUSY bit of register R_i is turned ON either when a LOAD R_i or STORE R_i instruction is initiated (i.e., is dispatched to the decode unit), or when an arithmetic instruction in which R_i is an operand, is initiated. When an arithmetic instruction completes, all associated interlock bits of the source registers are turned OFF; the target register READY bit is turned ON and its BUSY bit is turned OFF. When a STORE completes, the BUSY bit of the STOREd register is turned OFF. The STORE is decoded if and only if the register to be STOREd is READY and NOT BUSY. When a LOAD completes, the BUSY bit of the LOADed register is turned OFF and its READY bit is turned ON. The LOAD is decoded if and only if the register to be LOADed is NOT BUSY. An arithmetic instruction is allowed to decode if and only if the two source registers are READY and the target register is NOT BUSY. It is not necessary to assume that the dispatcher has access to the interlock information per se; rather, it may be assumed that the dispatcher blocks only if the decode unit of the processor blocks, in a given cycle.

Under these assumptions of hardware and architectural support, the code sequence generated for Example 1 by Algorithm A (N = 2) can be shown to execute in 30 cycles on Model B (with A = 3, E = 2). By contrast, for the serial, nonpipelined case, the time to execute would have been 34 cycles, assuming the timings to be identical, i.e. a LOAD or STORE takes 3 cycles and an arithmetic instruction takes 2 cycles to execute.

If the number of registers, N is just 2 then it can be shown that we cannot decrease total execution time in this case, under the simple issue and decode mechanisms adopted. However, if N ≥ 3, (i.e. greater than or equal to the label of the root), then one can obtain optimal performance, by using the following algorithm.

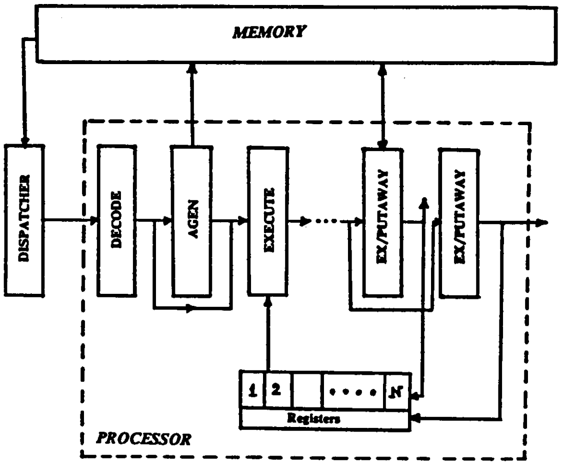


Figure 2. Machine Model B.

ALGORITHM B

Like Algorithm A, Algorithm B walks through the expression tree in emitting code; however, unlike Algorithm A, the number of visits per node is not a constant number. Also, in general, the overall tree traversal pattern can be quite complicated, depending on the tree structure and the pipeline parameters A and E. Nevertheless, the worst case complexity of the algorithm is still linear in the number of tree nodes (see section IV). For simplicity, we state this algorithm for the case: $N \geq L(\text{root})$, where N = number of registers and the labels L are generated by the basic tree labeling algorithm, as before. In other words, this algorithm generates (time) optimal code (without stores) for the machine model described (Model B, Figure 2).

Node attributes:

For a given node n of the tree, its ancestor node is denoted as father(n), its left descendant node as left-son(n) and its right descendant node as right-son(n). All non-leaf nodes of the tree are called op-nodes. Associated with each tree node is a set of attributes, which in a sense defines the "state" of the node at any given point durirg code generation. Following is the set of node attributes:
(a) n: integer; $1 \leq n \leq$ number of nodes; (* node number *)
(b) node-type: (leaf, op); (* node descriptor *)
(c) L: integer; $L \geq 1$; (* static label *)
(d) T_1, T_2: integers; $T_1 \geq 0$, $T_2 \geq 0$; (* dynamic attributes *)
(e) OP: ('ADD', 'SUB', 'MPY', 'DVD'); (* operation identifier *)
(f) ID: ('a', 'b', ..., 'z'); (* operand identifier *)
(g) l-enabled, r-enabled, ls-enabled, rs-enabled, l-visited, r-visited: boolean; (* dynamic attributes *)
(h) l-reg, r-reg: integer; (* source operand reg numbers *)
(i) t-reg: integer; (* target reg number *)

In addition, each node is understood to have upto three defined (static) pointer attributes: uptr, lptr and rptr, linking it to its ancestor, left descendant and right descendant respectively.

The static labels (L) are obtained by applying the basic tree labeling algorithm as before; in principle, these could be attached to tree nodes during actual parsing (i.e., tree-building). Similarly, the other static attributes [(a), (e), (f), along with "uptr", "lptr" and "rptr" (see above)], are all fixed during tree initialization. The attributes T_1 and T_2 may be looked upon as "time-to-fire" labels; i.e., $T_1(n)$ specifies the additional number of time units (after the last initiated instruction) one must wait before the left operand is available in a register; $T_2(n)$ specifies the same thing, but for the right operand. Thus, $T(n) = \max(T_1(n), T_2(n))$ specifies the waiting period (in machine cycles) for this op-node (instruction) to be decoded, counting from the last initiated instruction. Initially, these values are all set to ∞, for each node n.

An op-node is said to be "enabled" if LOAD/OP instructions for both its son nodes have already been generated. Initially, therefore, all the op-nodes are disabled. An op-node is said to be r- (l-) enabled if the LOAD/OP instruction for its right (left) son node has been generated. The boolean attributes l-enabled and r-enabled are used to record the above conditions. The attributes ls-enabled and rs-enabled are defined as follows: ls-enabled(n) = l-enabled (left-son(n)) and r-enabled (left-son(n)); rs-enabled (n) = l-enabled (right-son(n)) and r-enabled (right-son(n)). The attribute l-visited (r-visited) is set to "true" if the left (right) son of this node has been visited most recently. Initially, both flags are false; both these flags cannot be "true" at the same time.

Global data structures:

The registers (numbered 1 through N) are initially organized as a "free queue", called f-queue. This can be imagined to be a simple FIFO queue, for example. Each entry of this queue has two fields: (a) the actual register number, i (ϵ [1, 2, ..., N]); and, (b) a dynamic attribute t(i), associated with the register i. The t value may be looked upon as a "time-to-free" attribute, which specifies the additional number of (machine) cycles, after the last initiated instruction, one must wait before an instruction using that register as an operand can be allowed to decode. Initially, the "t field" of each entry in the free queue is set to 0. Any time a register is needed to emit a LOAD instruction, it is taken from the head of the queue; when a register is freed (after a STORE or arithmetic instruction), it is put back at the tail of the queue. The variable "f-reg" always refers to the register at the head of the free queue; f-reg = 0 signifies that the free queue is empty. Initially, f-reg = 1 and size (f-queue) = N.

Also, an "enabled queue", called e-queue, is maintained: this is a FIFO queue of pointers to enabled op-nodes at any given point of the algorithm. Initially, this queue is empty, of course. During the (algorithmic) step that a node becomes enabled, it is entered at the tail of e-queue in case it is not scheduled immediately. During the step in which an op-node (arithmetic instruction) is scheduled (emitted) by the algorithm, the corresponding entry at the head of e-queue is removed. Each valid entry of e-queue consists of two fields: (a) the (static) node number (n) of the enabled node; and (b) T(n), a dynamic attribute, signifying the "wait period", in machine cycles, (counting from the cycle of last initiation), before this node can be decoded. When a node n is first entered into e-queue, the value of T(n) is set to $\max(T_1(n), T_2(n))$. The variable "e-ptr", when defined, is the node pointer at the head of e-queue; i.e., the node pointed to by e-ptr, called "e-ptr@", is always the enabled node with the minimum T value; (e-ptr = nil <==> e-queue is empty).

In addition, a "partially enabled queue", called pe-queue is maintained. This is similar to e-queue, except that it carries information about partially enabled nodes, i.e., those which are l-enabled or r-enabled, but not enabled. Once again, nodes are entered into this queue, only if necessary, (i.e., if m becomes 0), in the order in which they become (partially) enabled. The timing field T of each entry in this case is $T_1(n)$, if the node n under consideration is l-enabled, and $T_2(n)$ if n is r-enabled. At a given step, a partially enabled node is entered into pe-queue if and only if: (a) its disabled son is a leaf node, and (b) the number of available registers at that step is 0. An entry is removed from the head of pe-queue when the next LOAD has to be scheduled. The variable "pe-ptr", when defined, is the node pointer at the head of pe-queue; i.e.,'the node pointed to by pe-ptr, called pe-ptr@, is the current candidate op-node whose disabled leaf node must be LOADed.

The queues f-queue, e-queue, pe-queue, the associated variables f-reg, e-ptr, pe-ptr and the dynamic time attributes T, T_1, T_2, and t are basically managed and manipulated (updated) by the code emission routines "gen-load" and "gen-op". These routines, it should be noted, make use of the pipeline parameters A and E (see section II) in setting and updating the above attributes (see below). The variables enabled-node-count and leaf-node-count keep track of the number of enabled nodes and the number of remaining leaf nodes of the tree respectively. The variable m records the current number of available (free) registers, where $0 \leq m \leq N$. The condition (e-ptr = nil) is identical to the condition (enabled-node-count = 0). (Note that once an op-node is enabled, its son nodes may conceptually be deleted from the tree, since they will not be visited by the algorithm again).

Informal Description of Algorithm B:

The expression tree is first initialized, with the node attributes set to proper initial values. The L labels are generated by using the basic tree labeling algorithm as in Algorithm A. The global data structures are also initialized as indicated during their definition. Tree traversal is initiated with the root node. The initial traversal pattern down the tree from the root is almost exactly along the lines of the corresponding traversal in Algorithm A; the purpose, at this stage, is essentially: selection of a start leaf node for emission of the first LOAD instruction. During subsequent traversals up and down the tree, code is generated by taking into account several factors: (a) availability of "free" registers (for LOADs); (b) state of the node visited: "enabled" or "not enabled"; (c) the scheduling strategy (based on maintained timing information) which will ensure emission of optimal code. The normal traversal pattern down the tree is: father, least recently visited son (subtree), other son (subtree), father. If neither son has been visited earlier, the one with the larger label (L) is visited first, as in Algorithm A. Similarly, the normal traversal pattern up the tree is: current son, father, other son, father.

A visit to node n may be classified as casual or busy according to the complexity of the tasks involved. A casual visit results in routine checking and book-keeping operations, without actual code emission. The book-keeping actions are mostly local node operations; for instance, the attributes l-visited, r-visited, ls-enabled or rs-enabled of the father may be set. A partially enabled or enabled node may be entered into the corresponding queue. The checking actions are local as well as global; for instance, the current state (enabled, l-enabled, r-enabled, etc.), of the node is checked; also, global values like m (number of free registers), enabled-node-count, leaf-node-count, etc., may have to be checked. A busy node visit results in some routine checking operations, as well as actual code emission by invocation of the routines "gen-load" and "gen-op". Substantial local and global attribute modification actions are associated with such invocations (as described shortly). Thus, during a given visit to a node n, the corresponding code emission task may be accomplished by invoking gen-load or gen-op (busy visit), or, if the conditions checked are not right the next node in the traversal path is visited, postponing code emission for this node to a subsequent visit. The normal traversal path is altered, at a given step of the algorithm, only if the current node visit is a casual visit, and, a specific condition is detected; for example, m = 0. In such special cases, the next node to be visited is eptr@ or pe-ptr@. (See the actual algorithm for details). A busy node visit (code emission) always results in continuation of a normal traversal pattern; i.e., the next node visited, in this case, is always the father. Invoking "gen-load" or "gen-op" requires updating of one or more of the queues f-queue, e-queue and pe-queue. "Updating", includes addition/deletion of entries as well as decrementing T and t values by proper amounts (see below).

The routines: "gen-load" and "gen-op"

These are the basic routines invoked by the algorithm for generating LOADs and OPs. By construction, the algorithm visits a leaf node if and only if the number of available registers, m, at that point is non-zero and the corresponding LOAD can be scheduled. Thus gen-load is invoked if and only if the visited node n is a leaf node. In addition to actual emission of the LOAD instruction, the following actions are associated with an invocation of gen-load from (leaf) node n:
(a) The entry at the head of f-queue is removed; the corresponding register f-reg is used for the target, t-reg, of the LOAD; f-reg is updated to indicate the new entry at the head of f-queue; the values of leaf-node-count and m are decremented by 1.
(b) The following attributes of father(n) are set/modified as applicable: l-enabled (or r-enabled), T_1 (or T_2). Thus, for example, if n is the left son of its father, then: T_1(father(n)) is set to the integral value A, where A is the pipeline parameter for LOADS (see section II); also, l-enabled (father(n)) is set to true.
(c) The t values of registers in f-queue, and the (T_1, T_2, T) values of father(n), together with those of all nodes in e-queue and pe-queue are decremented by the amount k = 1 + t(t-reg), where the necessary t value was obtained from the head entry of f-queue, during target register (t-reg) selection. (This decrementation of time attributes corresponds to advancing the pipeline execution by the minimum number (k) of cycles before the next instruction can be decoded).

The next node visited after the invocation: "gen-load(n)" is always father(n).

The routine gen-op generates the required OP with l-reg and r-reg as source registers, and either l-reg or r-reg as the target (depending on context) as far as this particular algorithm is concerned. In addition to emission of the required OP instruction (ADD, SUB, MPY or DVD), the following book-keeping operations are associated with the invocation of gen-op from node n:
(a) The freed register(s) are added to the tail of f-queue; the associated t value(s) are copied from the T value of this op-node. The value of m is incremented.
(b) Node attributes of the father are updated in the same manner as before, except that now the pipeline parameter E is used.
(c) The time attributes of the global data structures f-queue, e-queue, pe-queue, as well as those of father(n) are decremented by k = 1 + T((father(n))).

The next node visited after the invocation: gen-op(n) is always father(n).

We now present Algorithm B using a high-level (Pascal-like) notation. The detailed specification of routine book-keeping operations for casual node visits is omitted for clarity; also, the expansions of gen-load and gen-op (with their associated book-keeping actions) are not shown explicitly. (The informal description above should be referred to for completeness).

The Actual Algorithm

After the tree and the data structures are initialized as described,
(1) Apply (2) to the root with all N registers available (free).
(2) Let n be a node, with $L(n) \geq 1$ and $T(n) \geq 0$; suppose m registers are available, $0 \leq m \leq N$.

```
CASE 1. L(n) = 1, hence, m > 0 (n is a leaf node).
begin
    gen-load(n);
    apply (2) to father(n);
end; (* CASE 1 *)

CASE 2. L(n) > 1 (n is an op-node).
begin
    update-local-attributes(n);
    if enabled(n) then begin
        if (n = root) or (T(n) = 0) or (n = e-ptr@) then begin
            gen-op(n);
            if (n ≠ root) then apply (2) to father(n);
            if (n = root) then return;
        end
        else begin
            enter-into-e-queue(n);
            if (m > 0) and (pe-ptr ≠ nil) and (min (T_1(pe-ptr@),
            T_2(pe-ptr@)) ≤ T(e-ptr@)) then apply (2) to pe-ptr@
            else apply (2) to e-ptr@;
        end
    end
```

```
else if l-enabled(n) or r-enabled(n) then begin
    if (m > 0) then begin
        if r-enabled(n) and r-visited(n) and (not ls-enabled(n))
        then apply (2) to left-son(n)
        else if l-enabled(n) and l-visited(n) and (not
        rs-enabled(n)) then apply (2) to right-son(n)
        else apply (2) to father(n);
    end
    else begin
        if (l-enabled(n) and (node-type(right-son(n)) = leaf)) or
        (r-enabled(n) and (node-type(left-son(n)) = leaf)) then
        begin
            enter-into-pe-queue(n);
            apply (2) to e-ptr@;
        end
        else apply (2) to e-ptr@;
    end
end
else (* n is disabled *) begin
    if (m > 0) and (leaf-node-count > 0) and (pe-ptr = nil) then
    begin
        if rs-enabled(n) then apply (2) to left-son(n)
        else if ls-enabled(n) then apply (2) to right-son(n)
        else begin
            if (L(right-son(n)) ≥ L(left-son(n))) and (not
            r-visited(n)) then apply (2) to right-son(n)
            else if (not l-visited(n)) then apply (2) to left-son(n);
        end
    end
    else if (m > 0) and (pe-ptr ≠ nil) and (min ($T_1$(pe-ptr@),
    $T_2$(pe-ptr@)) ≤ T(e-ptr@))then apply (2) to pe-ptr@
    else apply (2) to e-ptr@;
end;
end; (* CASE 2 *)
```

The proof of the following theorem is straightforward; it follows from construction of the algorithm, as discussed earlier. The formal proof is omitted in this paper.

Theorem 2: Algorithm B produces an optimal code sequence (without stores) for the stated machine model and assumptions, with $N \geq L(\text{root})$.

■

For N = 4, the followg code generated by Algorithm B (with A = 3, E = 2), executes (on model B) in 16 cycles (optimal).

```
1. LOAD R1, addr(f)
2. LOAD R2, addr(e)
3. LOAD R3, addr(d)
4. LOAD R4, addr(c)
5. ADD  R1, R2, R1
6. LOAD R2, addr(b)
7. MPY  R1, R3, R1
8. LOAD R3, addr(a)
9. ADD  R2, R2, R4
10. DVD R2, R3, R2
11. SUB R1, R2, R1
```

Figure 3 shows the timing chart for the above example.

Note that the code generated by Algorithm A (with N = 4, A = 3, E = 2) would require 24 cycles (far from optimal) on Model B, even though the same number of instructions (11) would be emitted! Also, note that code emitted by Algorithm B is always optimal for Model A as well, in that a minimum number of instructions is generated. An additional note of interest is that for A = 1, E = 1, the code produced by Algorithm B would be identical to that generated by Algorithm A.

The generalization of Algorithm B to the case where N may be smaller than the L label of the root is easy. In this case STOREs have to be scheduled appropriately for optimal performance. Timing charactersitics of LOADs and STOREs are assumed identical in our implemented program, CODEGEN (see section IV).

Cycle #	DECODE	AGEN/REQ	LOAD/STORE	EXECUTE & PUTAWAY
1	①			
2	②	①		
3	③	②	①	
4	④	③	②	
5	⑤	④	③	
6	⑥		④	⑤
7	⑥			
8	⑦	⑥		
9	⑧		⑥	⑦
10	⑧			
11	⑨	⑧		
12	⑩		⑧	⑨
13	⑩			
14	⑪			⑩
15	⑪			
16				⑪

Figure 3. Timing Chart (Alg. B, Model B).

C. Serial, Multi-Dispatch, Pipelined, Decoupled Execution

Figure 4 shows the excution model, under consideration, which has been formed from the notions expressed in [3,4]. While retaining the single instruction flow from memory, this organization allows upto two instructions to be dispatched every cycle to the two separate functional units: one for processing LOADs/STOREs (access processor) and the other for performing actual computation (execute processor). We assume the same hardware interlock mechanism for registers as before, to ensure correct usage and storage of data. Our optimization problem, as before, is to generate code which would produce the required results of the computation in the fastest possible time.

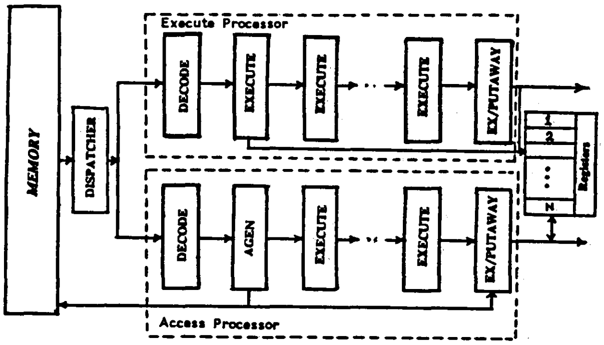


Figure 4. Machine Model C.

Algorithm C, for optimal code generation under this assumed model, is almost identical to Algorithm B. The main difference is in the management of some of the global data structures. The manner in which some of the dynamic variables are manipulated (updated) is different in this case, because of the provision of multiple instruction dispatch. The basic methodology and complexity of the algorithm remains unchanged. We omit the detailed description of Algorithm C in this paper. For N = 4, the code generated by Algorithm C (with A = 3, E= 2) executes (on model C) in 15 cycles (optimal).

In our implemented program CODEGEN (see section IV), we have coded Algorithms B and C, together with modified versions of C, in which buffers have been added before the decode units and also other organization/architecture changes have been experimented with. We leave the detailed description of these models and algorithms for presentation in a later, detailed version of this paper.

IV. ALGORITHM COMPLEXITY

In this section, we present an informal discussion of the complexity of our algorithms. We indicate, without formal proofs, that Algorithms A, B and C are all linear in the number of tree nodes, n. In each case, we shall estimate complexity of code generation after the tree has been constructed and initialized (i.e., static attributes and labels have been evaluated as applicable).

Algorithm A: In this case, as discussed in [1], the number of visits per node is constant, nan.ely 2: the first time each node is visited is when the tree is traversed from the root down to the leaves to determine the order of evaluation; the second visit occurs during traversal up the tree from the leaves producing the evaluation sequence. Hence, for this simple tree-walk algorithm, the total number of node visits is 2n; i.e., the number of steps required by the algorithm is linearly proportional to the number of tree nodes, n.

Algorithm B: In this case, the number of visits per node is not a constant number; however, there is a pre-determined upper bound on the number of visits per node (irrespective of the tree), as seen from the following lemmas.

Lemma 3: In Algorithm B, each leaf node is visited exactly once.

■

Lemma 4: In Algorithm B, the number of visits to an op-node, NV^o, is bounded as follows:

$3 \leq NV^o \leq 4$

■

The proofs of the above lemmas follow easily from the methodology expressed in the statement of our algorithm. A leaf node is visited if and only if the corresponding LOAD is to be scheduled; hence, the number of visits per leaf node is always 1. Under normal traversal pattern, each op-node is visited at least thrice. Upto 4 visits are needed in case a node is entered into e-queue or pe-queue and visited later to complete code generation.

Lemma 5: The total number of node visits (NV), i.e., the total number of invocations of part (2) of Algorithm B, is bounded as follows:

$NL + 3^*NO \leq NV \leq NL + 4^*NO$

where, NL is the number of leaf nodes and NO is the number of op-nodes. The above inequality can obviously be re-written as:

$(n + 2^*NO) \leq NV \leq (n + 3^*NO)$,

where n is the number of tree nodes (since n = NL + NO).

■

Lemma 5 clearly follows from Lemmas 3 and 4.

Thus, we see that in the worst case, the total number of node visits is linearly proportional to the number of nodes. Every visit to a node does not result in the same amount of work for the code generator (algorithm), however. Let us try to see how the total cost (TC) is split up among the tasks performed by the algorithm.

Let,

C_1 = cost per casual node visit (no code emission).

C_2 = cost per busy node visit (code emission).

Then,

$TC = (NV - n)^*C_1 + n^*C_2$.

If C_1 and C_2 are fixed (constant) costs per casual and busy node visit, respectively, then C must clearly be linear in the number of nodes, n, since NV is proportional to n. However, depending on the exact implementation of the data structures (e.g., fixed length arrays versus variable length arrays), and the manner in which they are manipulated (updated), C_2 may vary from visit to visit. For a given node, numbered i, $C_2(i) = C_{21} + C_{22}(i)$; where, C_{21} is the cost (invariant with i) for actual code emission and other routine tasks, and $C_{22}(i)$ is the cost for updating the global data structures. $C_{22}(i)$, at worst, is proportional to the lengths of f-queue, e-queue and pe-queue at the point of the algorithm where code for node i is emitted. The length of f-queue, at worst is proportional to the number of registers N, which is a constant, independent of n. The maximum length to which pe-queue can grow (for expression trees) can be shown to be 1; and, the maximum length to which e-queue can grow can be shown to be (A - E) where A and E are the pipeline parameters. It follows from the above discussion, that the total cost TC is linear in the number of tree nodes, n.

The basic methodology used in Algorithm C is the same as that in Algorithm B; it can easily be shown that in this case too, the worst case complexity is linear in the number of tree nodes, although, the number of actual node visits is slightly higher.

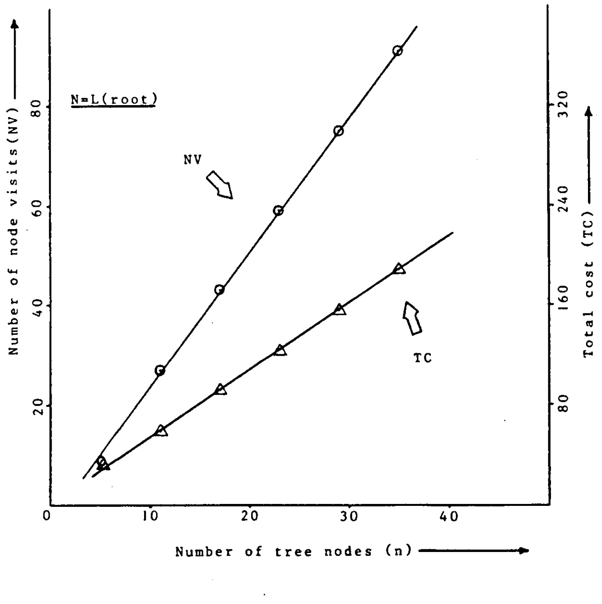


Figure 5. Performance of Algorithm B.

Family of expressions: (for Fig. 5)

n=5: a/(b+c).
n=11: a/(b+c)-(d*(e+f)).
n=17: a/(b+c)-(d*(e+f))+(g/(h-j)).
n=23: a/(b+c)-(d*(e+f))+(g/(h-j))-(k*(l+m)).
n=29: a/(b+c)-(d*(e+f))+(g/(h-j))-(k*(l+m))+(n/(p-q)).
n=35: a/(b+c)-(d*(e+f))+(g/(h-j))-(k*(l+m))+(n/(p-q))-(r*(s+t)).

Implemented Program

As mentioned previously, a program called CODEGEN has been implemented successfully, in order to experiment with tne various algorithms for code generation referred to in this paper. The fact that Algorithm B (or C) is linear can be easily proved, as discussed above. We present actual experimental results (Figure 5) for the family of arithmetic expressions shown. The total cost plotted is based on realistic (relative) cost units, and is based on actual numbers and sizes of tasks involved; thus, the TC plot is truly representative of the actual execution time characteristics of our implemented algorithm(s). Experimentation with a wide class of expression structures has exhibited the same general characteristics.

CODEGEN has been written in Pascal and runs on an IBM 370 mainframe. It includes an expression parser, tree builder and labeler, a tree printer along with the actual code generation procedures.

V. CONCLUSION

Code generation algorithms for more generalized models of execution than those considered in the classic papers of Sethi et al [1] and Aho et al [2], have been sketched. With the growing importance of high performance (super) computer architectures, such algorithms, which fully exploit the parallelism available, are deemed to be of considerable significance. We have presented outlines and performance results of basic algorithms for simplified models. Our implemented program CODEGEN is actually capable of handling more complicated models and is parametrized to handle various architecture/organization changes. In particular, methods for designing the instruction set suitably for enhanced performance, have been experimented with using techniques described in [9]; systematically derived stack [8] and other generalized instruction set architectures have been considered in conjunction with the various execution models. We are currently experimenting with an implemented program capable of handling basic blocks (expression dags), for our pipelined machine models.

REFERENCES

[1] R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," JACM, Vol. 17, No. 4, October 1970, pp. 715-728.

[2] A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," JACM, Vol. 23, No.3, July 1976, pp. 488-501.

[3] J. E. Smith, "Decoupled Access/Execute Computer Architectures," Proc. 9th. Ann. Symp. on Computer Architecture, pp. 112-119, April 1982.

[4] T. K. Agerwala, "How Fast Can a Single Instruction Counter Machine Execute?", several invited talks delivered: Stanford University (May 1983), University of Illinois (July 1983), Massachusetts Institute of Technology (April 1984).

[5] S. Weiss and J. E. Smith, "Instruction Issue Logic for Pipelined Supercomputers," Proc. 11th Ann. Symp. on Computer Architecture, pp. 110-118, June, 1984.

[6] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of Research and Development, Vol. 11, Jan. 1967.

[7] R. M. Russell, "The CRAY-1 Computer System," Comm. ACM, Vol. 21, No. 1, January 1978, pp. 63-72.

[8] J. L. Bruno and T. Lassagre, "The Generation of Optimal Code for Stack Machines," JACM, Vol. 22, No. 3, pp. 382-397.

[9] P. Bose and E. S. Davidson, "Design of Instruction Set Architectures for Support of High-Level Languages," Proc. 11th Ann. Symp. on Computer Architecture, June 1984, pp. 198-206.

[10] P. M. Kogge, "The Architecture of Pipelined Computers," McGraw-Hill, 1981.

[11] P. Bose, "Optimal Code Generation Algorithms for Arithmetic Expressions Executing on Pipelined, Decoupled Architectures," Proc. International Conference on Computer Design (ICCD), October 1986.

[12] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," SIGPLAN NOTICES, Vol. 17, No. 6, June 1982, pp. 98-105.

[13] F. E. Allen and J. Cocke, "A Catalogue of Optimizing Transformations," in Design and Optimization of Compilers, R. Rustin (Ed.), Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 1-30.

[14] M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler," SIGPLAN NOTICES, Vol. 17, No. 6, June 1982, pp 22-31.

[15] G. Radin, "The 801 Minicomputer," Proc. Symp. on Architectural Support for Programming Languages and Operating Systems, Palo Alto, March 1982, pp. 39-47.

[16] D. A. Patterson, "Reduced Instruction Set Computers," Comm. ACM, Vol. 28, No. 1, January 1985, pp. 8-21.

A SYMMETRIC CONCURRENT B-TREE ALGORITHM

Vladimir Lanin and **Dennis Shasha**

Courant Institute of Mathematical Sciences,
New York University, New York, New York 10012
lanin@nyu-csd2.arpa, shasha@nyu.arpa

ABSTRACT: We present a method for concurrent B-tree manipulation in which insertions are performed as in an earlier paper by Lehman and Yao, and deletions are done in a symmetrical, novel fashion. The result is an algorithm in which each process holds locks on at most one node at a time, except in rare cases. To allow this low level of synchronization, the integrity constraints on the data structure are re-examined. This is useful in verifying the algorithm by a general semantic serializability proof method. Simulation shows that the algorithm is capable of achieving significantly better concurrency than other algorithms that perform insertions and deletions symmetrically.

1. INTRODUCTION

B-trees are a popular implementation of the *dictionary*, an abstract data type (ADT) that supports the actions *search*, *insert*, and *delete*. A full review of B-trees may be found in [Comer 79], but let us say here that a sequential algorithm for performing the above actions in a B+ tree usually runs in three stages: a descent through the tree to a leaf node, an operation on the leaf node that either checks for the existence of a key, adds a key, or removes a key (for search, insert, and delete, respectively), and an optional ascent during which the tree is restructured in order to rebalance it. It is also possible to restructure the tree during the descent, as described in [Guibas, Sedgewick 78]. Restructuring is done by *splitting* a single node into two neighboring ones and by *merging* neighboring nodes into one. The execution of an entire action takes time logarithmic in the number of keys stored in the structure.

It is useful to allow several different, asynchronous processes to access the B-tree concurrently. This is easy when all the processes are searches, but the goal is to also allow concurrent insertion and deletion, while ensuring correctness and allowing as much concurrency as possible.

1.1. Previous Work

The first concurrent B-tree algorithms ([Samadi 76], [Bayer, Schkolnik 77]) ensure correctness by isolating each action from the effects of all other actions active in the structure concurrently: a concurrent action never encounters a situation that it could not encounter if executing alone. Two techniques are used to achieve this isolation. We present them with their rationale:
a) An action descending from a parent node to one of its children must acquire a lock on the child before releasing the lock on the parent (lock-coupling). Without this lock on the parent, a writer action could acquire locks on both nodes, change them (e.g. split the child), and release the locks (Fig. 1). This could cause the descending action to perform incorrectly, as the search(d) in fig. 1 does.
b) A writer action (which in [Samadi 76] and [Bayer, Schkolnik 77] restructures during the ascent) must lock out other writer actions from the entire subtree dominated by the highest node it has to modify. This averts another danger: suppose writer w needs to ascend to and modify node n, and writer w' is already active below n. Since w' might also have had to modify n, either w or w' would update n first. The other would then access a different version of n than during its descent, potentially causing errors.

These techniques guarantee correctness at the expense of concurrency. Locking processes out of entire subtrees is a severe disadvantage in itself. Furthermore, the most straightforward method of achieving the subtree-locked state (which is used in the above algorithms) is for writer processes to begin placing exclusive (or at least writer-exclusion) locks from the top of the tree. This temporarily locks out new actions entirely. The simultaneous use of lock-coupling by other processes aggravates the problem by increasing the interference.

Another algorithm in [Bayer, Schkolnik 77] mitigates the above effects of subtree-locking by having writers first perform an optimistic search-like descent that uses read locks on all nodes but the final leaf, on which it uses an exclusive lock. A writer can complete after the optimistic descent if it does not have to split or merge the leaf. This has a probability of about $\frac{k-1}{k}$ if the number of keys stored in each leaf ranges from k to $2k$. Otherwise, the writer must give up and re-descend using the normal protocol.

The algorithm in [Mond, Raz 85] is a concurrent version of the [Guibas, Sedgewick 78] algorithm, and avoids subtree-locking by doing restructuring during the descent. However, it still

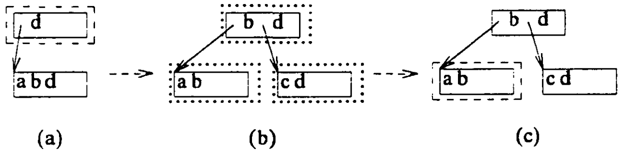


Figure 1. A sequence of events leading to error
in the absence of lock coupling

Descending action *search*(d) releases its lock (shown in dashes) on the parent node in (a) before acquiring lock on child node in (c). Meanwhile, action *insert*(c) splits the child node in (b), making the key d unavailable to the search.

This work was partially supported by the National Science Foundation under grant DCR8501611 and by the Office of Naval Research under grant N00014-85-K-0046

CH2345-7/86/0000/0380$01.00 © 1986 IEEE

employs lock-coupling and requires using exclusive locks starting at the root. We note that the [Mond, Raz 85] algorithm can also be enhanced by optimistic descents.

Simulation shows (as presented in this paper, as well as in [Ellis 80], [Shasha 84]) that in a high update environment, algorithms that use exclusive locks high up in the tree in the initial descent can not achieve significant concurrency. Although optimistic descents help considerably (especially in the [Mond, Raz 85] version), they depend on the number of keys in a leaf being large. They seem to employ more locking than absolutely necessary.

1.2. The Lehman and Yao Algorithm and Deletions

A more radical approach to achieving a minimal amount of locking is not to try to isolate actions from each other (by lock-coupling and subtree-locking), but to enable them to recover from the effects of other actions. [Ellis 80] and [Lehman, Yao 81] present such algorithms. The latter avoids both lock-coupling and subtree-locking and is simple; we base our approach on it.

The Lehman and Yao algorithm introduces the *B-link* structure, obtained from a B+ tree by connecting each level into a singly linked list. A pointer called a *rightlink* points from every node to its right neighbor, so actions can stray to the left of the direct path to a desired node, and still recover by following the rightlinks. For example, a split occurs in two stages (see Fig. 2): first, half the data is moved out of *n* into *n'*, and *n'* is inserted into the linked list; next, a pointer to *n'* is put into the parent of *n*. Between the two stages, descending processes can be allowed to make the transition from the parent of *n* to *n*, because even if they need data that was moved to *n'*, they can still reach it. Thus, lock-coupling is unnecessary. Subtree locking is unnecessary because ascending writers can follow rightlinks to the appropriate place to make their updates.

Although it is not apparent in [Lehman, Yao 81] itself, the B-link structure allows inserts and searches to lock only one node at a time. This is fully utilized in algorithms in [Sagiv 85] and [Shasha 84].

The major problem with the [Lehman, Yao 81] algorithm is that no provision is made for the merging of nodes. Thus, the structure never becomes smaller even after many deletions, although Lehman and Yao propose to rebalance it off-line. This deficiency has been addressed in the algorithms of [Salzberg 85] and [Sagiv 85], which give procedures for restructuring the entire tree that can run concurrently with the other actions. However, it is not clear how often such procedures should be run.

No algorithm has previously been given for performing a deletion analogously to an insertion, by using merges where the insertion uses splits. Independently, [Sagiv 86] has a nearly symmetric deletion algorithm similar to ours. Our merge is different and should result in fewer locks and/or accesses, but we expect the performance of our two algorithms to be close. We learned of the algorithm too late to include it in the simulation.

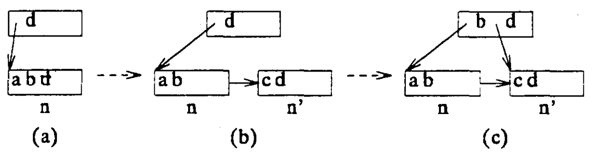


Figure 2. A split in the B-link structure.

Processes looking for *d* in *n* in (b) or (c) can proceed to *n'* and still find *d*.

1.3. Symmetric Deletion Approach

Let us consider various implementations of merging a node *n* with its right neighbor *n'* in the B-link structure. If we move the data in *n'* to *n* (Fig. 3a), there is no path that processes can follow from *n'* to the data moved out. This may, for example, lead a process to conclude that *c* is not in the structure.[1] If we move the data in *n* to *n'* (Fig. 3b), we will need to access the left neighbor of *n* to remove *n* from the linked list. Finding the left neighbor requires either much time or a doubly linked list, which increases complexity and the locking overhead (see [Shasha 84] for example).

Our novel way of doing merges solves the above problems. We move the data from *n'* to *n*, but also direct a pointer called an *outlink* from *n'* to *n* (Fig. 4). Any action needing data that was in *n'*, and still expecting to find it there, can access *n'*, see that it is empty, and follow the outlink to *n*.

The outlink is actually another application of the *link technique,* other examples of which include the original Lehman and Yao rightlinks, as well as elements in the algorithms of [Kung, Lehman 80] and [Ellis 83]. In general, the link technique calls for processes that change sections of a data structure in a way that would cause other processes to fail to provide a link to another section of the data structure where recovery is possible.

2. OUR ALGORITHM

The following section fleshes out the short description of our algorithm given above. Please refer to Fig. 5 for an example of a B-link structure and to the appendix for a pseudo-Pascal program implementing the algorithm.

2.1. Some Definitions

Each internal node consists of a rightlink, and a sequence

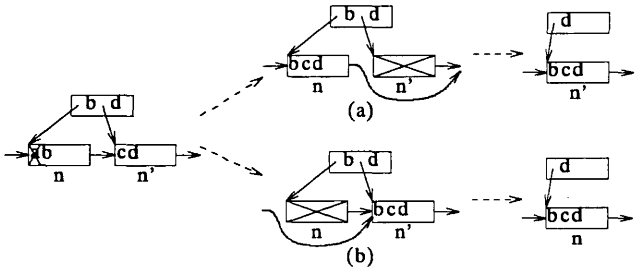


Figure 3. Two possibilities in merging *n* and *n'*.

Unfortunately, (a) is incorrect and (b) is inconvenient.

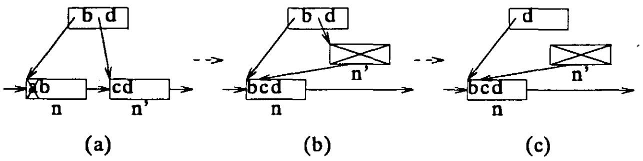


Figure 4. Merging in the B-link structure.

Processes looking for *d* in *n'* in (b) and (c) can proceed to *n* and still find *d*.

[1] This illustrates a general principle that data should only move to the right in the singly-linked B-link structure.

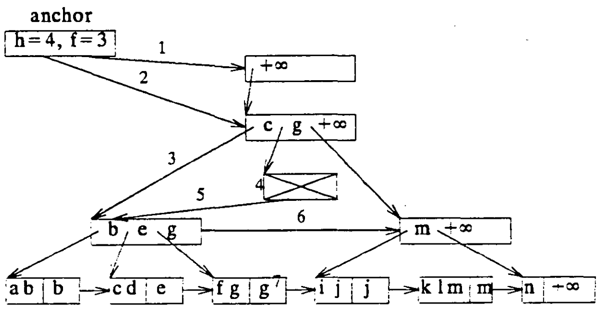


Figure 5. A possible legal state of a B-link structure.

1) The *top* pointer. 2) The *fast* pointer. 3) A downlink. 4) An empty node. 5) An outlink. 6) A rightlink. 7) A leaf's only separator.

$(p_1,s_1,p_2,s_2, \cdots ,p_q,s_q)$, where each p_i is called a *downlink* and s_i a *separator*. A downlink is a pointer to a (usually null) chain of empty nodes terminating in a non-empty node on the level below. A downlink is said to *trans-point* to this non-empty node. A separator is a value from the same domain as keys, but serves only as a navigation guide and not as a key entry. For $1 \le i < q$, $s_i < s_{i-1}$. Each leaf consists of a rightlink, a sequence of keys $(v_i, v_2, \cdots ,v_q)$, and a single separator s, where $v_i < v_{i-1} \le s$ for $1 \le i < q$.[2]

Let us define:

rightsep(d), where d is a downlink: the separator on the immediate right of d. Note that it is always in the same node as d.

rightsep(n), where n is a non-empty node: the rightmost (largest) separator in n.

leftsep(d), where d is a downlink in node n: the separator to the immediate left of d. If d is the leftmost downlink in n, $leftsep(d)$ is the rightmost separator in the left neighbor of n. If n is the leftmost node on its level, $leftsep(d)$ is considered to be $-\infty$.

leftsep(n), where n is a non-empty node and d is the leftmost downlink in it: $leftsep(n) = leftsep(d)$. If n is an empty node whose outlink points to n', let us define $leftsep(n) = leftsep(n')$.
coverset(n), where n is a non-empty node: $\{x \mid leftsep(n) < x \le rightsep(n)\}$.

2.2. Locks

The locking model used in [Lehman, Yao 81] assumed that an entire node could be read or written in one indivisible operation. This assumption enabled that algorithm to let reader processes access nodes without first locking them in any way. Since it is not always reasonable to assume the availability of atomic node reads or writes (such as when the structure is in primary memory), and in order to make comparisons to other algorithms easier, we use a more general locking scheme similar to the one in [Bayer, Schkolnik 77]. It is a simple matter to convert our algorithm to fit the [Lehman, Yao 81] model, as should be done when atomic node accesses are provided by the hardware.

Two kinds of locks are used: a read lock and a write lock. Many processes may hold read locks on a node at the same time, but a write lock demands exclusive access to a node, allowing no other locks. A process must hold at least a read lock to perform a non-modifying operation on a node; a process must hold a write lock to modify a node. Deadlock is not possible in our algorithm, so lock managers of different nodes may be co-independent.

Each action holds no more than one read lock at a time during its descent, an insertion holds no more than one write lock at a time during its ascent, and a deletion needs no more than two write locks at a time during its ascent.

2.3. The Locate Phase and the Decisive Operation

The first phase of all three actions (search(v), insert(v), and delete(v)) is called the *locate phase*, as its function is to locate and place a lock on a leaf n such that $v \in coverset(n)$. That is, n should be the node where v should be if it is anywhere in the structure. The locate phases of all three actions are identical, except that the last two must leave a write lock on n, not a read lock.

The locate phase does not use lock-coupling. On the path to the appropriate leaf, the locate phase locks each node, determines which link to follow out, and unlocks the node. It does not, however, unlock the final leaf. Each node-to-node transition usually follows a downlink, but follows a rightlink or outlink if necessary. Write locks are used on the leaf level if the calling action is insert or delete, and read locks elsewhere.

Once the locate phase completes, an action performs its *decisive operation*, which consists of checking for v (for search), adding v (for insert), or removing v (for delete) in the locked leaf.

2.4. Restructuring

Adding or removing information in a node may leave it too crowded or too sparse. As in sequential B-tree algorithms, this is rectified by *normalizing* the node: i.e. splitting it into two nodes or merging it with its neighbor.[3] Since a split or a merge involves inserting or removing the pointer from the parent node to the node being created or vacated, the normalization of a node may make further splits and merges higher in the structure necessary. Thus the tree is restructured by an ascent from the leaf accessed by the decisive operation.

2.4.1. Two-Phase Splits and Merges. In order to avoid having to lock the parent (and possibly more distant ancestors) of a node while normalizing it, our algorithm performs splits and merges in two stages, as shown in figures 2 and 4. The first stage (fig. 2b and 4b), which does not involve the parent of the node being split or merged, is called the *half-split* or the *half-merge*, respectively. After a half-split, n and n' may still be regarded as one logical node. The half-merge may be regarded as logically merging n and n', and re-directing to n the downlink to n'. After the completion of a half-split or a half-merge, all locks are released.

The second stage (Fig. 2c and 4c) is called the *add-link* or *remove-link*. This operation takes place on an "appropriate" node on the level above the node n which has just been half-split or half-merged. In a sequential computation the appropriate node is always the (unique) parent of the node involved. This can not work in our algorithm because a node may (rarely) have several or no parents. The purpose of a downlink — to channel to its target the locates of keys in the target's *coverset* — suggests a better criterion. Let s be the rightmost separator in the left node after a half-split, or the rightmost separator in the left node before a half-merge. The (unique) appropriate "parent" node p is the one for which $s \in coverset(p)$.

An add-link consists of inserting s and the downlink to the

[2] For the sake of simplicity, we have restricted the structure to holding only unique keys.

[3] Merging two nodes into one may result in a node that is too crowded. Thus a merge may be followed by a split; this is our version of rotation.

new node into the sequence in p, with s on the immediate left of the downlink. A remove-link consists of removing s and the downlink to the empty node from p, with s on the immediate left of the downlink[4].

Normally, finding the node with the right *coverset* for the add-link or remove-link is done as in [Lehman, Yao 81], by reaccessing the last node visited during the locate phase on the level above the current node. Sometimes (e.g. when the locate phase had started at or below the current level) this is not possible, and the node must be found by a new descent from the top.

2.4.2. Changing the Height of the Structure. If an ascent can not proceed because the current level is the top level, a new top level is created (by the *grow* operation). Safely removing unneeded top levels turns out to be too difficult. However, descending through them for every action is unacceptable. Thus, instead of actually removing them, we simply avoid searching through them. A continuously running process called the *critic* keeps track of (a good guess at) the highest level containing more than one downlink, called the *fast* level. The critic uses only read locks in getting this information. Pointers to the leftmost nodes of both the fast level and the top level are kept in the *anchor*, whose address is known to all processes. If need be, the critic modifies the fast pointer. The locate phase of all actions begins at the leftmost node on the fast level. (The locate phase will complete at the correct leaf regardless of the precise height of the fast level because it is in fact safe to start a locate at any level.) The levels above the fast level remain unused until they are needed again when the fast level contains too many (e.g. more than 2) non-empty nodes. Since these levels normally consist of only one node and number no more than the maximum height of the structure over time, they take up little space.

2.5. Freeing Empty Nodes

Once all pointers to an empty node are removed, the node becomes a candidate for re-use[5]. However, it can not then be freed immediately since processes may exist that obtained some pointer to it before the pointer was removed. If such a process were to access the node once it was re-inserted into the structure, the process might be misdirected or damage the node.

Two approaches are possible. One, introduced in [Kung, Lehman 80] and called the *drain technique*, is to delay freeing the empty node until the termination of all processes whose locate phase began when pointers to the node still existed. The other approach is to store information in every node (its height and *leftsep*) such that a process can recognize that it has become lost and recover by going back to the previous node it accessed or to the anchor. The appendix does not contain code for either approach, but freeing empty nodes by the drain technique is transparent to the rest of the algorithm and can be added on as a separate module.

3. CORRECTNESS

We verify the algorithm by applying the ideas of semantically-based concurrency control to the dictionary abstract data type ([Ford, Calhoun 84], [Goodman, Shasha 85]). [Kung, Papadimitriou 79] established the theory of a semantic approach to concurrency control by showing the relationship between the information available to a concurrency control algorithm and the achievable concurrency. Our algorithm, like Lehman and Yao's, is not serializable in the usual syntactic sense (i.e. in the unrestricted reads and writes model), thus the more general semantic approach is necessary.

The *state* of the dictionary abstract data type (ADT) is a set of keys. The set of all potential keys is called *KeySpace* and we assume it contains the values $+\infty$ and $-\infty$ which are, respectively, larger and smaller than all others. The dictionary ADT supports the dictionary actions *search*(x), *insert*(x), and *delete*(x), which map a dictionary state s to a dictionary state s' and return value v. For search(x), $s'=s$ and $v=(x \in s)$; for insert(x), $s'=s \cup \{x\}$ and $v=(x \notin s)$; for delete(x), $s'=s-\{x\}$ and $v=(x \in s)$.[6] A sequence of dictionary actions $a_1,a_2 \cdots ,a_k$ maps an initial dictionary state s_0 to a final state s_k and a sequence of return values $v_1,v_2, \cdots ,v_k$, where each a_i maps s_{i-1} to s_i and returns v_i. It is the job of our algorithm to implement dictionary actions correctly when they execute concurrently.

A search structure state (a set of nodes and edges with associated information) implements a dictionary state. For each node n in the search structure, *contents*(n) is the set of keys in the node. The dictionary state implemented by a set of nodes N is $\bigcup_{n \in N} contents(n)$. Operations on the search structure state are called *search structure operations*. For our algorithm, these are operations like half-split, add-link, and check-key. Search structure operations map a search structure state to another search structure state and a return value. Each dictionary action is implemented as a program of search structure operations.

Let a set of dictionary actions A be run concurrently.[7] The resulting computation C consists of an initial search structure state s, a final search structure state s', the set of dictionary actions A, the set of executed search structure operations O, a function *parent* from the members of O to A, a function r from the members of $A \cup O$ to their return values, and a partial order $<$ on the members of $A \cup O$ (where $o<o'$ means o completes before o' begins, and $\neg(o<o' \vee o'<o)$ means o and o' were at least partially concurrent). We assume that for dictionary actions a and a', $a<a'$ only if for every operation o of a and operation o' of a' $o<o'$, i.e. a terminated before a' began.

Definition: A computation C with s, s', A, r and $<$ defined as above is *serializable* if the members of A can be arranged in a sequence A' such that A' maps the dictionary state represented by s to the dictionary state represented by s', produces the same return vales for the members of A as does r, and extends the partial ordering imposed on A by $<$ (i.e. if $a<a'$, then a precedes a' in A'). An algorithm is serializable if all computations it produces are serializable.

In our algorithm, locking ensures that only non-modifying operations are allowed to access the same node concurrently. This implies that the final state and return values of every computation are as if the operations occurred in an interleaved sequence that extends the partial ordering $<$ ([Goodman, Shasha 85]). This makes it possible to think of each operation as occurring alone and starting and finishing in a well-defined search structure state, which facilitates reasoning about computations.

We now develop several results needed to show that our algorithm is serializable.

[4] There are special cases in performing the remove-link. If s is the rightmost separator in p, then the downlink is leftmost in the right neighbor of p. The simplest solution is to merge p and its neighbor prior to the remove-link and to normalize afterwards. In rare circumstances, s and the downlink will not be in p at all, in which case we reissue the remove-link later. We know that it will eventually succeed because the empty node was originally created by a half-split, and all half-splits are eventually followed by the appropriate add-links. The remove-link is this add-link's counterpart and was delayed because the add-link was slow in completing. Similarly, an add-link has to be tried again if s is already present in p; this happens only when this older s's remove-link has been slow.

[5] A counter of the number of outlinks pointing to the empty node must be used. Furthermore, just the absence of a downlink to the node is insufficient, as the add-link may simply be very slow and still active. The correct condition is the completion of the remove-link.

Definition: A B-link structure state is *legal* if

LS1) There exist values h and f, $1 \leq f \leq h$, such that the non-empty nodes form h linked lists through the rightlinks, where all leaf nodes are on list 1, all downlinks from list i trans-point to nodes on list $i-1$, and the anchor contains pointers to the source nodes of lists h and f, as well as the values h and f.

LS2) The separators on each list form a strictly ascending sequence terminating in $+\infty$. The keys in each leaf are in the leaf's *coverset*.

LS3) If a downlink d points to node n, $\mathit{leftsep}(d) \geq \mathit{leftsep}(n)$. (This is the minimal condition sufficient for safe descents. If $\mathit{leftsep}(n)$ were greater than $\mathit{leftsep}(d)$, actions could follow d to n while the data they seek lies to the left of n.)

B-link Proposition: For all operations mapping legal search structure state s to state s', for all nodes n present in s, if $\mathit{leftsep}(n)=l$ in s and $\mathit{leftsep}(n)=l'$ in s', then $l' \leq l$.

Proof: This follows directly from the definitions of the operations, as the reader can easily verify. For example, an add-link can not violate the proposition because it never adds the largest separator to a node, thus not changing any *leftsep*s. As another example, a half-merge decreases the *leftsep* of the node it empties (by the definition of *leftsep* for empty nodes). This proposition is a formal re-statement of the rule that information never moves to the left in the B-link structure. □

Lemma 1: If d is one of the operations check-key(x,n), add-key(x,n), or remove-key(x,n) in computation C of our algorithm (i.e. d is a decisive operation), and C starts in a legal state and all operations up to d finish in a legal state, then n is a leaf and $x \in \mathit{coverset}(n)$ in the search structure state preceding d.

Proof: It is the locate phase that finds and locks the node on which the decisive operation acts. We note that the height of a node is never changed by any operation that maintains a legal state. The locate phase starts out at the leftmost node n at some height f. Since f is read from the anchor and the heights of all nodes remain the same, the locate phase can correctly calculate the height of any node it accesses. Furthermore, since it starts at the leftmost node, $\mathit{leftsep}(n) = -\infty < x$. The next node to be visited, m, is chosen such that $\mathit{leftsep}(m)$ is as large as possible without violating $x > \mathit{leftsep}(m)$. (LS1, LS2 and LS3 guarantee that this decision can be safely based on the values of the separators in n.) By the B-link Proposition, $\mathit{leftsep}(m)$ can only get smaller with time. Thus, when a lock is placed on m, $x > \mathit{leftsep}(m)$ still. Since m now becomes the new value of n, $x > \mathit{leftsep}(n)$ remains invariant throughout the locate phase. The locate phase terminates only when the leaf level is reached and $x \leq \mathit{rightsep}(n)$, which, together with the invariant, implies $x \in \mathit{coverset}(n)$[8]. □

Lemma 2: If o is one of the operations add-link(s,c,p) or remove-link(s,c,p) in a computation C of our algorithm, and C starts in a legal state and all operations up to o finish in a legal state, then a pointer to c trans-points to a node on the level below p and $s \in \mathit{coverset}(p)$ in the search structure state preceding o.

Proof: The argument that this condition is met (by the locate-internal subroutine) is similar to the proof of Lemma 1. The condition that c is at height one below p is met because node height does not change and is calculated correctly. As for the *coverset* condition, it is satisfied as it was for the decisive operations, except that locate-internal does not always start at a leftmost node. However, when it doesn't, the *leftsep* of the starting node is known to have been smaller than s at some previous time. By the B-link Proposition, the *leftsep* can only have gotten smaller since that time and the invariant condition is met. □

Lemma 3: If o is an operation in a computation C of our algorithm, and C starts in a legal state and all operations prior to o finish in a legal state, then o will finish in a legal state.

Proof: Clearly, no operations other than those in Lemmas 1 and 2 are capable of mapping a legal state to an illegal one. By the lemmas, these remaining operations must satisfy their respective height and *coverset* conditions, which implies that they too can not violate LS1 or LS2.

It remains to consider LS3. The only operations that could conceivably change the *leftsep* of any existing downlink are add-link and remove-link, which add and remove separators as well as downlinks. However, since the separator added or removed is the one on the left of the downlink added or removed, these operations do not change the *leftsep* of any existing downlink. By the B-link Proposition, the *leftsep* of any existing node can only get smaller. Thus, LS3 can only conceivably be violated in the introduction of a new node or downlink. Newly created nodes do not have downlinks pointing to them, so LS3 is not violated. Thus, a problem can only arise if for some add-link(s,c,p), $s < \mathit{leftsep}(c)$. However, we note that an add-link(s,c,p) is done only after a half-split(c',c) which results in $s = \mathit{rightsep}(c') = \mathit{leftsep}(c)$. By the B-link Proposition, $\mathit{leftsep}(c)$ can only get smaller between the half-split and the add-link, thus LS3 is never violated. □

Theorem: All computations produced by our algorithm that begin in a legal state are serializable.

Proof: All keys reside in non-empty leaves. By LS1 and LS2, the *coverset*s of the non-empty leaves in a legal B-link structure partition *KeySpace*. By LS2, the keys in each leaf are in that leaf's *coverset*. Thus, for every key and every legal B-link structure there is exactly one node (i.e. the leaf into whose *coverset* the key falls) that could possibly contain the key.

By induction on Lemma 3 we have that the algorithm maintains a legal state throughout a computation if it begins in one. Then Lemma 1 shows that for every decisive operation on key x in node n, $x \in \mathit{coverset}(n)$ and n is a leaf, which implies that $x \notin \bigcup_{m \in N-\{n\}} \mathit{contents}(m)$ in the state preceding the operation.

By inspection of each action, only the decisive operation is capable of changing the set of keys in the structure and only the decisive operation affects the return value of the action. Furthermore, the definition of each decisive operation is identical to the definition of its parent action, except that it acts on a single node instead of the entire structure. Since we know that this node is the only one that contains any information pertinent to the action by Lemma 1, each decisive operation correctly performs all of the work of its parent action. Thus the sequence of actions given by the sequence of their decisive operations would produce the same final set of keys and the same return values as the actual computation. □

It should be pointed out that although the algorithm is susceptible to livelock, it is free from deadlock on lock resources because of a well-ordering on the acquisition of locks. Whenever a process holds two locks simultaneously, the two nodes are always adjacent at the same height, and the lock on the node on the left is always obtained first.

[6] For simplicity, we omit a formal description of the handling of records usually associated with keys.

[7] The following definitions are actually applicable to two-level computations on any abstract data type. Simply replace the words "dictionary" with "top level", and "search structure" with "bottom level".

[8] Note that there is no guarantee that the locate phase will terminate. For example, a slow search could be indefinitely delayed by continuous half-splits on a node it needs to access. Thus, the algorithm is susceptible to livelock.

4. PERFORMANCE SIMULATION

In order to compare the actual performance of the B-link algorithm to that of previous algorithms supporting insertion and deletion, we have run the algorithms under various conditions in a simulated concurrent environment provided by the language Concurrent Euclid [Holt 83].

4.1. Model of Concurrency

We compare speedup, i.e. the ratio of the time it takes one processor to do a given amount of work to the time it takes *n* processors to do that same amount of work. Ideally, the speedup achieved by *n* processors is *n*, but actual concurrent algorithms achieve lower speedups due to interference through waiting for locks.

In our simulation, the processors receive a common list of actions constituting the work to be done. Each processor removes an action from the list, executes it, gets another, etc. When the last action completes, the simulated time is noted.

The only activity which takes time in the simulation is reading and writing nodes. (An action reads a node after obtaining a lock on it, and possibly writes it out before releasing the lock.) However, a process may be delayed while waiting for a lock held by another process which itself is reading or writing a node.

A node access (a read or a write) always takes exactly one time unit. If *n* processors issue *n* node accesses at the same time, all *n* accesses complete at the same time, after one time unit. Thus, node accesses are simulated to occur completely in parallel. Studies ([Schultz, Mukkamala 85], [Ford et al 85]) have shown that when several physical processors contend for a single relatively slow storage device capable of handling only one request at a time, the storage device can become a bottleneck and prevent the realization of most of the potential concurrency. Thus, our model of complete non-interference at the physical storage access level is most applicable to systems where data is distributed over many storage modules (such as disk drives) :capable of operating in parallel. Our results are actually quite close to those obtained in [Schultz, Mukkamala 85] for the Lehman and Yao algorithm in a simulated system containing as many storage devices as processors.

4.2. Parameters of the Simulation

For each simulation run, the values of the keys in the initial B-tree and of the arguments of the job actions were randomly and uniformly distributed over *KeySpace*. The values picked for insertions and deletions were guaranteed not to be redundant.

The parameters that could be varied in comparing the algorithms were the number of keys in the initial B-tree, the order of the tree (i.e. the out-degree of nodes), and the proportion of searches, insertions, and deletions in the actions. Since searches never interfere with each other and in the absence of updates achieve perfect speedup, including them in the simulation only masks the amount of interference produced by updates, the source of the problem. Thus, we excluded searches from the actions performed and simulated a "mix" where half the actions were insertions and half deletions.

Besides the symmetric B-link algorithm, we simulated four other algorithms. Two of these, algorithm 1 of [Bayer, Schkolnik 77] and the [Mond, Raz 85] algorithm, use lock-coupling (and, for the former, subtree-locking) with exclusive locks from the top of the tree. Algorithm 2 of [Bayer, Schkolnik 77] strives to reduce subtree-locking and exclusive locking high up in the tree by having updates first perform an optimistic search-like descent using read-lock lock-coupling. We also simulated our own version of the [Mond, Raz 85] algorithm which uses an optimistic descent like algorithm 2 of [Bayer, Schkolnik 77] and re-descends by the [Mond, Raz 85] method.

We did not simulate algorithm 3 of [Bayer, Schkolnik 77] (as well as an analogous version of the [Mond, Raz 85] algorithm) because it differs from algorithm 1 only in the partial substitution of exclusive locks by writer-exclusion (or alpha) locks. Although this substitution can improve performance by allowing read-locks to be held on alpha-locked nodes, only searches use read-locks in this algorithm (alpha-locking and optimistic descents can not co-exist because of the possibility of deadlock). Since our simulation excludes searches, its results would not be changed by the substitution.

4.3. Results of the Simulation

Figure 6 shows the speedups achieved by the various algorithms in an initial B-tree containing 800 keys with each node having between 7 and 13 children or keys. Under these conditions (and all other update-intensive conditions we have tried), the algorithms that place exclusive locks from the root during the initial descent achieve a maximal speedup of only about two. Optimistic descents improve this result significantly, with our version of [Mond, Raz 85] achieving a maximal speedup of around 10 before ceasing to get better performance with larger

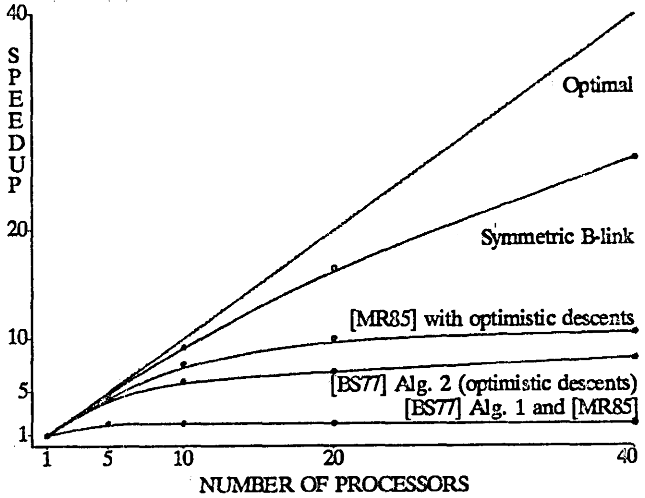


Figure 6.
Simulation of insertions and deletions in a 7-13 B-tree containing 800 keys.

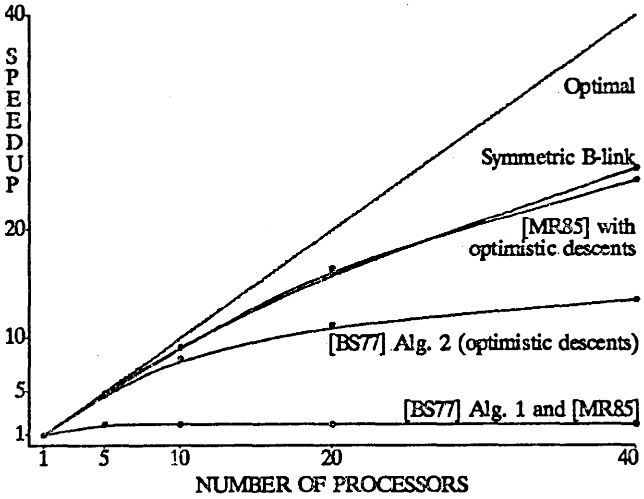


Figure 7.
Simulation of insertions and deletions in a 20-39 B-tree containing 10,000 keys.

numbers of processors. The somewhat worse performance of the second [Bayer, Schkolnik 77] algorithm must be attributed to the use of subtree-locking by the latter. The B-link algorithm, however, gets a speedup of 26.5 with 40 processors and would probably get a better speedup with a larger number of processors (compiler constraints prevented simulation of more than 40).

Varying the parameters to the simulation brings some significant changes to the results. When the algorithms were run on a 7-13 tree with 100 keys, the small size of the tree (only two levels) increased the interference and thus decreased the performance of all algorithms. The B-link algorithm was still best, but achieved a maximal speedup of only 7. This figure increased to 14 with an initial tree size of 200 keys.

More significantly, increasing the size of leaves greatly improves the performance of the algorithms employing optimistic descent. This is because splits and merges become less frequent, increasing the probability that the optimistic descent will succeed. In a 20-39 tree with 10000 keys (roughly three levels), our version of the [Mond, Raz 85] algorithm achieves only slightly lower speedups than the B-link algorithm, as shown in Fig. 7. The second [Bayer, Schkolnik 77] algorithm lags somewhat behind these two, probably because of its subtree-locking.

5. CONCLUSION

We have presented a deadlock-free concurrent B-tree algorithm supporting search, insert, and delete which employs less locking than any other algorithm that treats deletions symmetrically with insertions. To verify the algorithm, we use a correctness criterion based on the semantics of the dictionary ADT. This criterion permits executions that would not be considered correct in the model of uninterpreted reads and writes. Simulation has shown that the algorithm is capable of achieving much better concurrency than the earliest (and still most widely used) concurrent B-tree algorithms that employ much exclusive locking high in the tree. It performs at least as well as other algorithms which also try to avoid using exclusive locks, and performs significantly better than these in trees containing a small number of keys (e.g. under 20) in each leaf.

REFERENCES

[Bayer, Schkolnik 77] BAYER, R., and SCHKOLNIK, M. Concurrency of operations on B-trees. Acta Inf. 9,1 (1977) 1-21.

[Casanova 81] CASANOVA, M. A. *The concurrency problem for database systems, Lecture Notes in Computer Science, vol 116, Springer-Verlag, 1981*

[Comer 79] COMER, D. The ubiquitous B-tree. ACM Computing Surveys, 11,2 (June 1979) 121-137

[Ellis 80] ELLIS, C. Concurrent search and insertion in 2-3 Trees. Acta Inf. 14,1 (1980) 63-86.

[Ellis 83] ELLIS, C. Extendible hashing for concurrent operations and distributed data. Proc. Second ACM SIGACT-SIGMOD Symp. on Principles of Data Base Sys., Atlanta (1983) 106-115

[Ford, Calhoun 84] FORD, R. and CALHOUN, J. Concurrency control mechanisms and the serializability of concurrent tree algorithms. Proceed. of the ACM Symp. on the Princ. of Database Syst. 1984

[Ford et al 85] FORD, R., JIPPING, M., and SHULTZ, R. On the performance of an optimistic concurrent tree algorithm. Technical Report 85-07, Dept. of Comp. Science, University of Iowa, 1985

[Garcia 83] GARCIA-MOLINA, H. Using semantic knowledge for transaction processing in a distributed database. ACM Trans. on Database Syst. 8,2 (1983)

[Goodman, Shasha 85] GOODMAN, N., SHASHA, D. Semantically-based concurrency control for search structures. ACM SIGACT-SIGMOD Symp. on Princ. of Database Syst., 1985.

[Guibas, Sedgewick 78] GUIBAS, L., and SEDGEWICK, R. A dichromatic framework for balanced trees. Proc. 19th Annual Symposium of Foundations of Computer Science, 1978, 8-21

[Holt 83] HOLT, R. C. *Concurrent Euclid, the Unix system and Tunis, Addison-Wesley, Reading, Mass., 1983*

[Kung, Lehman 80] KUNG, H.T., and LEHMAN, P. Concurrent Manipulation of Binary search trees. ACM Trans. on Database Syst. 5,3 (1980) 339-353

[Kedem, Silberschatz 83] KEDEM, Z. and SILBERSCHATZ, A. Locking protocols: from exclusive to shared locks. J. of the ACM 30,4 (1983) 787-804

[Kwong, Wood 82] KWONG, Y. S. and WOOD, D. Method for concurrency in B-trees. IEEE Trans. on Software Engineering SE-8,3 (1982) 211-223

[Lehman, Yao 81] LEHMAN, P., and YAO, S. B. Efficient locking for concurrent operations on B-trees. ACM Trans. on Database Syst., 6,4 (Dec. 1981) 650-670.

[Mond, Raz 85] MOND, Y. and RAZ, Y. Concurrency control in B^+-trees databases using preparatory operations. Proceedings of Internat. Conf. on Very Large DataBases, 1985, 331-334

[Owicki, Gries 76] OWICKI, S., and GRIES, D. An axiomatic proof technique for parallel programs 1. Acta Inf. 6,1 (1976) 319-340

[Papadimitriou 79] PAPADIMITRIOU, C. H. Serializability of concurrent database updates. JACM, 26, 4 (Oct. 1979) 631-653

[Sagiv 85] SAGIV, Y. Concurrent operations on B-trees with overtaking. ACM SIGACT-SIGMOD Symp. on Princ. of Database Syst., 1985, 28-37

[Sagiv 86] SAGIV, Y. Concurrent operations on B^*-trees with overtaking. Manuscript, to appear in JCSS. Stanford Univ., Dept. of Comp. Sci., Stanford, Calif. (1986)

[Salzberg 85] SALZBERG, B. Restructuring the Lehman-Yao tree. Manuscript. College of Comp. Sci., Northeastern Univ., Boston, Mass. (1985)

[Samadi 76] SAMADI, B. B-trees in systems with multiple users. Inf. Process. Lett. 5, 4 (Oct. 1976) 107-112

[Shasha 84] SHASHA, D. Concurrent algorithms for search structures. Ph. D. Thesis, Harvard University. TR-12-84, Junè 1984

[Schultz, Mukkamala 85] SHULTZ, R., and MUKKAMALA, R. Multiprocessor B-link Tree Access. Manuscript. Dept. of Comp. Science, University of Iowa, 1985

APPENDIX

The following implementation of our algorithm is written mostly in Pascal. Two non-Pascal constructs are used: the use of records as return values of functions, and the use of *spawn* as a way of invoking processes. A value of a record type is represented as $(f_1, \cdots, f_n)$. The semantics of *spawn*(procedure call) are to create a process to perform the procedure call. No further communication takes place between the spawner and the process spawned, and the two execute at the same time and terminate independently. The parameters to this procedure must be entirely call-by-value; in particular, the descent stacks must actually be copied[1]. No requirement is placed on the length of time that may pass before the spawned process becomes active. If the drain technique is used, the starting time of each spawned process must be recorded as the starting time of the parent process.

[1] This restriction may be lifted for those parameters that will not be accessed again by the spawner (or passed to another spawned process).

The ascent is performed by creating a new process for each add-link or remove-link to be done. We choose this method for simplicity of presentation, although it is more efficient to spawn fewer processes. For example, the add-link and remove-link necessitated by a rotate are usually both done in the same node and should thus be done by a single process. However, this can not always be done and makes the code more complex, although this heuristic was implemented for the simulation. Whatever method is used, it can not make one add or remove wait for the completion of another if the other is redundant and must wait, since this can cause deadlock.

```
type
    locktype = (readlock, writelock);
    nodeptr = ^node;
    height = 1 .. maxint;
    task = (add, remove);

var
    anchor: record
        fast: nodeptr; fastheight: height;
        top: nodeptr; topheight: height;
    end;

function search(v: value): boolean;
    var
        n: nodeptr;
        descent: stack;
    begin
        n := locate-leaf(v, readlock, descent);
        {$v \in \mathit{coverset}(n)$, n read-locked}
        search := check-key(v, n); {decisive}
        unlock(n, readlock)
    end;

function insert(v: value): boolean;
    var
        n: nodeptr;
        descent: stack;
    begin
        n := locate-leaf(v, writelock, descent);
        {$v \in \mathit{coverset}(n)$, n write-locked}
        insert := add-key(v, n); {decisive}
        normalize(n, descent, 1);
        unlock(n, writelock)
    end;

function delete(v: value): boolean;
    var
        n: nodeptr;
        descent: stack;
    begin
        n := locate-leaf(v, writelock, descent);
        {$v \in \mathit{coverset}(n)$, n write-locked}
        delete := remove-key(v, n); {decisive}
        normalize(n, descent, 1);
        unlock(n, writelock)
    end;

function locate-leaf(v: value; lastlock: locktype; var descent:
    stack): nodeptr;
    { locate-leaf descends from the anchor to the leaf whose
    coverset includes v, places a lock of kind specified in lastlock
    on that leaf, and returns a pointer to it. It records its path in
    the stack descent. }
    var
        n,m: nodeptr;
        h,enterheight: height;
        ubleftsep: value;
        { ubleftsep stands for "upper bound on the leftsep of the
        current node". This value is recorded for each node on
        the descent stack so that an ascending process can tell if
        it's too far to the right. }
    begin
        lock-anchor(readlock);
        n := anchor.fast; enterheight := anchor.fastheight;
        ubleftsep := $-\infty$;
        unlock-anchor(readlock);
        set-to-empty(descent);
        for h := enterheight downto 2 do begin { $v > \mathit{leftsep}(n)$}
            move-right(v, n, ubleftsep, readlock);
            { $v \in \mathit{coverset}(n)$ }
            push(n, ubleftsep, descent);
            (m, ubleftsep) := find(v, n, ubleftsep);
            { $v > \mathit{leftsep}(m)$ }
            unlock(n, readlock);
            n := m
        end;
        move-right(v, n, ubleftsep, lastlock); {$v \in \mathit{coverset}(n)$ }
        locate-leaf := n
    end;

procedure move-right(v: value; var n: nodeptr; var ubleftsep:
    value; rw: locktype);
    { move-right scans along a level starting with node n until it
    comes to a node into whose coverset v falls (trivially, n
    itself). It assumes that no lock is held on n initially, and
    leaves a lock of the kind specified in rw on the final node. }
    var
        m: nodeptr;
    begin {assume $v > \mathit{leftsep}(n)$}
        lock(n, rw);
        while empty(n) or (rightsep(n) < v) do begin
            { $v > \mathit{leftsep}(n)$ }
            if empty(n) then
                m := outlink(n) { $v > \mathit{leftsep}(n) = \mathit{leftsep}(m)$ }
            else begin
                m := rightlink(n); {$v > \mathit{rightsep}(n) = \mathit{leftsep}(m)$
                    }
                ubleftsep := rightsep(n);
            end;
            unlock(n, rw);
            lock(m, rw)
            n := m;
        end;
    end;

procedure normalize(n: nodeptr; descent: stack; atheight: height);
    { normalize makes sure that node n is not too crowded or
    sparse by performing a split or merge as needed. A split may
    be necessary after a merge. n is assumed to be write-locked.
    descent and atheight are needed to ascend to the level above
    to complete a split or merge. }
    var
        sib, newsib: nodeptr;
        sep, newsep: value;
    begin
        if too-sparse(n) and (rightlink(n) <> nil) then begin
            sib := rightlink(n);
            lock(sib, writelock);
            sep := half-merge(n, sib);
            unlock(sib, writelock);
            spawn(ascend(remove,    sep,    sib,    atheight+1,
                descent))
        end;
        if too-crowded(n) then begin
            allocate-node(newsib);
            newsep := half-split(n, newsib);
            spawn(ascend(add, newsep, newsib, atheight+1,
                descent))
        end
    end;

procedure ascend(t: task; sep: value; child: nodeptr; toheight:
    height; descent: stack);
    { adds or removes separator sep and downlink to child at
    height toheight, using the descent stack to ascend to it. }
    var
        n: nodeptr;
        ubleftsep: value;
    begin
        n := locate-internal(sep, toheight, descent)
        while not add-or-remove-link(t, sep, child, n, toheight,
            descent) do begin
            { wait and try again. very rare }
            unlock(n, writelock);
            delay; { sep > leftsep(n) }
            move-right(sep, n, ubleftsep, writelock)
            { sep ∈ coverset(n) }
        end;
        normalize(n, descent, toheight);
        unlock(n, writelock)
    end;

function add-or-remove-link(t: task; sep: value; child: nodeptr; n:
    nodeptr; atheight: height; descent: stack): boolean;
    { tries to add or removes sep and downlink to child from
    node n and returns true if succeeded. if removing, and sep is
    rightmost in n, merges n with its right neighbor first. (if the
    resulting node is too large, it will be split by the upcoming
    normalization.). A solution that avoids this merge exists, but
    we present this for the sake of simplicity. }
    var
        sib: nodeptr;
        newsep: value;
    begin
        if t=add then add-or-remove-link := add-link(sep, child,
            n)
        else begin {t=remove}
            if rightsep(n)=sep then begin
                { the downlink to be removed is in n's right
                neighbor. }
                sib := rightlink(n);
                {rightsep(n) = sep < +∞, thus
                rightlink(n)<>nil}
                lock(sib, writelock);
                newsep := half-merge(n, sib); {newsep = sep}
                unlock(sib, writelock);
                spawn(ascend(remove, newsep, sib, atheight+1,
                    descent))
            end;
            add-or-remove-link := remove-link(sep, child, n)
        end
    end;

function locate-internal(v: value; toheight: height; var descent:
    stack): nodeptr;
    { a modified locate phase; instead of finding a leaf whose
    coverset includes v, finds a node at height toheight whose
    coverset includes v. if possible, uses the descent stack (whose
    top points at toheight) }
    var
        n, m, newroot: nodeptr;
        h, enterheight: height;
        ubleftsep: value;
    begin
        if empty-stack(descent) then
            ubleftsep := +∞ { force new descent }
        else pop(n, ubleftsep, descent);
        if v <= ubleftsep then begin
            { a new descent from the top must be made}
            lock-anchor(readlock);
            if anchor.topheight < toheight then begin
                unlock-anchor(readlock); lock-anchor(writelock);
                if anchor.topheight < toheight then begin
                    allocate-node(newroot);
                    grow(newroot)
                end;
                unlock-anchor(writelock); lock-anchor(readlock)
            end;
            if anchor.fastheight >= toheight then begin
                n := anchor.fast;
                enterheight := anchor.fastheight
            end
            else begin
                n := anchor.top;
                enterheight := anchor.topheight
            end;
            ubleftsep := -∞; { v > leftsep(n) }
            unlock-anchor(readlock);
            set-to-empty(descent);
            for h := enterheight downto toheight+1 do begin
                { v > leftsep(n) }
                move-right(v, n, ubleftsep, readlock);
                { v ∈ coverset(n) }
                push(n, ubleftsep, descent);
                (m, ubleftsep) := find(v, n, ubleftsep);
                { v > leftsep(m) }
                unlock(n, readlock);
                n := m
            end
        end;
        { v > leftsep(n), height of n = toheight }
        move-right(v, n, ubleftsep, writelock); { v ∈ coverset(n) }
        locate-internal := n
    end;

procedure critic;
    { the critic runs continuously; its function is to keep the target
    of the fast pointer in the anchor close to the highest level
    containing more than one downlink. }
    var
        n, m: nodeptr;
        h: height;
    begin
        while true do begin
            lock-anchor(readlock);
            n := anchor.top; h := anchor.topheight;
            unlock-anchor(readlock);
            lock(n, readlock);
            while numberofchildren(n)<=3 and rightlink(n)=nil
                and h>1 do begin
                m := leftmostchild(n);
                unlock(n, readlock);
                n := m;
                lock(n, readlock);
                h := h - 1
            end;
            unlock(n, readlock);
            lock-anchor(readlock);
            if anchor.fastheight = h then
                unlock-anchor(readlock)
            else begin
                unlock-anchor(readlock);
                lock-anchor(writelock);
                anchor.fastheight := h; anchor.fast := n;
                unlock-anchor(writelock)
            end;
            delay
        end
    end;
```

```
{ The search structure operations, arranged alphabetically.
    Locking ensures that they are atomic. }

function add-key(v: value; n: nodeptr): boolean;
    { if v is not a key in n then it is added into the sequence of
    keys in the leaf n at an appropriate location and true is
    returned. otherwise, return value is false. }

function add-link(s: value; child, parent: nodeptr): boolean;
    { The smallest index i in parent such that $s_i \geq s$ is identified.
    If $s_i = s$, the operation returns false. Otherwise, it changes the
    sequence in parent to $( \cdots ,p_i,s,child,s_i, \cdots )$, and returns
    true. }

function check-key(v: value; n: nodeptr): boolean;
    { returns true if v is a key in n, false otherwise. }

function empty(n: nodeptr): boolean;
    { returns true if n is an empty node. }

function find(v: value; n: nodeptr; ubleftsep: value): (nodeptr,
    value);
    { The smallest $s_i$ in n such that $v \leq s_i$ is identified. If $i>1$,
    returns $(p_i, s_{i-1})$, otherwise $(p_i$, ubleftsep).}

procedure grow(n: nodeptr);
    { n is made an internal node containing only a downlink to
    the current target of the anchor's top pointer and the
    separator $+\infty$ to its right. The anchor's top pointer is then
    set to point to n, and its height indicator is incremented. }

function half-merge(l, r: nodeptr): value;
    { The sequence in r is transferred to the end of the sequence
    in l. The rightlink of l is directed to the target of the
    rightlink in r. r is marked empty, its outlink pointing to l. If l
    is a leaf, its separator is set to the largest key in it. The
    previous value of the rightmost separator in l is returned. }

function half-split(n, new: nodeptr): value;
    { The rightlink of new is directed to the target of the rightlink
    of n. The rightlink of n is directed to new. The right half of
    the sequence in n is moved to new. If n and new are leaves,
    their separators are set equal to the largest keys in them. The
    return value is the new rightmost separator in n. }

function leftmostchild(n: nodeptr): nodeptr;
    { returns the leftmost downlink in n. }

function numberofchildren(n: nodeptr): integer;
    { returns number of downlinks in n. }

function outlink(n: nodeptr): nodeptr;
    { returns the outlink in node n. }

function remove-key(v: value; n: nodeptr): boolean;
    { if v is a key in n, it is removed and true is returned.
    otherwise, return value is false. }

function remove-link(s: value; child, parent: nodeptr): boolean;
    { If the sequence in parent includes a separator s on the
    immediate left of a downlink to child, the two are removed
    and true is returned; otherwise the return value is false. }

function rightlink(n: nodeptr): nodeptr;
    { returns the rightlink in node n. }

function rightsep(n: nodeptr): value;
    { returns the rightmost separator in n. }

function too-crowded(n: nodeptr): boolean;
    { returns true if n contains too much information. }

function too-sparse(n: nodeptr): boolean;
    { returns true if n contains too little information. }
```

ARCHITECTURE OF A FIBER OPTICS BASED DISTRIBUTED INFORMATION NETWORK FORTIS: Local Area Network

Paul C. Barr* - Suban G. Krishnamoorthy**

Aetna Telecommunication Labs, Westboro, Mass
* Northeastern University, Boston and MITRE corp. Bedford, Mass
** Framingham State College, and Prime Computer, Framingham, Mass

ABSTRACT

This paper describes a fiber optics based distributed information network named FORTIS. FORTIS is an integrated (voice, data, video, and sensor), end-to-end digital packet switching system. FORTIS is implemented as a multilevel, multiprocessor distributed network system using MPUs, I/O processors and the latest VLSI components. The system can modularly expand from a one device-one node system, to a subnet, net and supernet. The system can be connected to existing networks and various data LANs, as well as serve as a data extension to existing PBXs. Currently FORTIS can support the RS232C asynchronous protocol for data, and RJ11 physical connection for phones. It will support other higher level protocols in the near future.

INTRODUCTION

There are three major areas of computer applications growing in importance during the next decade. One, as is traditional, will be the use of computers for the purpose of numerical analysis [1]. This computer application will require computers with greater throughput than presently available, especially for applications in meteorology, nuclear physics and modeling large scale systems.

The second major area of computer applications concerns the massive quantity of data in large databases used for decision support systems [2]. This application area will continue to play a growing role in banking, insurance, record processing, information retrieval, etc. In these applications, the number-crunching capabilities of traditional computers are not as important as the capability to store and retrieve large amounts of data rapidly and at reasonably low cost. Computer networking will play an expanding role in these applications, for they will have to access not just a single database but a collection of interconnected databases. In response to this need, many advances in computer networking are required [3].

The third major area for computer applications is what has become to be known as "Knowledge Engineering". Knowledge engineering is one of the major areas of Artificial Intelligence [4]. Within knowledge engineering, a field of primary importance is the expert system. The required computers for this application are different than those needed in the other two areas.

All of these computer application areas are growing in importance, some more rapidly than others. This paper primarily addresses the second area: the accessing of interconnected databases. It also addresses the interconnection of all types of communication devices such as telephones, data terminals, workstations, disks, printers, plotters and sensors, as well as computers.

Technology in areas of voice communications, data communication, distributed computing, and fiber optics have evolved in parallel but through fairly independent approaches. The creation of functionally and physically independent networks to share resources for voice communication and data communication is inefficient and provides for a costly solution to the user's management of information. Presently, users are increasingly demanding a convergence of these technological approaches as a result of their need to share voice and data information by more convenient and less costly methods.

A traditional approach to handling intrapremise communication consists of a centrally located circuit switch such as a PBX which connects pairs of intercommunicating devices for the duration of a session. This approach is adequate for voice applications since voice traffic is continuous but is inefficient for interactive data terminal usage. The capacity of the PBX circuit switch is inefficiently used in the case of the in-

CH2345-7/86/0000/0390$01.00 © 1986 IEEE

teractive data terminal usage since most data sessions consist of short bursts with long idle periods. A local area network which uses packet switching for interconnection is superior to circuit switching because the network connection is not used during idle periods and is available for use by others.

The Fiber Optics based Distributed Information Network (FORTIS) is a state of the art, end-to-end digital packet switching system which is capable of fully integrating voice, data, video communication and other types of information, such as sensor data, into an intelligent distributed system. FORTIS utilizes a packet switching technique to permit devices connected to the network to transmit in bursts, thereby allowing the bandwidth of the network to be used more efficiently. At the same time, it preserves the real time nature of voice communication by prioritizing it.

The system provides to the user the properties of economy (cost per connection), integrity (robust and fault tolerant design), performance (parallel processing through local and network distribution) and modularity (expandable and simple to maintain). It uses multiple dsitributed microprocessors and the latest in VLSI components to provide maximum programmability, reliability and flexibility at minimum cost.
The system can also be configured with existing networks and various data LANs [5-10] as well as a data extension to an existing PBX. The FORTIS system is described in detail in the rest of the paper.

FORTIS SYSTEM DESCRIPTION

By design, the FORTIS system is modular. The system can be visualized in three ways: (1) as a standalone node, (2) as a subnet with two or more nodes linked together and (3) as a network with two or more subnets connected together. Each of these structures is explained in detail in this section.

The node design is a unique three level distributed processor architecture as shown in figure 1. A design requirement for the standalone node was to provide the capability of allowing 256 devices (telephones and/or terminals) to communicate with each other in a non-blocking environment. The design was also predicated upon being event-driven, rather than polled. It was evident that a single or multiple processor (68000 class) connected in a traditional bus structure would not be able to handle that many devices in real time. An architecture was needed which exhibited the characteristics of pipelining and parallel processing, as well as asynchronous processing. It was also determined that a bus structure which permitted parallel transfers was necessary. The rationale for this unique architecture was driven by the tradeoff of speed versus intelligence. The more intelligence demanded of a processor, the more time it takes to achieve it. Level 1 is the intelligence level. Level 2 peforms the task of I/O and is the device handler level. Level 3 is the communication level whose task is to communicate to devices in a network that are external to the node.

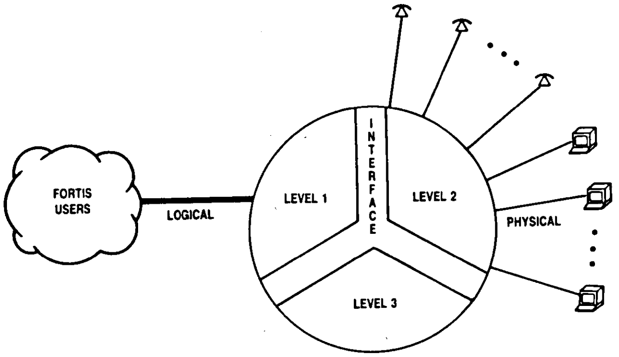


Figure 1. FORTIS: NODE ARCHITECTURE

As depicted in figure 1, a user is logically connected with level 1. Level 1 contains the node specific information such as the types of devices connected to the node, the list of valid node users, etc., as well as subnet and network specific information. Level 1 also has the necessary system software for supporting user interface, establishing connectivity between various resources in the system, and for system monitoring. A more detailed configuration of Level 1 is shown in figure 2a. As shown, it is a traditional architecture using a 68000 MPU interconnected with a VME bus structure and a disk subsystem.

User resources such as terminals, phones, (micro, mini, and mainframe) processors, etc., are physically connected at level 2. The devices in this level are configured into clusters. Each cluster can have up to eight devices of the same

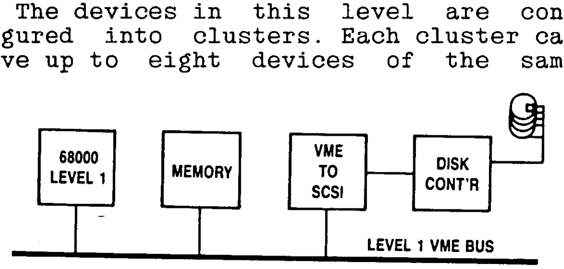


Figure 2A. LEVEL 1 ARCHITECTURE

type, as shown in figure 2b for the phone cluster and figue 2c for the terminal cluster. The minimum configuration is one cluster with up to 8 devices.The system can expand by modules of eight devices, up to 256 devices (or 32 clusters of 8 devices per cluster). Clustering increases the modularity and reduces the cost of the system.

Each of the clusters of level 2 has a processor, memory and enough intelligence to handle packet creation and transfers. As shown in figure 2, a cluster is composed of three logical elements: (1) the cluster control unit (CCU) (2) a device line interface card (telephone or data terminal - TLIC or DLIC) and (3) a physical card to hold the connectors (PIC or DIC). The CCU is a high speed I/O processor (5 mips). Level 2 handles intracluster and intercluster information exchange within a node. It has the ability to do self-diagnosis and to report any abnormality or error conditions to the network/node manager resident in level 1.

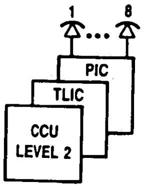


Figure 2B. LEVEL 2 PHONE CLUSTER (PC)

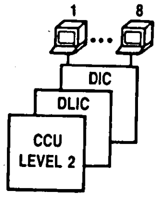


Figure 2C. LEVEL 2 TERMINAL CLUSTER (TC)

The physical connectivity of level 1 and level 2 with the parallel bus structure is shown in figure 2d. Level 1 communicates to level 2 clusters via the I/O bus controller. This path is selected when level 1 intelligence sets up the path between devices (for example, source device 1 in cluster 4 is to be virtually connected to destination device 1 in cluster 7 for packet transfers). The actual packet transfer is accomplished in a burst mode on the level 3 burst bus. The burst bus controller performs a hardware poll of all clusters requesting the use of the burst bus. The arbitration is accomplished by ring priority; the last to use the bus has the lowest priority. The actual transfer between clusters is achieved at a 10 megabyte rate (a 64 byte transfer is achieved in 6.4 microseconds).

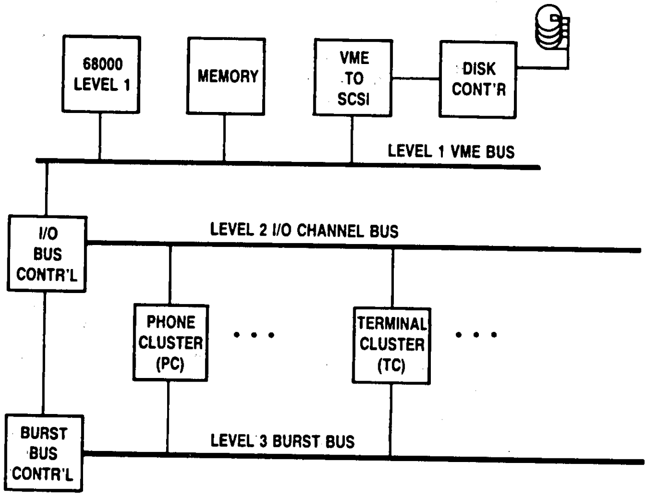


Figure 2D. FORTIS: NODE ARCHITECTURE

The FORTIS architecture design is aimed at providing a maximum of asynchronous parallel operation at all levels by design of multiple parallel processors and multiple parallel bus structures within each node. Each level has at least one processor, and memory. Level 2 could have as many as 32 processor with 32 programming streams executing in parallel.

Subnet Architecture

The architecture of the FORTIS system allows networked expansion of the self-standing nodes. Two or more nodes could be interconnected through passive fiber optic star couplers to form a subnet as shown in figure 3a. The connectivity using a passive star coupler between nodes operates in a broadcast fashion and therefore is a fully interconnected topology.

After level 1 has determined the connectivity, level 2 performs the packet transfers. A packet transfer is achieved by a handshake and a burst transfer to another level 2 cluster or to level 3. Level 3 is the off-node communication handler. As depicted in figure 3b, level 3 consists of a cluster control unit (CCU) identical to the level 2 CCU's (and a programming stream), a link line interface card (LLIC) whose function is to provide the interface between the CCU and the optics interface card (OIC). The OIC contains the following modules: (1) byte to bit serial and bit serial to byte convertors, (2) modulator and demodulator to

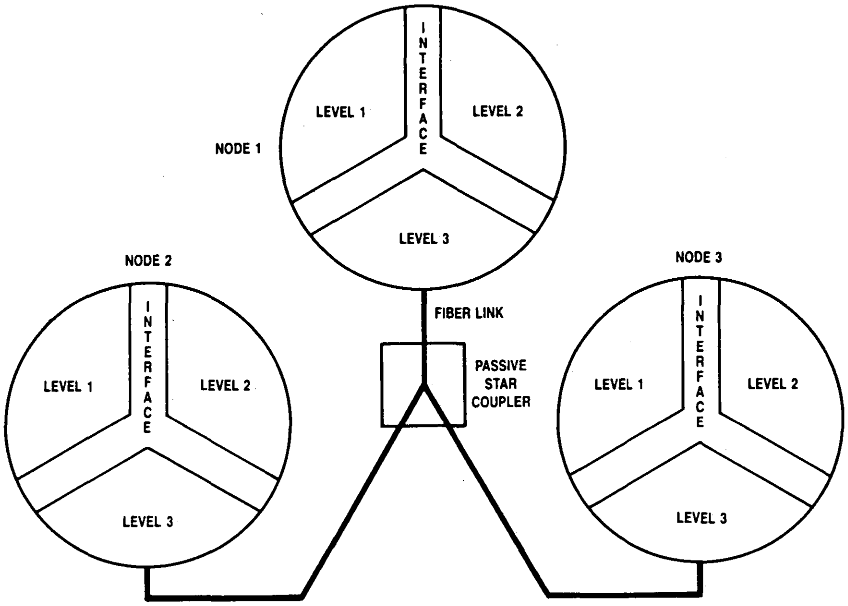


Figure 3A. FORTIS: SUBNET ARCHITECTURE

drive the fiber optic transmitter, (3) LED or laser transmitter to launch the energy onto the fiber, (4) receiver module designed to settle in 4 bit times, and (5) high speed state machine to provide real time state information to the level 3 CCU.

The link arbitration protocol is a priorty round-robin technique. This technique combines priority with a virtual token passing scheme. With the combination of packet switching and the priority round robin link protocol, a real time response of 8 milliseconds is guaranteed for resources such as phones. The same modular design philosophy is followed in deployment of the fiber communications link. A minimum of two links is provided, each operating at a 20Mhz rate. More than one link supports a more robust system operation, and eliminates any single point failure mechanism [11]. However, the system is capable of operating on a single link. If more throughput on the communication between nodes is desired, the system is designed to stack multiple links. As technology advances, the link rate could be increased to the 40-100Mhz range. Also, the modular design of FORTIS enables the use of advanced technology such as Wave Division Multiplexing (WDM) [12].

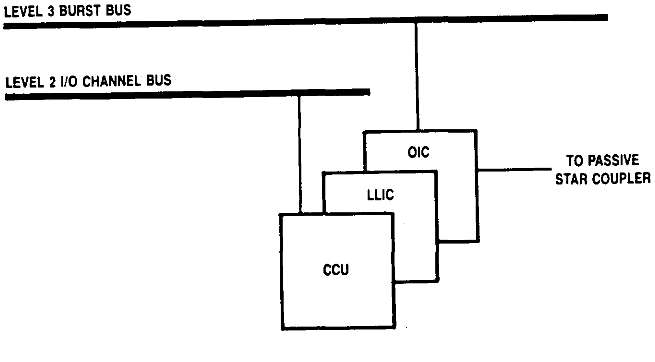


Figure 3B. FORTIS: FIBER LINK ARCHITECTURE

The minimum number of nodes in a subnet is two; the maximum number in a single subnet is 20. The maximum is governed by present fiber component technology and optical power budget calculations. As fiber technology advances, this number can increase to the logical FORTIS addressing limit of 256 nodes per subnet. The framework of each of the nodes in a subnet is identical and represents an independently operating part of the system. The intelligent nodes therefore operate in a distributed fashion which is inherently fault tolerant.

Network Architecture

Subnets can be interconnected via bridges to form a FORTIS net as shown in figure 4a. The subnets and/or nets can also be connected to other networks such as Ethernet [10], through gateways as in figure 4b. FORTIS allows 256 subnets per net. Two hundred and fifty six such nets can be interconnected to form a supernet as in figure 4c. (Practical problems in designing nets and supernets are currently under study.) FORTIS has a total address space of 256x256x256x256 = 4,294,967,296 devices. At the same time

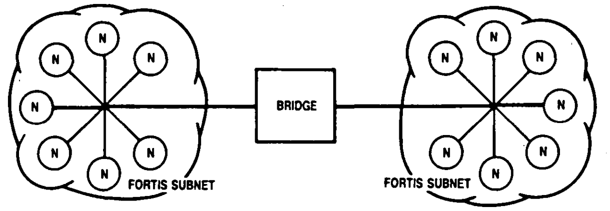


Figure 4A. FORTIS NET USING SUBNETS

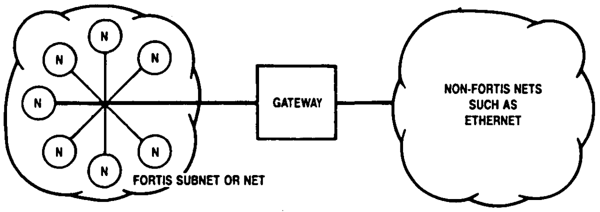


Figure 4B. FORTIS NET WITH OTHER NETWORKS

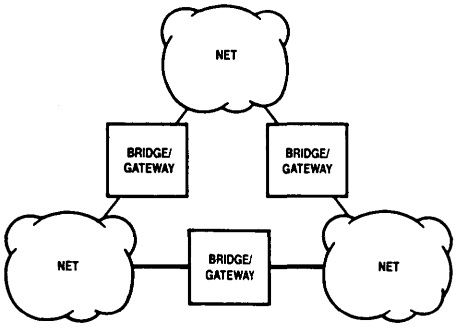


Figure 4C. FORTIS SUPERNET

it is modular enough to grow in small steps of one cluster to satisfy the future requirements of customers. Modular expansion alleviates large capital expense problems associated with capacity planning.

Software Structure

The FORTIS system was designed to allow software visibility to all levels of the hardware. This allows greater programmability and more extensive diagnostics. Resident in each of the nodes is a real-time networking operating system. The software is structured in a layered fashion as shown in figure 5. Also, 99% of the code is written in C language. Hence, it is easy to debug, integrate, maintain and transport the software.

The FORTIS software can be divided into three major categories: (1) system software, (2) application software and (3) diagnostic software. The system software performs major networking functions. It can be further classified into: distributed real-time operating system, networking, user interface and system microcode. The distributed real-time opeating system performs functions such as multitasking, memory management, interrupt handling, real time clock, error processing, performance monitoring, etc. The networking software has many modules to perform network related functions such as node supervisory, phone supervisory, data supervisory, global and nodal systems management and monitoring. The system microcode resides in level 2 and level 3 of the node. It performs tasks such as packet transfer between different devices anywhere in the FORTIS system.

The user interface is the logical connection between the FORTIS users and level 1 mentioned earlier. This portion of software provides user friendly interfaces to phone and data including help provision. It also has a command processor to process all the user commands. An example of commands provided for the data users include connectivity, programmability, and positive flow control which allows speed matching. With a standard 2500 telephone set, the phone commands include standard PBX features such as hold, transfer, forward, pickup, etc.

The FORTIS application software consists of voice mail and electronic mail. In the future, it will be possible to provide many more application software packages such as word processors, distributed file servers, database management, language processors and even computing power.

The final piece of system software is the diagnostic software. Diagnostic software exist at all levels to test various parts of FORTIS system. Provisions exist to diagnose individual boards, each node, and the link in the network.

FORTIS also provides many value-added features to the end-users as well as to the network. Some of the value-added network features are: accounting, load balancing, security by password and encryption, on-line help, system manager, system monitor.

It is important to note that each node has information about its own devices/resources only. Information about devices/reources existing in other nodes in the system are obtained 'on demand' basis in real time. This allows dynamic changes to resources in a node without any need to update or inform other nodes in the system. The totality of information is distributed across the network.

Network Protocols

Currently FORTIS system supports RS232 asynchronous protocol. Data terminals and

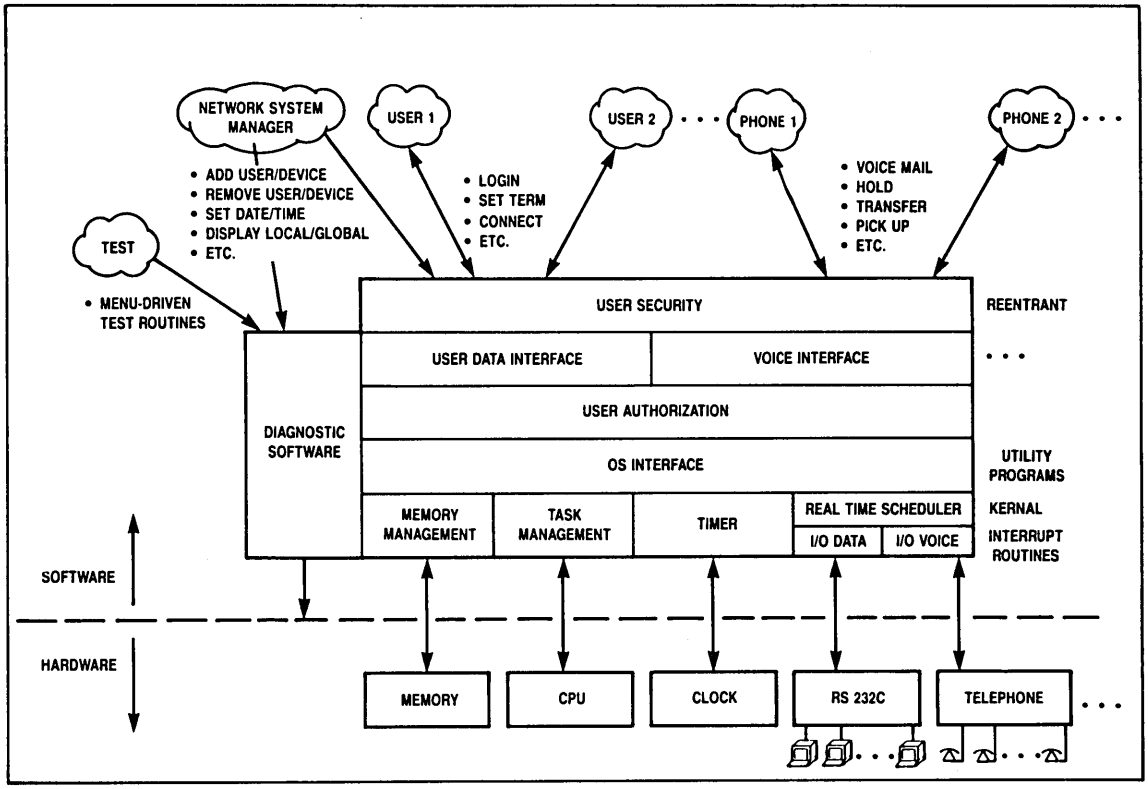


Figure 5. FORTIS SOFTWAY STRUCTURE

CPUs are connected to the system through RS232 lines. Phones are connected to the system using standard RJ11 connectors. However, in the future, FORTIS can easily support other higher level protocols such as HDLC, X.25, ISO-OSI, SNA, etc. [3,13].

FORTIS APPLICATIONS

The typical environments where FORTIS is applicable today and in the near future include: offices, companies and universities, factories, laboratories, hospitals, distribution and sales, banking, easily maintainable and cost saving building wiring distribution, and so on. FORTIS supports office automation functions such as electronic mail, text/word processing, document distribution, and voice mail. It provides communication, links for accessing centralized or distributed databases and mainframes using terminals or teller machines. It also provides a medium for factory automation, including automated manufacturing techniques such as CAD/CAM, robotics, and numerically controlled manufacturing processes requiring real-time response. It can provide the communication links which are needed for order entry and inventory control systems, patient diagnostics, file retrieval and status monitoring. Other user needs [14] are extensive network management services for system administration and maintenance. FORTIS allows the users to efficiently accomplish these required tasks and increase individuals' productivity. To further emphasize possible applications, two are explained in detail in the following paragraphs.

Multistory Building or Industrial Park

A typical application for the FORTIS is a multiple story building or an industrial park with multiple tenants. For the purpose of this example, a floor in a multistory building can be configured identically to a building in an industrial park. The pictorial view of this application is depicted in figure 6. As shown, it represents an eight story building or eight separate buildings in an industrial park. Each floor is divided into multiple tenants.

To distinguish business types of tenants, four classes are defined below: (a) class 1 busines - 64 phones, 32 terminals, (b) class 2 business - 64 phones, 8 terminals, (c) class 3 business - 16 phones, 4 terminals and (d) class 4 business - 8 phones, 0 terminals. The dis-

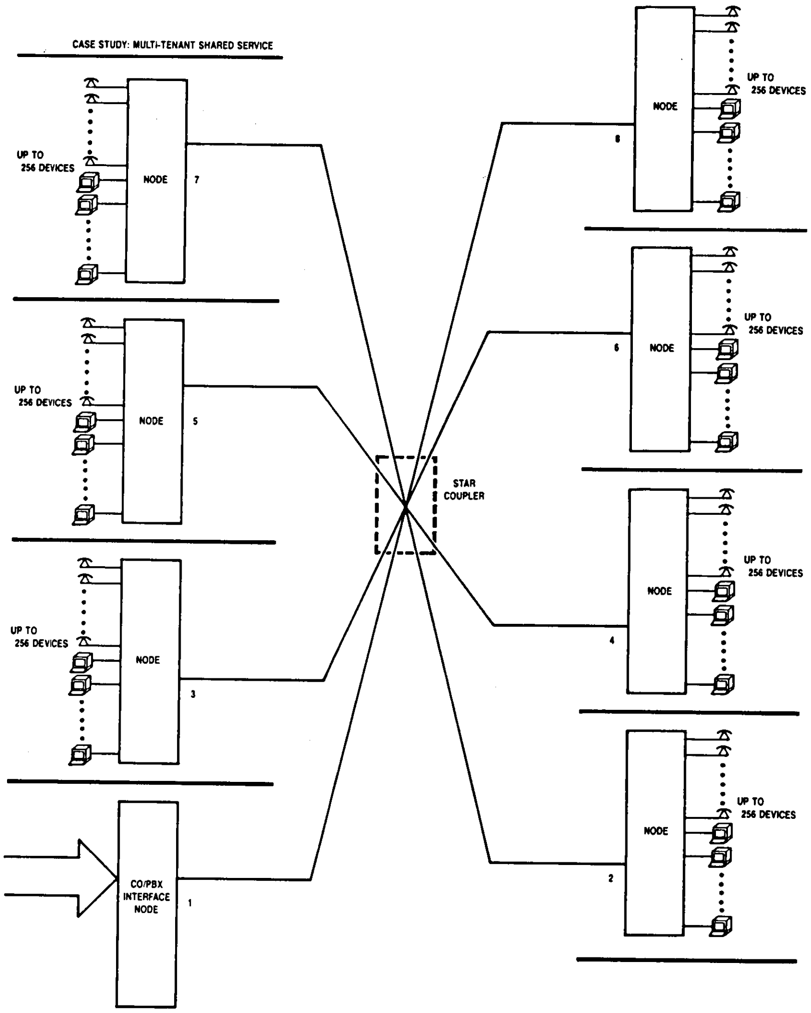


FIGURE 6 MULTIPLE STORY BUILDING OR INDUSTRIAL PARK

tribution of the multiple tenants and their respective business types for each of the floors is provided in Table 1.

Floor 1 will be reserved for the interface to the local central telephone office. The interface could be multiple wire-pairs as used in analog voice connections and/or T1 carrier and SLC 96 as used for digital connections. The cabling media to interface could be wire or fiber. All of the tenants share the trunk lines and the inherent lower costs of connection to the outside business world. This is true because the subnet pools the outside lines and requires fewer trunks to provide the same grade of service. In this application, the subnet connects 520

Table 1. Distribution of Multiple Tenants

floor/building	tenants	phones	terminals
8	one class 1	64	32
7	one class 1	64	32
6	one class 2	64	8
6	four class 3	16	4
5	one class 2	64	16
4	two class 3	64	8
4	one class 4	32	8
4	one class 1	8	0
3	one class 1	64	32
2	eight class 4	64	0
2	one class 3	16	4
totals	22	520	144

telephones for a grade of service of B.05 (4 ccs) and requires only 58 trunks or 3 T1 lines into the network. For the 22 tenants this reflects a savings of greater than 40 percent over that of each tenant bringing in individual trunk lines, as well as simpler cable installation. Further, each tenant has the potential to share a mainframe, word processing services, energy management and security to the building. The building management offers additional value to the user of the space which reflects a competitive edge over other building space of comparable value.

Data Networking Architecture

Examining the requirements of the data market in a little more detail could provide some further networking insight. There exist today two architectures for networking computer systems. These are the "Computer Network Architecture" and the "Terminal Access Network Architecture". The "Computer Network Architecture" is suited for CPU to CPU communication and sharing of peripherals (SNA of IBM Corp., DNA of Digital Equipment Corp., DEC Clustering Architecture, etc. [3].) and is an adjunct to "back-end processing". "Terminal Access Network Architecture" is, as the name implies, an interconnection of a terminal to a computer port and is an adjunct to "front-end processing". The relationship of both of these networks is shown in figure 7a. At present, both networks coexist in many computer systems. FORTIS can combine both of these networks into one for greater economy and ease of maintenance as shown in figure 7b.

SUMMARY AND CONCLUSION

This document describes the FORTIS network and provides the rationale for its development. FORTIS provides the user with real-time response which is a requirement for voice communication, and other applications such as process control. The network differs from many systems by providing a packetized virtual circuit between devices such as digitized voice, data, video, sensor interfaces and the central telephone office. Ease of use was an important consideration in the design of the node, subnet, net, operating system and other support software.

FORTIS provides a cost effective solution for interconnection requirements by providing an integrated information network with productivity enhancements provided by software value-added features. The truly integrated approach and the value-added capability provided by the software of the FORTIS system creates the foundation and framework for new solutions for a variety of application areas. The programmable FORTIS hardware framework allows quick development of specific interfaces as dictated by a rapidly changing market place.

Presently, three nodes interconnected as a subnet are operational in our Westboro facility. They are connected with two passive star couplers at a distance of 1 kilometer from each other with a link rate of 20 Mbps. Multiple telephones, terminals and CPUs are connected in the subnet. Aetna Telecom. Labs has many patents on various technical aspects of the FORTIS network.

The system described in this paper was conveived in the last quarter of 1982, designed and successfully implemented by the spring of 1985 as a fully populated three node system. An equal number of standard telephone sets and VT100 like terminals were connected to these nodes. Now that we have demonstrated the system, the team can reflect back and provide others with observations which we feel are meaningful.

In this age of openness, standards, and multi-vendor environments, propriety communication systems have a short future. The impetus for the work of the IEEE 802 LAN committee was that standardized LAN's are a requirement. The team extends this requirement further suggesting that there should be one LAN network which not only handles interconnection between mainframes, and mainframe peripheral equipment but also handles voice communication. Further we feel that not only physical integration between the telephone sets and terminals should be a requirement but logical integration as well.

It would be a serious omission on our part when a paper on high-speed fiber-based networks does not mension (the stanard currently being developed by the IEEE and ANSI) the Fiber Distributed Data Interface (FDDI), which is a 100 Megabit Token-ring communication system, using fiber. (FDDI-2 is an alternative specification using slots for synchronous (voice) data. A reference [15] is provided for additional information.

ACKNOWLEDGEMENT

The FORTIS net is the accomplishment of a highly talented team whose efforts sometimes knew no bounds. The authors express their sincere appreciation to all the members of Aetna Telecom. Labs for their cooperative efforts.

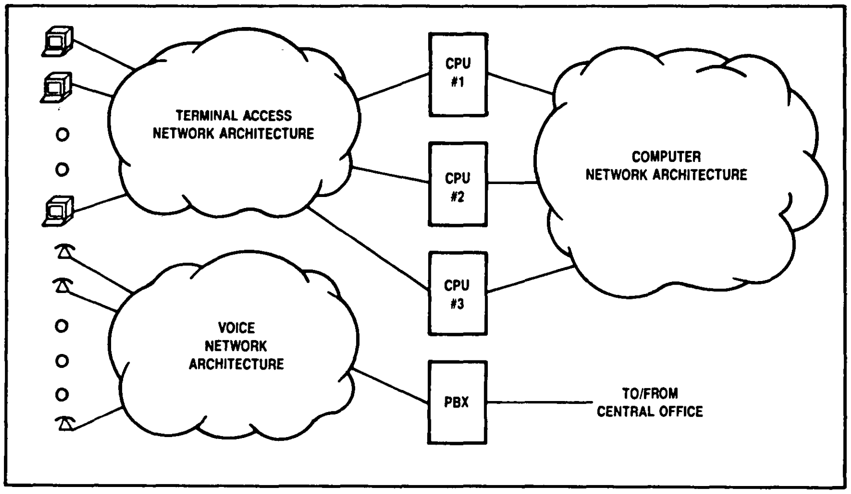

TERMINAL ACCESS NETWORK ARCHITECTURE
VOICE NETWORK ARCHITECTURE
CPU #1
CPU #2
CPU #3
PBX
COMPUTER NETWORK ARCHITECTURE
TO/FROM CENTRAL OFFICE

Figure 7A. PRESENT COMPUTER NETWORKING SCHEME

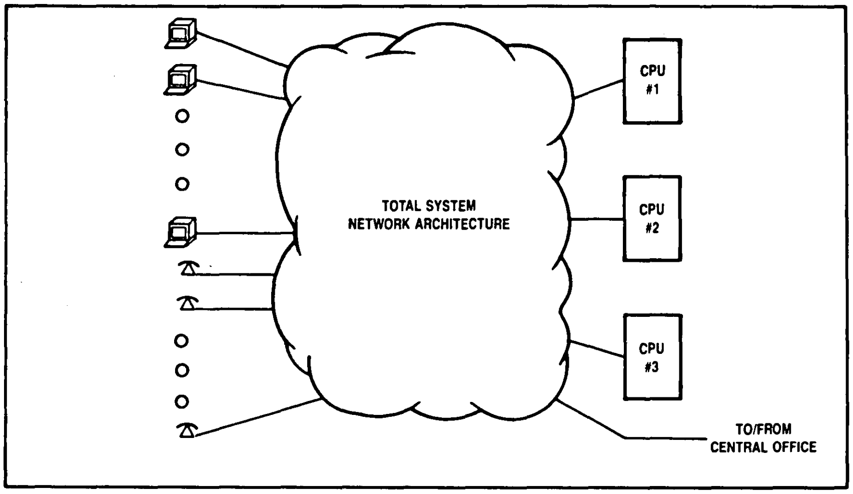


Figure 7B. FORTIS COMPUTER NETWORKING SCHEME

REFERENCES

[1] A. Ralston and P. Rabinowitz, A First Course in Numerical Analysis, 2nd Ed., McGraw-Hill, 1978.

[2] M. M. Gorman, Managing Data Base: Four Critical Factors, QED Information Sciences, Inc., 1984.

[3] A. S. Tanenbaum, Computer Networks, Prentice-Hall, Inc., 1981.

[4] R. Davis and D. B. Lenant, Knowledge-Based Systems in Artificial Intelligence, McGraw-Hill, 1982.

[5] K. Kummerle and M. Reiser, "Local Area Communication Networks-An Overview," J. Telecomm. Networks, Vol. 1, no. 4, pp. 349-370, 1982.

[6] M. Wilkes and D. Wheeler, "The Cambridge Digital Communication Ring," Proc. Local Area Comm. Network Symp., pp. 47-62, May 1979.

[7] B. K. Penney and A. A. Baghdadi, "A Survey of Computer Communication Loop Networks, part 1 and 2," Comput. Communications, Vol. 2, pp. 165-180 and pp. 224-241, 1979.

[8] W. Bux, F. Closs, P. Janson, K. Kummerle and R. R. Muller, "A Reliable Token-Ring System for Local Area Communication," NTC Proc., pp.A2.2.1-A2.2.6, 1981.

[9] C. David Tsao, "A Local Area Network Architecture Overview," IEEE Communications Magazine, Vol. 22, no. 8, pp. 7-11, 1984.

[10] R. M. Metcalfe and D. R. Boggs, "Ethernet Distributed Packet Switching for Local Computer Networks," Comm. ACM, Vol. 19, pp. 395-404, 1976.

[11] A. Albanese, "Fail-Safe Nodes for Lightguide Digital Networks," Bell Syst. Tech. J., Vol. 16, no. 2, pp. 247-256, 1982.

[12] B. Hillerich, et al, "Wavelength Division Multiplexing in Fiber Optics Systems," Telecommunications, pp 73-78, July 1985.

[13] H. Zimmerman, "OSI Refrence Model - The ISO Model of Architecture for Opening Systems Interconnection," IEEE Trans. C omm., COM-28, no. 4, pp. 425-432, 1980.

[14] G. J. Lanford, "Local Area Network User Needs," Localnet '83 Conf. Proc., pp. 31-40, Sept. 1983.

[15] F. E. Ross, R. K. Moulton, "FDDI Overveiw - A 100 Megabit per second Solution", 1984 WESCON Convention Proc., pp 2/1, 1 - 9 Sept. 1984.

ON THE DESIGN OF FAULT-TOLERANT SYSTOLIC ARRAYS WITH LINEAR CELLS

Chien-Yi Chen and Jacob A. Abraham

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

ABSTRACT

In many numerical systolic systolic arrays, each processing element in the regular part of the array is itself a linear system (we call this a linear cell). A systematic approach to the design of fault-tolerant systems for such systolic arrays is developed in this paper. Most of the proposed systolic arrays for matrix operations, polynomial operations, and digital signal processing can be made fault-tolerant using our procedure. The design procedure preserves the structure of the original (non-fault-tolerant) systolic array making it easy to incorporate fault tolerance; the faulty units can be identified, which permits reconfiguration if necessary.

The design methodology encodes the inputs data at a high level and ensures that the algorithm generates encoded output data; the encoding is tailored to the structure of the systolic array. The encoded input data are passed through the systolic array in ways which will avoid problems with error masking due to failures. This approach, therefore, results in extremely low overhead for fault tolerance.

I. INTRODUCTION

Conventional general purpose machines are unable to support, without high cost, the increasing demands for large amounts of computation in many real-time scientific and signal processing applications. Fortunately, the tremendous progress of VLSI technology has made feasible cost-effective parallel computation. Among these parallel schemes, systolic arrays have earned a great deal of research attention since they are amenable to special-purpose VLSI design. Many algorithms, well suited for VLSI implementation, have been developed for matrix multiplication, QR decomposition, LU decomposition, Discrete Fourier Transform (DFT), solution of Toeplitz linear systems, eigenvalue problems, etc [1]. However, along with the tremendous increase in the performance, reliability has become a serious consideration: all cells or Processing Elements (*PE*s) in the array need to be fault free for correct operation, and the overall system reliability decreases rapidly as the number of *PE*s increases.

Fault tolerance can be obtained either by masking the error caused by physical failures or by detecting them, isolating the faulty unit, and reconfiguring the system around the faulty unit. Fault masking is usually done by replicating the hardware and voting on the outputs of the replicated modules, but this is extremely costly in terms of hardware redundancy. An alternative is to detect errors by periodic testing. Unfortunately, the increase in transient errors caused by decreasing geometries means that off-line testing alone cannot be used to detect erroneous modules. Techniques for *Concurrent Error Detection (CED)* are therefore needed to detect errors concurrently with normal operation.

Fault-tolerant structures for systolic arrays have been proposed by many authors [2-7]. However, in all these papers, discussions are limited to the problem of *reconfiguration*, and the difficult problems of CED and *fault location* have not been discussed. Other papers [8-12] did discuss the problem of concurrent error detection. However, as far as systolic arrays are concerned, [8] and [9] can only be applied to matrix multiplication. In [10], each *PE* has to be implemented with PLAs, and this is impractical for VLSI implementation of the multipliers and adders. Techniques proposed in [11] and [12] require double or triple calculations to compare or vote, resulting in time penalties. It has been shown in [13-15] and [8] that we can always fully utilize the systolic arrays, i.e., with no idle *PE*s during the operation of the systolic array, even for the case of bidirectional systolic arrays. Therefore, duplicated or triplicated operations will certainly mean a large overhead.

In this paper, efficient fault-tolerant schemes for systolic arrays are developed through the analysis of some common features of most existing numerical systolic arrays. These require low hardware overhead and low time redundancy, and can be applied to most existing numerical systolic arrays, including those for matrix multiplication, Finite Impulse Response (FIR) filters, Infinite Impulse Response (IIR) filters, LU decomposition, matrix triangularization, least square minimization, convolution, polynomial multiplication, polynomial division, etc. It is well known that there is no clear definition of a systolic array. Generally speaking, according to [15], a systolic array is a computing network possessing the following features: (1) synchrony, (2) regularity, (3) temporal locality, and (4) pipelinability. From this definition, it is clear that systolic arrays will differ significantly in their specific structures. Moreover, even with the same structure, the systolic arrays may still differ a lot from each other due to the different functions of each individual *PE* and the irregular *PE*s at the boundaries. Thus, since systolic arrays are not clearly defined, it is likely that a universal approach toward the design of fault-tolerant schemes for diverse systolic arrays is extremely difficult, or even impossible. Therefore, in this paper, instead of trying to derive a scheme which can be applied to all systolic arrays, we set our goal to develop a scheme to successfully handle most of the existing systolic arrays for numerical computations.

In this paper, the general fault model is defined in Section II, and the linear cells and the linear property are defined in Section III. Section IV develops the CED design for one-dimensional systolic arrays using the linear property encoding, while Section V develops the CED design for two-dimensional systolic arrays. The fault location and reconfiguration problems are discussed in Section VI. finally, Section VII presents the conclusion of this paper.

This research was supported by the Semiconductor Research Corporation (SRC) under Contract SRC-RSCH 84-06-049-4.

II. FAULT MODEL

As the geometric features of integrated circuits become smaller, any physical defect which affects a given area on a chip will affect a greater amount of the circuitry and will cause a larger logic block to become faulty. Thus, the traditional gate level fault models are not satisfactory. In this paper, we allow a computation unit (such as multiplier or adder) to produce any arbitrary logical errors under failures. Furthermore, a single computation unit fault model is used, where we assume that at most one computation unit could be faulty within a given period of time which will be reasonably short compared with the mean time between failures.

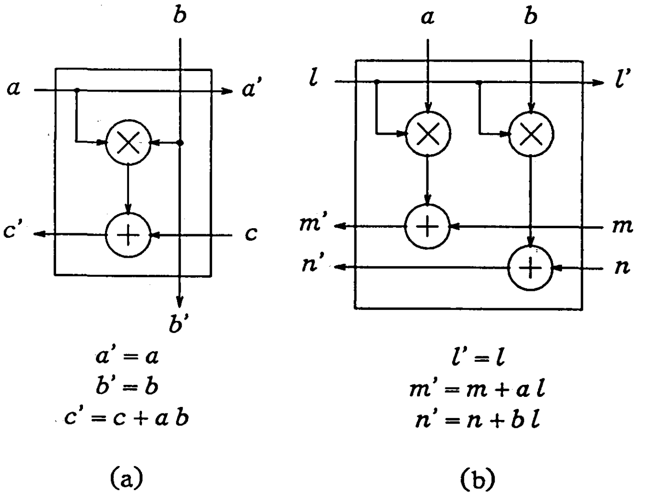
Since effective error detecting (or detecting and correcting) schemes, such as parity, Hamming codes [16] and alternate data retry [17], exist for communication lines and memories, failures in the communication lines and memories can be readily detected (or detected and corrected) by those methods. We therefore concentrate on detecting and correcting errors due to the failure of each computation unit, e.g., an adder or a multiplier. However, faults in the communication lines will also be specifically discussed (to allow the fault model to cover faulty lines and registers) in this paper.

III. LINEAR CELLS AND THE LINEAR PROPERTY

Through an extensive study of systolic schemes, we found an interesting feature which is common to most existing numerical systolic arrays proposed for Digital Signal Processing (DSP), matrix-related operations, polynomial operations, etc. That is, each Processing Element (*PE*) in the repetitive part of the systolic array is itself a linear system (we call it a *linear cell* in the following discussion). As an example, the inner product cell used in matrix multiplication is a linear system. Figure 1(a) shows the function of the inner product cell. Note that, to simplify the discussion, input and output buffers are not shown in this diagram. If we use L to denote this linear system, then the input vector and coefficient of this linear system L can be identified as follows: (b, c) is the input vector of L, and a is the coefficient of L (will be explained later). In the following discussion, L_a is used to represent a linear system L with coefficient (or coefficient vector) a. When we say that a is the coefficient (or coefficient vector, when a contains more than one value) of the linear system L, we mean that if a is kept fixed, and X_1 and X_2 are two input vectors (we use the term vector to specify that it may contain more than one value) to L, then

$$L_a(m_1X_1+m_2X_2)=m_1L_a(X_1)+m_2L_a(X_2),$$

where, m_1 and m_2 are any real numbers. (Notice that, in this paper, when we use the term *PE* we normally mean the *PE* in the repetitive part of the systolic array (instead of the irregular boundary *PE*).) By using this terminology, Figure 1(a) is shown to be a linear cell as follows:



$a' = a$
$b' = b$
$c' = c + a\,b$

$l' = l$
$m' = m + a\,l$
$n' = n + b\,l$

(a) (b)

Figure 1. Example of Linear cells.

For any real numbers m_1 and m_2, and input vectors (b_1, c_1) and (b_2, c_2):

$$\begin{aligned} &L_a(m_1(b_1, c_1)+m_2(b_2, c_2)) \\ &=L_a(m_1b_1+m_2b_2, m_1c_1+m_2c_2) \\ &=m_1c_1+m_2c_2+a(m_1b_1+m_2b_2) \\ &=m_1(c_1+ab_1)+m_2(c_2+ab_2) \\ &=m_1L_a(b_1, c_1)+m_2L_a(b_2, c_2). \end{aligned}$$

This shows that L_a is a linear system. Note that, given the functional description of a linear cell, there may be different ways to identify the input vector and the coefficient vector. For example, in Figure 1(a), we can either say that a is the coefficient and (b, c) is the input vector, or we can also say that b is the coefficient and (a, c) is the input vector. In other words, we can also show that L_b is a linear system with input vector (a, c). Another example of a linear cell is the *PE* used in the design of Infinite Impulse Response (IIR) filters (shown in Figure 1(b)) [15], where l is the coefficient of the linear cell, and (m, n, a, b) is the input vector. Although each linear system L can be of any form of a linear function, say, a differentiator, an analog linear system, etc., in this paper, we will focus on the situation where each L consists of only multipliers and adders, because this is the case with most existing numerical systolic arrays.

A property exists for linear systems (we call it the linear property):

Linear Property :

$$L_a(m_1X_1)+L_a(m_2X_2)+\cdots+L_a(m_nX_n)=L_a(\sum_{i=1}^{n}m_iX_i),$$

where, each X_i is an input vector of L_a (i.e., a linear system L with coefficient (or coefficient vector) a), and each m_i is a real number. Based on this linear property, a word level encoding scheme (we call it the *linear property encoding*) is proposed as follows:

Linear Property Encoding :

In this encoding scheme, each $L_a(m_iX_i)$ is calculated by a distinct *PE* and is accumulated to get $\sum_{i=1}^{n}L_a(m_iX_i)$ (i.e., by adding extra variables to pass through the systolic array to get this value). A different *PE* is used to compute $L_a(\sum_{i=1}^{n}m_iX_i)$. Finally, $\sum_{i=1}^{n}L_a(m_iX_i)$ is compared with $L_a(\sum_{i=1}^{n}m_iX_i)$. Any inconsistency will reveal that there exists a faulty *PE*. Since roundoff errors may occur during the computation, a relative small difference must be allowed in checking for equality, even though we will imply mathematical equality in equations and lemmas by using the equal sign. Notice that if each $L_a(m_iX_i)$ is not calculated by a distinct *PE*, it is possible that an error may be masked. For convenience of description, we call the requirement that each $L_a(m_iX_i)$ be calculated by a distinct *PE*, the *independence goal*. Note that during the normal operation, we set the value of each m_i to be 1. Different values of m_i will be used in the fault location phase discussed in Section VI.

In order to obtain fault tolerance, we are trying to encode data into the form of a linear property encoding, and trying to

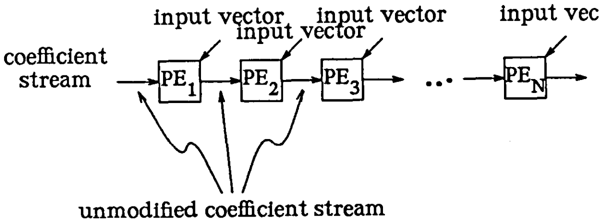


Figure 2. Diagram to show the coefficient stream passing through the array.

achieve the independence goal at the same time.

IV. CED TECHNIQUES FOR ONE-DIMENSIONAL SYSTOLIC ARRAYS

Another feature common to most existing numerical systolic arrays is that the coefficient stream for each *PE* (which is a linear cell) passes through the whole array without being modified. For example, in Figure 1(b), $l' = l$. This situation is shown in Figure 2, where the coefficient stream passes from the left end to the right end of the array without being modified. At each PE, there will be an input vector. Note that each value of the input vector can come from an adjacent *PE*, external input, or an accumulator inside the *PE*. For example, if each *PE* is the same as in Figure 1(b), then the l stream is the coefficient stream which passes through the array without being modified, and the input vector is (m, n, a, b). A CED scheme can be designed for those systolic arrays which satisfy the following two conditions (these are the two features common to most numerical systolic arrays):

Condition 1:

Each *PE* in the repetitive part of the systolic array is a linear cell.

Condition 2:

The coefficient stream passes through the array without being modified.

As was discussed in the previous section, there may be different ways to identify the input vector and coefficient vector. For example, if each *PE* in Figure 2 is like that of Figure 1(a), then we will choose the a (instead of b) to be the coefficient because it passes through the array without being modified.

The CED scheme using the linear property encoding is described as follows:

If v is a coefficient vector (we use the term *vector* to specify that it may contain more than one value) of the coefficient stream, then, when v reaches PE_i, PE_i becomes L_v (i.e., a linear system L with coefficient v). Suppose that v meets input vector X_1 at PE_1 to generate $L_v(X_1)$, meets input vector X_2 at PE_2 to generate $L_v(X_2)$, ..., meets input vector X_N at PE_N to generate $L_v(X_N)$. During the operation, each $L_v(X_i)$ is accumulated to generate $\sum_{i=1}^{N} L_v(X_i)$. By using a distinct (extra) PE_i to calculate $L_v(\sum_{i=1}^{N} X_i)$ and compare it with $\sum_{i=1}^{N} L_v(X_i)$, we can achieve CED through the linear property encoding (with each $m_i = 1$). Note that the independence goal is also achieved by using a distinct *PE* to calculate each $L_a(X_i)$.

Therefore, it is clear that we need an extra *PE* to calculate $L_v(\sum_{i=1}^{N} X_i)$, and, thus, we also need to pass the vector $\sum_{i=1}^{N} X_i$ to this extra *PE* for the calculation of $L_v(\sum_{i=1}^{N} X_i)$. Finally, we will also have to pass the vector $\sum_{i=1}^{N} L_v(X_i)$ to this extra *PE*, and compare it with $L_v(\sum_{i=1}^{N} X_i)$ to achieve the CED through the linear property encoding. All of these extra operations are performed by adding extra variables to pass along with the coefficient stream. In most cases, the number of extra variables can be reduced. by using the specific structure of a systolic algorithm.

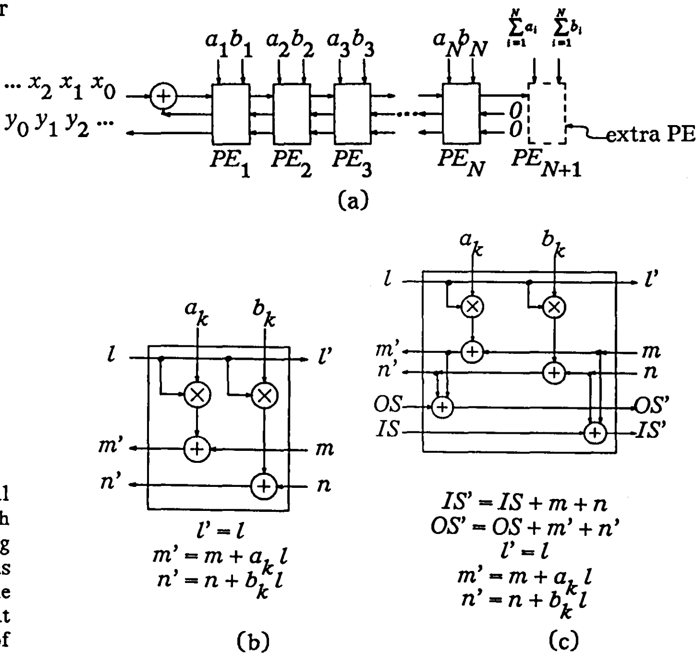


Figure 3(a). A systolic array for the IIR filter, (b). PE used in the original (non-CED) design, (c). PE used in the CED design.

Details are shown in the following examples:

EXAMPLE 1: Infinite Impulse Response (IIR) filter.

A systolic array for an Infinite Impulse Response (IIR) filter defined by the difference equation

$$y(k) = \sum_{m=1}^{N} x(k-m)b(m) + \sum_{m=1}^{N} y(k-m)a(m)$$

is shown in Figure 3(a), and the function of each *PE* is shown in Figure 3(b). This design is by S.Y. Kung in [15], which is derived from the signal flow graph in [18]. Note that in this design $b(0)$ is assumed to bo zero. However, if $b(0)$ is not zero, we can slightly modify the left boundary part (simply add a multiplier and an adder), and the *PE*s at the regular part of the array remain the same. Detailed procedures for transforming a signal flow graph into a systolic array are discussed in [15]. We add an extra *PE* (shown by the dashed box PE_{N+1}) at the right end of the array for the CED design. The procedure to incorporate CED into this IIR filter is described in the following three steps:

Step 1: Check the function of each *PE* to make sure that it is a linear cell. In this example, it is easy to show that this condition is true.

Step 2: Identify which variables form the coefficient vector or the input vector of each *PE*, and see if the coefficient stream passes through the array without being modified. In this example of the IIR filter, the coefficient for each *PE* is l, and the input vector is (m, n, a_k, b_k). Moreover, the coefficient stream (i.e., l stream) flows through the array from left to right without being modified.

Step 3: Add extra variables and the extra PE to the systolic array.

The extra variables and the extra PE are added as follows. Consider l_p which is a coefficient of the l coefficient stream. It is clear to see that l_p passes through the array without being modified. When l_p reaches PE_k, PE_k becomes L_{l_p} (a linear system L with coefficient l_p), and the input vector is (m_k, n_k, a_k, b_k), such that

$$(m_k', n_k') = L_{l_p}(m_k, n_k, a_k, b_k).$$

Here, m_k and n_k are used to represent the m-value and n-value which l_p meets at PE_k, and m_k' and n_k' are used to represent the m'-value and n'-value generated at PE_k (when l_p is at PE_k). Now, since that l_p meets (m_1, n_1, a_1, b_1) at PE_1 to generate (m_1', n_1'), meets (m_2, n_2, a_2, b_2) at PE_2 to generate (m_2', n_2'), ..., meets (m_N, n_N, a_N, b_N) at PE_N to generate (m_N', n_N'), the independence goal has been achieved. Then according to the linear property encoding discussed in Section III, we have to compare

$$L_{l_p}(m_1, n_1, a_1, b_1) + L_{l_p}(m_2, n_2, a_2, b_2) + \ldots + L_{l_p}(m_N, n_N, a_N, b_N)$$

with

$$L_p(\sum_{i=1}^{N}(m_i, n_i, a_i, b_i)).$$

Here,

$$L_{l_p}(m_1, n_1, a_1, b_1) + L_{l_p}(m_2, n_2, a_2, b_2) + \ldots + L_{l_p}(m_N, n_N, a_N, b_N)$$

$$= \sum_{i=1}^{N}(m_i', n_i')$$

$$= (\sum_{i=1}^{N} m_i', \sum_{i=1}^{N} n_i'). \qquad (1)$$

On the other hand, $L_{l_p}(\sum_{i=1}^{N}(m_i, n_i, a_i, b_i))$ is equal to

$$(\sum_{i=1}^{N} m_i + (\sum_{i=1}^{N} a_i) l_p, \sum_{i=1}^{N} n_i + (\sum_{i=1}^{N} b_i) l_p). \qquad (2)$$

Thus, if the linear property is satisfied, vector (1) will have to be equal to vector (2). However, it will be shown in this example that it is easy to implement the PE such that any faulty adder or multiplier will cause at most one component of vector (1) or vector (2) to become erroneous. Therefore, instead of comparing vector (1) with vector (2), we can simply compare

$$(\sum_{i=1}^{N} m_i' + \sum_{i=1}^{N} n_i') \qquad (3)$$

with

$$(\sum_{i=1}^{N} m_i + (\sum_{i=1}^{N} a_i) l_p + \sum_{i=1}^{N} n_i + (\sum_{i=1}^{N} b_i) l_p). \qquad (4)$$

That is, we can simply compare the sum of the two components of vector (1) (i.e., term (3)) with the sum of the two components of vector (2) (i.e., term (4)). Therefore, it is clear that we need an extra PE to calculate term (4), and, thus, we also need to pass $\sum_{i=1}^{N} m_i + \sum_{i=1}^{N} n_i$, $\sum_{i=1}^{N} a_i$, and $\sum_{i=1}^{N} b_i$ to this extra PE for the calculation of term (4). Since each a_i and each b_i are system parameters, which are precalculated and used for a long time without being modified, we can precalculate both $\sum_{i=1}^{N} a_i$ and $\sum_{i=1}^{N} b_i$, and directly apply them to the extra PE (as shown in Figure 3(a)). Thus, we only have to pass $\sum_{i=1}^{N} m_i + \sum_{i=1}^{N} n_i$ to the extra PE for the calculation of term (4). Finally, we will also have to pass term (3) to this extra PE and compare it with term (4).

Based on the above analysis, a CED scheme for the IIR filter can be designed. Along with the clocking of each coefficient of the l coefficient stream, two new variables are added :

$$IS' = IS + m + n \ ,$$

$$OS' = OS + m' + n' \ ,$$

where IS' and OS' are the updated values of IS and OS at each PE, and the IS and OS which enter PE_1 are both 0 (see Figure 3(c)). Finally, at the right end of the array, we use an extra PE (i.e., PE_{N+1} in Figure 3(a)) to compute

$$val\,1 = IS + l(\sum_{i=1}^{N} a_i + \sum_{i=1}^{N} b_i),$$

and compare $val\,1$ with OS. Any inconsistency will reveal that there exists a faulty PE (will be shown in Lemma 1). According to the implementation of IS and OS, it is easy to show that, under the fault free situation, at PE_{N+1} (the extra PE),

$$IS = \sum_{i=1}^{N}(m_i + n_i),$$

and

$$OS = \sum_{i=1}^{N}(m_i' + n_i').$$

Note that

$$OS = \sum_{i=1}^{N} m_i' + \sum_{i=1}^{N} n_i'.$$

This is the sum of the two components of vector (1) (i.e., term (3)).

Notice that, under the fault free situation,

$$val\,1 = IS + (\sum_{i=1}^{N} a_i + \sum_{i=1}^{N} b_i) l_p$$

$$= \sum_{i=1}^{N} m_i + (\sum_{i=1}^{N} a_i) l_p + \sum_{i=1}^{N} n_i + (\sum_{i=1}^{N} b_i) l_p .$$

This is the sum of the two components of vector (2) (i.e., term (4)).

Notice that since the linear property encoding is mainly used to detect the faults in the regular part of the array, the adder at the left end of the array (see Figure 3(a)) has to be duplicated or specially designed with some other technique to make sure that it functions correctly.

LEMMA 1: In the above design, by using different adders to compute IS' and OS', if there exists a faulty multiplier or adder in the array, then we have $val\,1 \neq OS$ at PE_{N+1}.

PROOF: To satisfy the linear property, vector (1) will have to be equal to vector (2). Thus, the sum of the two components of vector (1) will have to be equal to the sum of the two components of vector (2), i.e., at PE_{N+1}, OS will have to be equal to $val\,1$. If there is a faulty multiplier or adder in PE_i (where $1 \leqslant i \leqslant N$), then either the correct value of $L_{l_p}(m_i, n_i, a_i, b_i)$ will not equal (m_i', n_i') (i.e., OS' will be erroneous), or IS' will be erroneous. Thus, either IS' or OS' (but not both) at PE_i will become erroneous. If IS' at PE_i becomes erroneous, then $val\,1$ at PE_{N+1} (the extra PE) will become erroneous. If OS' at PE_i becomes erroneous, then OS at PE_{N+1} will be erroneous. If there is a faulty multiplier or adder in PE_{N+1}, then $val\,1$ will become erroneous. Therefore, under the assumption of single computation unit fault (described in Section II), any faulty adder or multiplier will change the value of either OS or $val\,1$ (but not both) at PE_{N+1}, and results in $OS \neq val\,1$. □

Notice that the same adder should not be used to calculate IS' and OS'; otherwise, it is possible that a faulty adder will cause both IS and OS to become erroneous. Since we assume that any faulty module may result in a random output, it is possible that the error will be masked (i.e., it is possible to cause both $val\,1$ and OS at PE_{N+1} to become erroneous). However, the probability of a fault being masked due to this reason is very small (will be discussed later).

In the above discussion, we only consider the faults coming from an adder or multiplier. However, it is very easy to extend the discussion to cover the faults from the busses or interconnection lines between two computation units. Consider the PE in Figures 3(c) and 4, which is redrawn in Figure 4 with each line being labeled with a number. Notice that in Figure 4, two extra adders are shown to calculate IS' and OS' (see Figure 3(c)). Actually, there are several ways to avoid the increase of the number of extra adders. For example, the extra adders can be shared between two adjacent PEs to cut the number of extra adders to half (by using some tricky interconnections). The other way is to use the original adders to calculate IS' and OS'. For the latter case, if a transient fault occurs during the operation of an addition, then only IS' or OS' (but not both) will be faulty, thus it will be detected at PE_{N+1}. If it is a permanent fault, and if the probability of a fault being masked at a faulty PE during a single linear property checking is p, then the probability of having an undetectable fault is p^t, where t is the total number of inputs (x's). The reason that we have this value is explained as follows. Since there are t inputs (x's), there will be t linear property encoding groups. Thus the fault will have to be masked for t times. An experiment has shown that p is about 0.02. This means that, for the case where $t = 50$, the probability of having an undetectable fault is about $p^t = 0.02^{50}$, which is almost equal to zero. This shows that even if we use the original adders to calculate IS' and OS', although it is theoretically possible to have an undetectable fault, the probability of having such an undetectable fault is almost equal to zero. Moreover, it will be shown later that we can timeshare the same bus to pass l, IS, and OS. However, to simplify the discussion, we simply assume that the two extra adders and two extra busses are used as shown in Figure 4. The results are applicable to other cases (i.e., the case with no extra busses, or the case with fewer extra adders).

It is easy to prove that any single fault in lines 1, 4, 6, 9, and 10 will make m' become erroneous, which in turn causes OS' to become erroneous. Any single fault in lines 2, 5, 7, 13, and 14 will make n' become erroneous, which in turn causes OS' to become erroneous. Any single fault in lines 16, 17, 20, and 21 will also cause OS' to become erroneous. Any single fault in lines 18, 19, 22, and 23 will cause IS' to become erroneous. Since all the above line faults will cause either IS' or OS' to become erroneous (but not both), they will be detected at PE_{N+1}. Now consider the case where line 11 becomes faulty. Assume that the value at line 11 is different from the correct value by d, i.e., its value is equal to (correct value + d). Then IS' will be increased by d, and OS' will be increased by d, and this fault will not be detected. Similarly, the fault at line 15 will not be detected, either (note that line 8 and line 12 are treated similarly at the next PE). Finally, if line 3 becomes faulty in PE_k, and its value is different from the correct value by d, i.e., its value is (correct value + d), then, at PE_{N+1}, $val\,1$ will be equal to

$$(\textit{correct value} + d\,(\sum_{i=1}^{N} a_i + \sum_{i=1}^{N} b_i)),$$

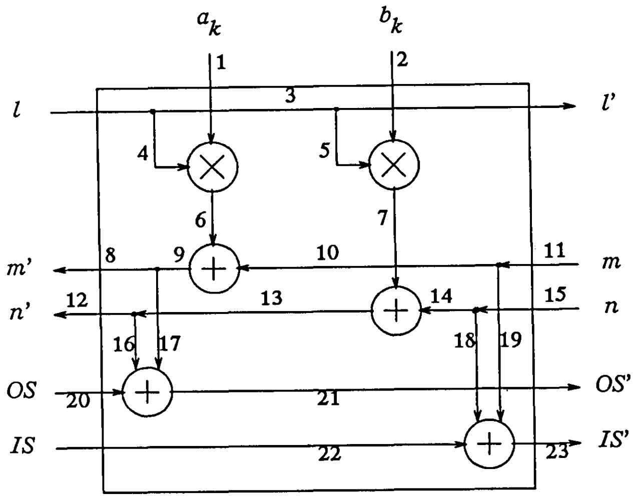


Figure 4. Circuit diagram to show the possible line faults of Figure 3(c).

while OS will be equal to

$$(\textit{correct value} + d\,(\sum_{i=k}^{N} a_i + \sum_{i=k}^{N} b_i)).$$

Therefore, it is clear to see that, normally, only when $k=1$ will a fault possibly cause an undetectable error. Thus, by specially designing the line 3 in PE_1 (such as adding a parity bit), we do not have to consider the line 3 fault in all other PEs. From this discussion, it can be seen that the only lines to which we have to apply some encoding scheme (e.g., parity bit) are lines 8, 11, 12, and 15. Faults in all other lines will be detected by the linear property encoding.

Notice that the faults in the input or output buffers (to simplify the discussion of linear property encoding, they are not shown in the figures) can be considered similarly as line faults, and are covered by the above discussion.

Since each coefficient l is multiplied by a_k and b_k and then m and n are added at each PE_k, while each IS and OS is only added once, and since l, IS, and OS are going in the same direction, we can use the same bus for the l stream to pass IS and OS by time-sharing the bus. In other words, the processing time for the value l at each PE is much longer than that for IS or OS, thus we can pass the value l and let it be processed first, and pass IS and OS subsequently by using the same bus as for the value l. Since the bus transmission time is usually very small compared with the processing time, the clock period will be almost the same. Therefore, the system throughput will also be the same. The two extra busses shown in Figure 4 can thus be avoided, that is, no extra busses are needed between two adjacent PEs. However, the adder at the left end of the systolic array has to be duplicated or specially designed with some other techniques to make sure that it functions correctly. Consider the case where each bus contains 24 bits, and a fixed point number system is used (where the layout area of an array multiplier is much larger than that of an adder), then the estimated hardware redundancy in terms of CMOS layout area for each PE is less than 5% and can be assumed to be negligible. In the case of floating point number systems, where the complexity of an adder is similar to that of an array multiplier, the redundancy ratio at each PE is about 15%. However, as is discussed previously, even if we use the original adders to calculate the IS' and OS', the probability of having an undetectable fault is almost equal to zero. Thus, we can avoid the redundancy at each PE if we use this approach. Moreover, the extra PE at the right end of the array is about half the complexity of the regular PE. Thus, the total hardware redundancy is about $\frac{1}{2N}$. Since the whole computation, including the extra PE, is pipelined, there is no time redundancy in terms of performance (i.e., the throughput is the same as before). However, if we consider the total job latency (between the time a job enters the IIR filter and the time this job leaves the IIR filter), the redundancy is less than $\frac{1}{N}$.

Another way to implement the IIR filter can be found in [19], where again each PE is a linear cell, and the coefficient stream (x, y) passes through the array without being modified. Thus a similar CED design approach can be applied.

In Figure 3(a), there are N PEs used in the IIR filter. However, in actual implementation, only $\frac{N}{2}$ PEs are needed. The reason is as follows. It is clear that for a bidirectional linear systolic array such as in Figure 3(a), there will be a zero value between any two consecutive input x's (for the sake of simplicity, we did not show it in Figure 3(a)), thus only one of two consecutive PEs will be active at any instant. Therefore a group of two PEs can share a common arithmetic unit (including multipliers and adders) without any penalty in the throughput rate. Another way to keep the processor utilization ratio equal to 1 (i.e., no idle processors during the operation) is to interleave two jobs. Similar arguments can be found in [13,14,15]. From the linear property encoding discussed previously, it should be clear that having a zero value between any two consecutive input x's or interleaving two jobs will not change our scheme at all. In other words, even with $\frac{N}{2}$ PEs (a zero value between any two x's) or N PEs (two jobs are interleaved), the linear property encoding will remain valid. That is, we still can fully utilize all the PEs without leaving any PE idle during the operation even for the case of bidirectional systolic arrays. Considering this, those schemes which duplicate the operations and compare the results of these operations to detect the faults (discussed in Section I) will certainly mean at least 100% time redundancy. □

EXAMPLE 2: Elimination of a subdiagonal of a band matrix [20].

A systolic array to eliminate a subdiagonal of a band matrix is shown in Figure 5(a). Here, we use N to represent the number of PEs in the repetitive part of the systolic array. For example, in Figure 5(a), $N = 4$ (the extra PE is not included).

The procedure to incorporate CED into this systolic array is described in the following three steps:

Step 1: Check the function (in the repetitive part of the array) of each PE to make sure that it is a linear cell. From the equation in Figure 5(b), it is easy to show that this condition is true.

Step 2: Identify which variables are part of the coefficient vector or input vector of the linear cell. In this example, the coefficient vector is (c, s), and the input vector is (x_i, y_i). The coefficient stream, i.e., the (c, s) stream, passes through the array from left to right without being modified.

Step 3: Add extra variables and an extra PE into the systolic array.

The way to add the extra variables and an extra PE is analyzed as follows. Consider a coefficient vector (c_p, s_p) which passes through the array from the left end to the right end of the array without being modified. When (c_p, s_p) reaches PE_i, PE_i becomes $L_{(c_p,s_p)}$ (i.e., a linear system L with coefficient vector (c_p, s_p)), and the input vector is (x_i, y_i). That is, (c_p, s_p) meets input vector (x_1, y_1) at PE_1 to generate (x_1', y_1'), meets input vector (x_2, y_2) at PE_2 to generate (x_2', y_2'), ..., meets input vector (x_N, y_N) at PE_N to generate (x_N', y_N'). Thus, the independence goal has been achieved. Here we use (x_i, y_i) to represent the input vector that (c_p, s_p) meets at PE_i. Thus, at each PE_i,

$$L_{(c_p,s_p)}(x_i, y_i) = (x_i', y_i').$$

According to the linear property encoding, we want to compare

$$L_{(c_p,s_p)}(x_1, y_1) + L_{(c_p,s_p)}(x_2, y_2) + \cdots + L_{(c_p,s_p)}(x_N, y_N)$$

with

$$L_{(c_p,s_p)}(\sum_{i=1}^{N}(x_i, y_i)).$$

That is, to compare

$$(\sum_{i=1}^{N} x_i', \sum_{i=1}^{N} y_i') \qquad (5)$$

with

$$L_{(c_p,s_p)}(\sum_{i=1}^{N}(x_i, y_i)).$$

Note that

$$L_{(c_p,s_p)}(\sum_{i=1}^{N}(x_i, y_i))$$

$$=(c_p \sum_{i=1}^{N} x_i + s_p \sum_{i=1}^{N} y_i, -s_p \sum_{i=1}^{N} x_i + c_p \sum_{i=1}^{N} y_i). \qquad (6)$$

However, it will be proved in this example that it is easy to implement the PE such that any faulty multiplier or adder will cause at most one component of vector (5) or vector (6) to become erroneous. Therefore, instead of comparing vector (5) with vector (6) we can simply compare

$$(\sum_{i=1}^{N} x_i' + \sum_{i=1}^{N} y_i') \qquad (7)$$

with

$$(c_p \sum_{i=1}^{N} x_i + s_p \sum_{i=1}^{N} y_i - s_p \sum_{i=1}^{N} x_i + c_p \sum_{i=1}^{N} y_i). \qquad (8)$$

That is, we can simply compare the sum of of the two components of vector (5) (i.e., term (7)) with the sum of the two components of vector (6) (i.e., term (8)). Therefore, it is clear that we need an extra PE to compute term (8), and, thus, we also need to pass $\sum_{i=1}^{N} x_i$ and $\sum_{i=1}^{N} y_i$ to this extra PE for the calculation of term (8). Finally we also need to pass term (7) to this extra PE and compare it with term (8).

Based on the above analysis, a CED scheme is proposed. Along with the propagation of each (c, s) to the right of the array, three new variables are added:

$$XI' = XI + x_i,$$

$$YI' = YI + y_i,$$

$$val\,1' = val\,1 + x_i' + y_i'.$$

This is shown in figure 5(c). Here, the XI, YI, and $val\,1$ which enter PE_1 are all 0. Finally, at the right end of the array, there is an extra PE (shown by a dashed box in Figure 5(a)). Inside this extra PE (at the right end of the array), we compute

$$val\,2 = c\ XI + s\ YI - s\ XI + c\ YI.$$

Then we compare $val\,2$ with $val\,1$, any inconsistency will reveal that there is a faulty PE (will be discussed in Lemma 2). From the implementation of XI, YI, and $val\,1$, we have, at PE_{N+1} (this is the extra PE, i.e., the dashed box in Figure 5(a)),

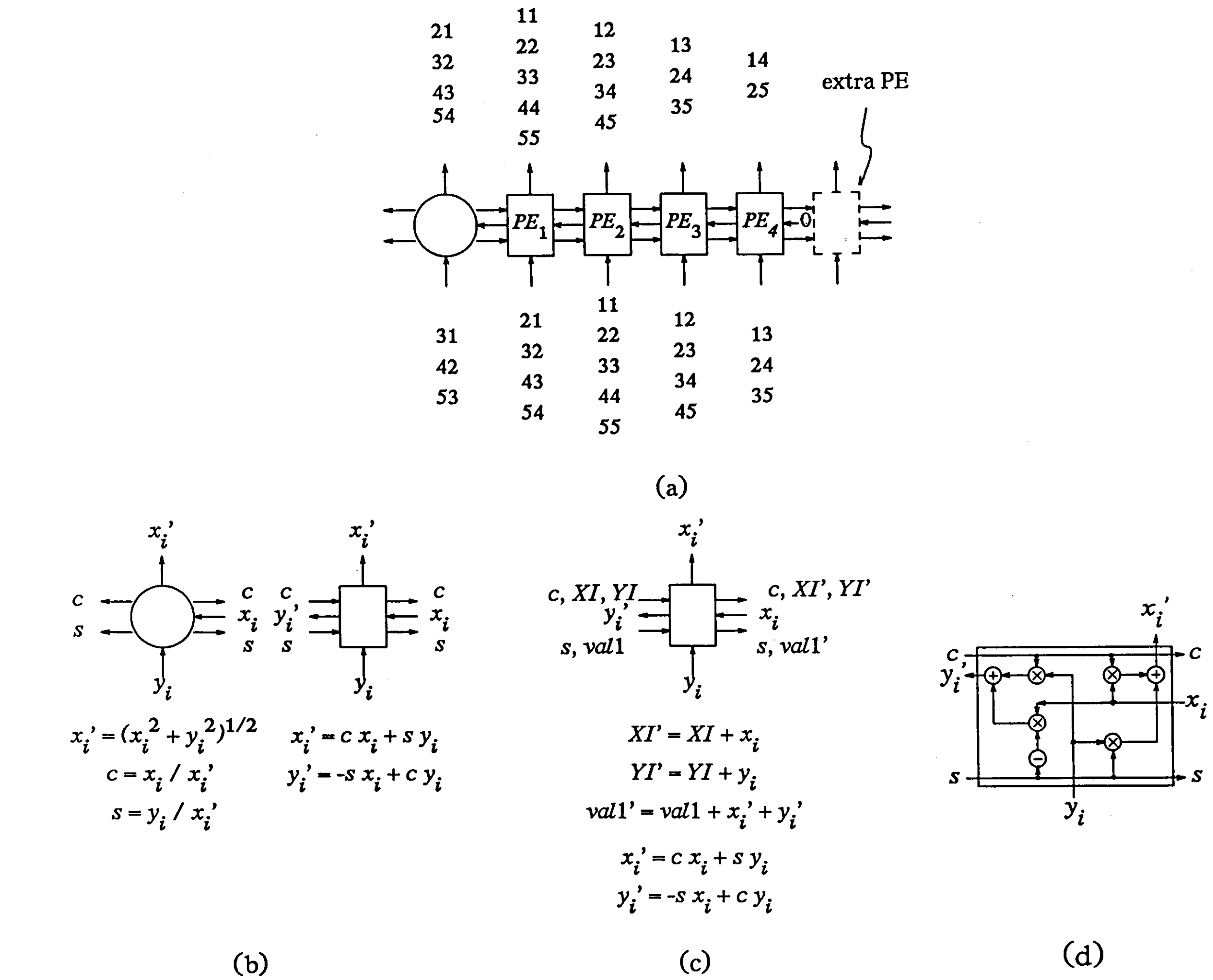


Figure 5(a). A systolic array to eliminate a subdiagonal of a band matrix. (b). PE used in the original (non-CED) design. (c). PE used in the CED design. (d). Circuit diagram for the regular PE in Figure 4(b).

$$XI = \sum_{i=1}^{N} x_i ,$$

$$YI = \sum_{i=1}^{N} y_i ,$$

$$val\,1 = \sum_{i=1}^{N} (x_i' + y_i').$$

It is easy to see that, under normal operation, *val* 1 will be equal to term (7), and *val* 2 will be equal to term (8). Note that the boundary *PE* at the left end of the array needs to be duplicated and outputs compared to make sure that it functions correctly.

LEMMA 2: In the above design, by using different adders to compute *XI'*, *YI'*, and *val* 1', if there is a faulty adder or multiplier in the array, then we have *val* 1≠*val* 2.

PROOF:

If the linear property is satisfied,

$$\sum_{i=1}^{N} x_i' + \sum_{i=1}^{N} y_i' = val\,1$$

will have to be equal to

$$(c_p \sum_{i=1}^{N} x_i + s_p \sum_{i=1}^{N} y_i - s_p \sum_{i=1}^{N} x_i + c_p \sum_{i=1}^{N} y_i) = val\,2.$$

That is, *val* 1 will have to be equal to *val* 2. By using the same argument as we did for IIR filter (due to the limited space, it will not be given here), it is easy to show that any faulty adder or multiplier will cause either *val* 1 or *val* 2 to become erroneous (but not both). Thus, we have *val* 1≠*val* 2. □

Note that the same adder should not be used to calculate *XI'*, *YI'*, and *val* 1'. Otherwise, it is possible that a faulty adder will cause both *val* 1 and *val* 2 (at PE_{N+1}) to become erroneous. Since we assume that a faulty unit will result in a random output, it is possible that the error will be masked. However, the possibility of a fault being masked due to the reason that we use the same adder to calculate *XI'*, *YI'*, and *val* 1 is very small as is discussed in the example of IIR filter. By using the same approach to analyze the communication lines as we did for the previous IIR filter example, we can show that only some lines need to be treated by a coding scheme (e.g., parity bit). All other line faults will be detected by the linear property encoding at PE_{N+1}. Due to limited space, this analysis will not be shown

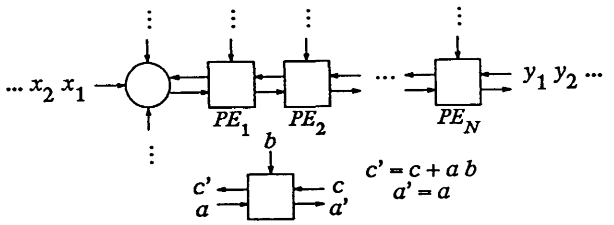


Figure 6. A systolic array to solve linear systems.

here.

Since each coefficient vector (c, s) will need four multiplications and two additions in each *PE*, while each *XI*, *YI*, and *val* 1 just does one addition in each *PE*, we can time-share the busses for the (c, s) stream to pass *XI*, *YI*, and *val* 1 as discussed in the example of IIR filter. However, the boundary *PE* (whose structure is different from all other *PE*s) must be duplicated or specially designed to make sure that it functions correctly. Generally speaking, the hardware redundancy is about $\frac{2}{N+1}$. The time redundancy in terms of performance is almost zero (i.e., the throughput is the same as before). However, if we consider the total job latency (between the time a job enters the systolic array and the time this job leaves the systolic array), the redundancy is less than $\frac{1}{N}$. □

Example 3: Solving linear systems [14].

The systolic array to solve this problem is shown in Figure 6. Here, each *PE* is again an inner product cell (which has been shown to be a linear cell), and the a stream is the coefficient stream flowing from the left end to the right end of the array without being modified. Therefore, the same scheme as in the previous examples can similarly be applied to this example. □

From the above examples, we know that the key point of the CED design is that each *PE* has to be a linear cell, and coefficient stream flows through the whole systolic array without being modified. The question arises as to whether this will become a serious restriction to the above design. The answer is no, since most all the existing systolic arrays for numerical computations do share this common feature. In other words, the design proposed in this section can be similarly applied to many other systolic algorithms, including those for convolution [21], Finite Impulse Response (FIR) filter [22], matrix multiplication [14], polynomial multiplication [23], polynomial division [23], polynomial evaluation [24], Kalman filter [25], etc.

V. CED FOR TWO-DIMENSIONAL SYSTOLIC ARRAYS

The key point in designing a CED for two-dimensional systolic arrays is to partition the array by rows or by columns, and to let each row or each column function like a one-dimensional systolic array which has been discussed in Section IV.

Example 4: LU decomposition [14-15].

In LU decomposition, a given matrix C is decomposed into $C = A*B$, where A is a lower- and B is an upper-triangular matrix. The recursions involved (from [15]) are

$$c_{ij}^{(k)} = c_{ij}^{(k-1)} - a_i^{(k)} * b_j^{(k)},$$

where

$$a_i^{(k)} = \frac{1}{c_{kk}^{(k)}} c_{ik}^{(k-1)},$$

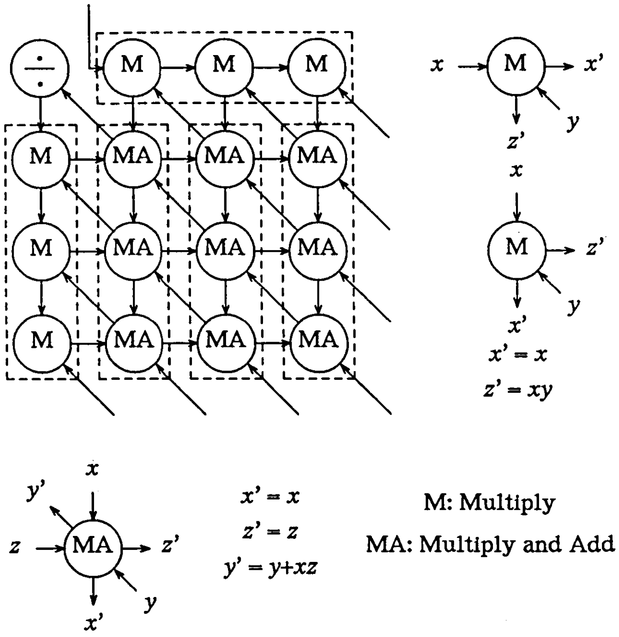


Figure 7. A systolic array to solve LU decomposition.

and

$$b_j^{(k)} = c_{kj}^{(k-1)},$$

for $k = 1, 2, \ldots, n$; $k \leqslant i \leqslant n$, $k \leqslant j \leqslant n$.

The systolic array for the above iteration is shown in Figure 7 [14-15]. The whole n by n array is divided into n columns and one row, each of which is identified by a dashed box, where each *PE* is an inner product cell (which has been shown to be a linear cell) and an unmodified coefficient stream flows through each one-dimensional systolic array (shown by the dashed box). Therefore, the scheme proposed in the previous section can be individually applied to each of these one-dimensional systolic arrays. Since the divider at the left top of the array does not contain the same structure as the others, we will have to duplicate it. The hardware redundancy is about $O(\frac{1}{n})$. The time redundancy in terms of the performance is almost zero (i.e., the throughput is the same as before). However, if we consider the job latency, the redundancy is less than $\frac{1}{n}$. □

Example 5: Orthogonal Triangularization [26].

A systolic array to perform orthogonal triangularization is shown in Figure 8 (x' is the updated value of x), where each dashed box denotes a one-dimensional systolic array, with each *PE* being a linear cell and with an unmodified coefficient stream flowing through each one-dimensional array. Thus, by using the same scheme as for one-dimensional systolic arrays, a systolic array with CED for orthogonal triangularization can be designed. Notice that all the boundary cells (which are not linear cells) represented by circles should be duplicated to make sure that failures in them will be detected. □

Other examples to which the scheme proposed in this section can similarly apply include QR decomposition [27], least square minimization [28-29], eigenvalue computation [30], two-dimensional convolution [31], etc.

Before finishing our discussion of CED, one thing should be especially noted; that is, even though all the above examples in Sections IV-V are confined to the discussion of systolic arrays,

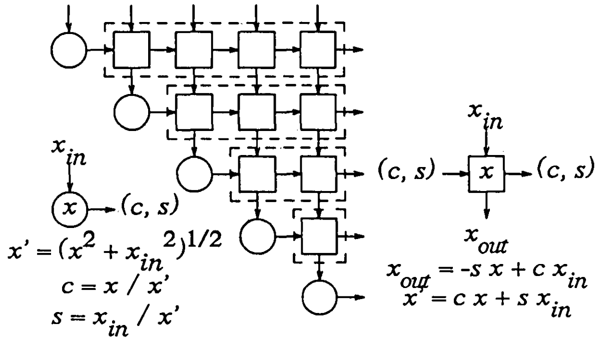


Figure 8. A systolic array for the orthogonal triangularization.

the same scheme can be similarly used to more general cases, for example, wavefront array processors [32], and general multiprocessor systems. The reason that we specifically emphasized systolic arrays is that it is quite difficult to design systolic arrays with CED capability due to their local interconnections, simple structure for each *PE*, and synchrony of the whole system.

VI. FAULT LOCATION AND RECONFIGURATION

A successful recovery from a hardware failure is essential to ensure continuous system operation. To achieve this i.e., to design a fault-tolerant system, three things have to be done: (1) Concurrent Error Detection (CED), (2) fault location, and (3) error correction and reconfiguration. Among these three steps, CED is the most important factor which affects the performance of the whole system, because it is necessary to continuously check for errors during the normal operation of the system. Therefore, it is always important to use as little time redundancy as possible during the CED phase. However, most existing schemes proposed [11-12][33] use duplicated operation as described in Section II. This will take too much time redundancy even under the fault free condition and may not be suitable for high performance special-purpose machines. Other schemes such as Triple Modular Redundancy (TMR) with voting [34] and Hybrid Redundancy [35] require several times the hardware of a single machine. However, the CED design proposed in the previous sections requires overheads of only about $O(\frac{1}{n})$ in hardware redundancy, and almost zero in time redundancy (the throughput is the same as before).

Fault location, correction, and reconfiguration in our design are performed only after an erroneous signal has been detected. Upon detecting the erroneous signal, we can choose one of the following two methods to recompute and obtain the correct result, while identifying the faulty unit.

Method 1: Recompute the operation by adding a weighted linear property encoding to pass along with the normal linear property encoding, i.e., without weighting. Note that during the normal CED phase, the linear property encoding is

$$L_a(X_1)+L_a(X_2)+\cdots+L_a(X_n)= L_a(\sum_{i=1}^{n} X_i),$$

where a is a coefficient vector flowing through the array, and X_i is an input vector to PE_i which meets a at PE_i during the propagation. That is, the weighting for each term is 1. Now, we have a weighted linear property encoding passing along with the normal encoding. In other words, the weighted linear property encoding

$$L_a(w_1X_1)+L_a(w_2X_2)+\cdots+L_a(w_n X_n)= L_a(\sum_{i=1}^{n} w_i X_i),$$

is also applied.

For the convenience of description, we again take the IIR filter as an example. At each PE_i (where $1 \leq i \leq N$), add four variables to pass along with the l coefficient stream, i.e.,

$$IS' = IS + m + n ,$$

$$OS' = OS + m' + n' ,$$

$$NIS' = NIS + w_i (m + n),$$

$$NOS' = NOS + w_i (m' + n').$$

At the extra *PE* (at the right end of the array) and its adjacent *PE*, we compute

$$IS + l(\sum_{i=1}^{N} a_i + \sum_{i=1}^{N} b_i) \tag{9}$$

and compare it with *OS*, and compute

$$NIS + l(\sum_{i=1}^{N} w_i a_i + \sum_{i=1}^{N} w_i b_i) \tag{10}$$

and compare it with *NOS*. Suppose PE_k $(1 \leq k \leq N)$ is faulty, the amount of difference detected with the first comparison is D_1, and the amount of difference for the second comparison is D_2. Then it can be shown that $D_2 = w_k D_1$. Thus, by dividing D_2 by D_1 we can get w_k. Therefore, if we choose $w_1 w_2 \cdots w_N$ to be N distinct values, then as soon as we get the value w_k, we can identify the faulty *PE*. However, the calculation of term (9), term (10), and the subsequent comparison should be performed by two different *PE*s (e.g., PE_{N+1} and PE_N), and the results are compared to make sure that the operation is correct. Thus, if PE_{N+1} becomes faulty, it will be detected. This finishes the fault location phase. Immediately following the fault location phase, we can simply bypass the faulty *PE* by using the schemes shown in [2-7][36-37] and continue the operation (of course, the last job should be recomputed).

Method 2: Each single operation is performed by two adjacent *PE*s and compared to each other to locate the fault. Then, continue the operation as described in Method 1.

Therefore, through either Method 1 or Method 2, the system can resume its normal operation, after bypassing the faulty unit.

VII. CONCLUSION

Low overhead fault-tolerant schemes have been proposed, which are applicable to most existing numerical systolic arrays. Partitioned by rows or by columns, two-dimensional arrays are similarly treated as one-dimensional arrays.

By using the schemes proposed in this paper, the function of each *PE* remains unchanged (except for the added variables) after becoming a fault-tolerant systolic array. Moreover, the structure of the systolic array is almost the same as before. All the designs proposed in this paper are based on optimal systolic arrays; therefore, the fault-tolerant systolic arrays are also optimal. All the schemes proposed in this paper can be readily extended to wavefront array processors and general multiprocessor systems.

REFERENCES

[1] H.T. Kung, "On the Implementation and Use of Systolic Array Processors," *Proceedings of ICCD*, pp. 370-373, Oct. 1983.

[2] D.S. Fussel and P.J. Varman, "Designing Systolic Algorithm for Fault Tolerance," *Proceedings of ICCD*, pp. 616-621, Oct. 1984.

[3] H.T. Kung and M.S. Lam, "Fault-Tolerance and Two-level Pipelining," *MIT Conference on Advanced Research in VLSI*, pp. 74-83, Jan. 1984.

[4] M. Sami and R. Stefanelli, "Reconfigurable Architectures of VLSI Processing Arrays," *Proceedings of the IEEE Special Issue on Fault Tolerance in VLSI*, vol. 74, no. 5, pp. 712-722, May 1986.

[5] A.L. Rosenberg, "The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors," *IEEE Trans. on Computers*, vol. C-32, no. 10, pp. 902-910, Oct. 1983.

[6] A. Khurshid and P.D. Fisher, "Design of a Reconfigurable Systolic Array Using LSSD Techniques," *Proceedings of ICCD*, pp. 171-175, Oct. 1984.

[7] I. Koren, "A Reconfigurable and Fault-Tolerant VLSI Multiprocessor Arrays," *Proceedings of the 8th Annual Symposium on Computer Architecture*, pp. 425-442, May 1981.

[8] K.H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Computers*, vol. C-33, no. 6, pp. 518-528, June 1984.

[9] J.Y. Jou and J.A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of the IEEE Special Issue on Fault Tolerance in VLSI*, vol. 74, no 5, pp. 732-741, May 1986.

[10] V.S. Iyengar, "Concurrent Testing of Microprogrammed Control Units and Systolic Arrays," *Ph.D. dissertation*, Univ. of Minnesota, June, 1983.

[11] Y.H. Choi, S.H. Han, and M. Malek, "Fault Diagnosis of Reconfigurable Systolic Arrays," *Proceedings of ICCD*, pp. 451-455, Oct. 1984.

[12] J.H. Kim, "A Fault-Tolerant Systolic Array Design Using TMR Method," *Proceedings of ICCD*, pp. 769-773, Oct. 1985.

[13] C.E. Leiserson, "Systolic and Semisystolic Design," *Proceedings of ICCD*, pp. 627-632, Oct. 1983.

[14] C. Mead and L. Conway, "*Introduction to VLSI Systems*," Reading, MA: Addison-Wesley, 1980.

[15] S.Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors," *Proceedings of the IEEE*, vol. 72, no. 7, pp. 867-884, July 1984.

[16] R.W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technology Journal*, vol. 29, no. 1, pp.147-160, Jan. 1950.

[17] J.J. Shedletsky, "Error Correction by Alternate-Data Retry," *IEEE Trans. on Computers*, vol. C-27, pp. 106-114, Feb. 1978.

[18] A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, inc., Englewood Cliffs, NJ, 1975, pp. 151.

[19] H.T. Kung, "Special-purpose devices for signal and image processing: an opportunity in very large scale integration (VLSI), *Proceedings of SPIE, Real-Time Signal Processing*, vol. 241, pp. 76-84 1980.

[20] I. Ipsen, "Singular Value Decomposition with Systolic Arrays," *Proceedings of SPIE, Real Time Signal Processing VII*, vol. 495, pp. 13-21, 1984.

[21] H.T. Kung, "Why Systolic Architecture?", *IEEE Computer, pp. 37-46, Jan. 1982.*

[22] D.E. Heller, "Decomposition of Recursive Filter for Linear Systolic Arrays;" *SPIE proceedings, Real Time Signal Processing VI, pp.55-59, 1983.*

[23] H.T. Kung, "Use of VLSI in Algebraic Computation: Some Suggestions," *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, pp.218-222, 1981.

[24] M.S. Lam and J. Mostow, "A Transformational Model of VLSI Systolic Design," *IEEE Computer*, pp. 42-52, Feb. 1985.

[25] S.Y. Kung, H.J. Whitehouse, and T. Kailath, Editors, "*VLSI and Modern Signal Processing*," Reading, NJ: Prentice Hall, pp. 375-388, 1985.

[26] W.M. Gentleman and H.T. Kung, "Matrix Triangularization by Systolic Arrays," *SPIE proceedings, Real Time Signal Processing*, vol. 298, pp. 19-26, 1981.

[27] D.E. Heller and I.C.F. Ipsen, "Systolic Networks for Orthogonal Equivalence Transformations and Their Applications," *Conference on Advanced Research in VLSI, MIT*, pp. 113-122, 1982.

[28] J.G. McWhirter, "Recursive Least-Squares Minimization Using a Systolic Array," *SPIE Proceedings, Real Time Signal Processing VI*, pp. 105-112, 1983.

[29] C.R. Ward, A.J. Robson, P.J. Hargrave, and J.G. McWhirter, "The Application of a Systolic Least Squares Processing Array to Adaptive Beamforming," *Proceedings of ASSP Conf.*, vol. 2, pp. 34A.3.1-34A.3.4, 1984.

[30] R. Schreiber, "Systolic Arrays for Eigenvalue Computation," *Proceedings of SPIE, Real Time Signal Processing V*, vol. 341, pp. 27-34, 1982.

[31] H.T. Kung and S.W. Song, "A Systolic 2-D Convolution Chip," *Tech. Report, CMU-CS-81-110*, Carnegie-Mellon Univ., Computer Science Dept., Mar. 1981.

[32] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, and D.V. Bhaskar Rao, "Wavefront Array Processor: Language, Architecture, and applications," *IEEE Trans. on Computers*, vol. C-31, no. 11, pp. 1054-1066, Nov. 1982.

[33] S. Laha, and J.H. Patel, "Error Correction in Arithmetic Operations Using Time Redundancy," *Proceedings of 13th Annual International Symp. Fault-Tolerant Computing*, pp. 298-305, June 1983.

[34] J. Von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata studies*, no. 34, pp. 43-99, Princeton U. Press, Princeton, NJ.

[35] F.P. Mathur and A. Avizienis, "Reliability Analysis and Architecture of a Hybrid-Redundancy Digital System: Generalized Triple Modular Redundancy with Repair," *IEEE Computer Group Workshop on "Reliability and Maintainability of Computing Systems,"* Lake of Ozarks, Missouri, Oct. 1969.

[36] S.Y. Kung, D.D. Souza, and J.T. Johl, "On Fault-tolerance in Array Processing," *Proceedings of ICCD*, pp. 764-768, Oct. 1985.

[37] J.A.B. Fortes and C.S. Raghavendra, "Dynamically Reconfigurable Fault-Tolerant Array Processors," *Proceedings of 14th Int'l Symp. on Fault-Tolerant Computing*, pp. 386-392, 1984.

The Design and Development of a Very High Speed System Bus - The Encore Multimax Nanobus

David J. Schanin

Infinity Systems, Inc.
5 July Road
Sudbury, Massachusetts 01776
(617) 443-5104

Abstract

This paper presents the design analysis and tradeoffs that drove the design of a very high speed backplane interconnect in a new, tightly-coupled multiprocessor system. The involvement of cache and memory subsystem designs with the interconnect architecture are explored. The various tradeoffs in arbitration techniques and bus protocols are analyzed. In addition, a discussion of atomic operations in a multiprocessor environment are covered. Finally, an innovative technique for interrupt vector handling in a multiprocessor system is described.

Introduction

Through improving technology and innovative CPU architectures, the performance capability of modern computing systems is growing at a rapid pace. As the CPU's computing capabilities increase, the rest of the system's requirements tend to scale up with it. The physical and virtual memory capacity, the I/O bandwidth, and the secondary storage traffic generated all increase.

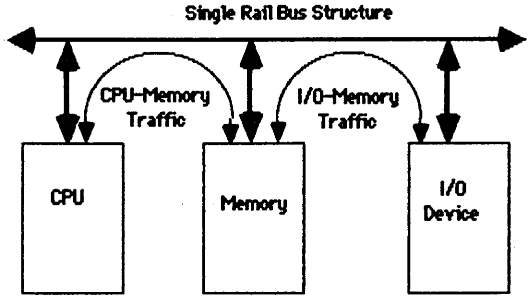


Figure 1 - The Von Neumann Bottleneck

These developments aggravate the traditional Von Neumann bottleneck between the Central Processing Unit (CPU) and the program/data memory subsystem, illustrated in figure 1. Current trends in computer architecture are towards multiprocessors, and in particular tightly-coupled multiprocessors, where many CPUs share a common memory subsystem. Tightly-coupled systems increase the pressure on the bottleneck by creating even higher memory bandwidth requirements, as illustrated in figure 2. In addition, a common memory computer system which contains more than one CPU complicates memory contention logic because there is now a need to arbitrate fairly among the many CPUs, as well as among the I/O devices accessing memory.

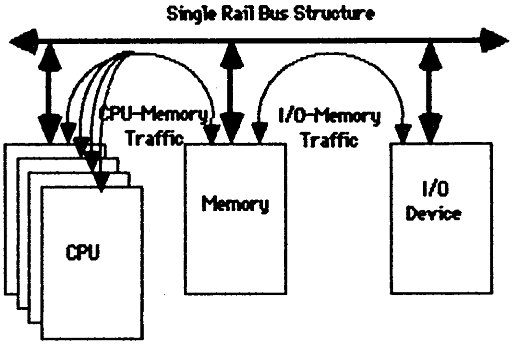


Figure 2 - The Von Neumann Bottleneck in an Mp System

The traditional method for addressing this bottleneck involves the design of a memory subsystem, an interconnect, and module interfaces (CPU and I/O) that will support the theoretical bandwidth required. In the design of the Nanobus,

CH2345-7/86/0000/0410$01.00 © 1986 IEEE

the interconnect used in the Encore Multimax (a tightly-coupled multiprocessor), a different approach was taken. The Nanobus interconnect and the memory subsystem were actually designed as a single, integrated unit.

The Multimax - An Architectural Overview

The Multimax is a classic tightly-coupled, Multiple Instruction Multiple Data (MIMD) computer system. Figure 3 illustrates its architecture.

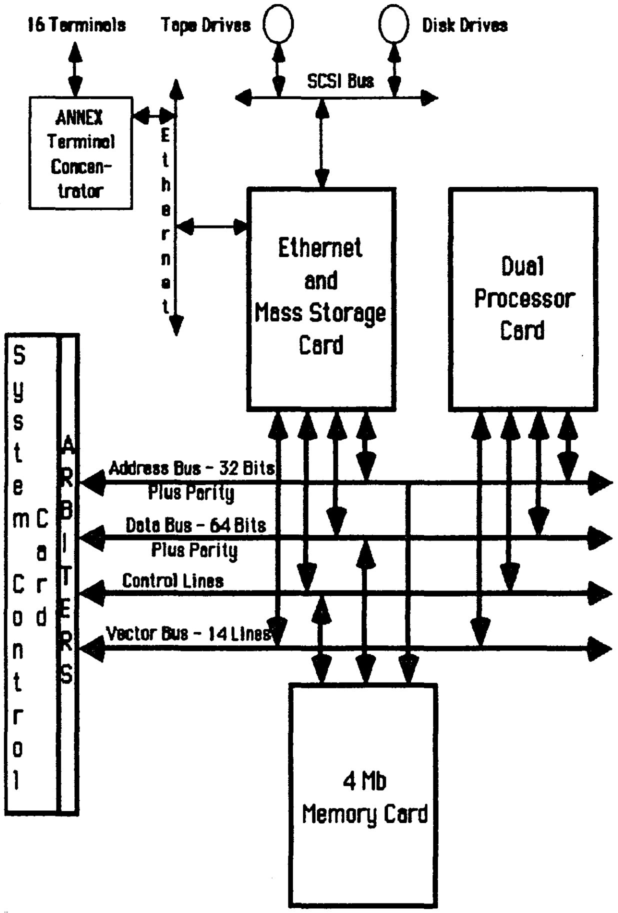


Figure 3 - The Multimax System

To reduce project risk and enhance time to market, the design philosophies of the Multimax were simplicity of hardware design and ease of program development; features which created programing restrictions were avoided whenever possible. Some examples of hardware features implemented with this goal of simplicity are:

- A bus design that is non-multiplexed and very wide, rather than employing complex (and potentially error prone) protocols, and multiplexing addresses and data.
- Complete parity protection on every operation and memory element in the system, to help detect any design problems.
- Identical bus interfaces on all modules in the system, so that only one interface type needed to be designed and debugged.

Some system features implemented to provide the simplest software interface possible are:

- Caches which were made software transparent, so cache concurrency was always handled by the hardware.
- Semaphores which were implemented to allow their placement anywhere in physical memory, not to restrict them to certain special regions or to restrict their number.
- Hardware which was designed to allow load leveling and directing of interrupt vectors to the "most interruptible" CPU.

All Multimax mass storage operations are handled through an intelligent Ethernet/Mass Storage Card, which needs only high level commands. All terminal traffic interfaces to the Multimax from an Ethernet connection through an intelligent terminal concentrator box called the Annex. Each Annex interfaces with up to sixteen simple RS-232 type terminals.

Memory Subsystem Design

In most new supermini computers, the memory subsystem usually consists of a single memory controller with multiple attached memory array cards. The former contains all of the bus interface and control logic, and the latter contains only arrays of memory chips with some driver/receivers. The problem with such an architecture lies in the contention that occurs for the single controller. The VAX family, from the 11/750 on up, uses buffering and FIFOs to reduce the busy state of the controller. However, this technique is valuable only for a system where one or two CPUs are generating traffic. In a large, tightly-coupled multiprocessor, these techniques would prove inadequate for handling the high volume of memory traffic generated.

Cycle Number	1	2	3	4	5	6	7
CPU	Request and Grant Cycle	Transfer Address	If Write Operation, Transfer Write Data		Receive Accept or Reject Response	Retry Request and Grant Cycle	Transfer Address
MEMORY			Decode Address	Send Accept or Reject Response	Return Read Data		

Figure 4 - CPU and Memory Data Transfer States

Examining the problem yields the following analysis: Since the lowest indivisible level in the memory subsystem is really one row of RAM chips, it is ideally at that level only that one would want contention to occur. Therefore, the ideal memory subsystem would have one controller and bus interface per row of memory chips. This would allow the maximum overlapping of memory operations since each bank could cycle independently. Unfortunately, this is also clearly the most expensive solution. In compromising between these two extremes, by grouping just a few rows of memory chips on each controller, one can achieve a reasonable tradeoffs between cost and contention. The current Multimax memory subsystem groups four rows of RAM chips per controller, and builds a memory subsystem out of multiple cards, each of which contains its own controller. Note that this very much mirrors the traditional mini-computer memory architecture, where each memory card contained its own controller.

The design of the memory subsystem had to insure that the memory subsystem and the Nanobus would function as a single, synchronous unit. The usual synchronizer delays, bus interface protocol overhead, and general mismatch in performance between the RAM controllers and the bus had to be avoided. The solution was to have the memory cards actually derive all of their internal clocks from the Nanobus itself, and they therefore run completely synchronous to the bus. This not only eliminates any lost access time due to synchronizer delays (which would be present if the memory was clocked asynchronously to the bus), but also makes memory access time deterministic. As a result, the bus protocol on the Nanobus was designed specifically around the ability of a memory controller using 150 nsec access time RAMs to receive addresses, validate them for both range and parity errors, and respond. As can be seen in figure 4, the bus protocol allows for the transfer of an address in cycle 2 and, if a write operation is being performed, the write data in cycle 3. Both are checked for range and parity errors, and the internal state of the memory card is checked to see if the requested bank is available. If all these tests are passed, an accepted signal is returned in cycle 4. If a read operation has been requested, and a refresh or a bank busy condition has not occurred, the data will be ready for return in cycle 5. As can be seen from the figure, memories can cycle every four bus cycles. The Multimax supports data transfers at the full bus bandwidth by supporting four way interleaved memories.

By designing the bus protocol to match the memories, and by running the memories synchronously to the bus, the memory system functions as a direct extension of the bus. This yields the optimum performance interface between the memory subsystem and a CPU or I/O card.

Nanobus Structure

The Multimax's Nanobus is designed to support eight memory and twelve requester (CPU or I/O) modules for a total of twenty interfaces. The target bandwidth for the Nanobus was 100 Mb per second to allow for a minimum of a five year product life, and to support very high bandwidth custom modules, such as high speed video interfaces, array processors, and custom compute engines.

It was intended that 100 Mb per second be the actual data transfer rate through the system, not just a theoretical protocol limit, i.e. using zero access time memories. To insure support for such a transfer rate, a data path width of eight bytes (64 bits) would have to be clocked at 12.5 Mhz. However, maintaining the 100Mb data transfer rate on a single bus would not allow time slots for multiplexing addresses and data, so separate address and data buses are used. The physical address space of the Multimax is four gigabytes (2^{32}), so the Address Bus contains 32 bits of address. Even interrupt vectors do not contend for the Data Bus - there is a separate 10 bit wide bus dedicated to the transfer of vectors.

The technologies that made the Nanobus speeds possible are the new Advanced Schottky and Fast technologies. Bidirectional bus data signals are driven by 74AS652, and most other tri-state bus signals are driven by 74AS821, 841. Bus control lines are driven by 74AS821, except for open collector signals which are driven by 74AS756.

The Nanobus was designed as a pended bus, so that after a read address is transferred to the memory, the bus is released. This makes the bus available for other address/data transfers while the requester is awaiting the return of the requested read data from memory. To insure guaranteed delivery of addresses and write data, an accept/reject handshake is built in to the Nanobus so immediate confirmation of delivery is received. A rejected cycle is then automatically retried by the bus interface. However, a pended bus adds a new dimension to the traditional address/data bus. Addresses transferred to memory are typically used to identify which memory location is being read or written. But when the read data is ready to be returned by the memory to the requester, the requester has already released the bus. Therefore the data must be "addressed" back to its requester. To support this function, a second set of addresses, that of one of twelve possible requestors, is also implemented on the Nanobus. This address is referred to as the "Requester I.D.". This feature has some innovative implications when dealing with caches, as discussed later.

All address, data and protocol paths are parity protected, to insure reliability. A module receiving a parity error can reject the bus cycle, and the bus interface will retry the operation. Retrys on errors are performed automatically by the bus state machines, and are therefore transparent to the rest of the system. Although all address and data lines are tri-state, some critical control lines are open collector, including the retry signal. This is important to note because it allows any module that detects a parity or protocol error on the bus to assert signals which cause the errant cycle to be re-tried. A system with distributed caches requires this capability. Since all the distributed caches monitor the system bus for addresses of data that is locally cached, a cache must be able to cause a cycle to be retried if it detects an address parity error and therefore cannot validate the address of the transfer in process.

Distributed Cache Support

The Nanobus was designed specifically to support tightly-coupled multiprocessing. Such a system must support cache memories at each processor module to achieve acceptable performance. The Multimax integrated support for a multiprocessor's distributed caches into the system bus itself, to ensure that cache concurrency can be simply and absolutely maintained. The first Multimax system CPU modules were built using caches employing a standard write-through algorithm. A write-through cache forces all write operations to pass through the cache to main memory, regardless of whether or not the data is cached. The data in a write-through cache may be updated or invalidated, and the cache may or may not be allocated. In the Multimax system, write-through data is updated on a cache hit, and the cache is allocated on a longword write. This algorithm produced the highest cache hit ratio. Regardless, a write-through cache's advantage is that it is the simplest cache algorithm to implement from a hardware perspective. However, this scheme is less than optimal. It incurs performance penalties at the CPU level (because all write operations require bus and main memory access) and on a system level (Because all CPU write traffic appears on the system bus and at the memories).

Maintaining data consistency across the distributed caches and main memory is an absolute requirement. In order to ensure consistency, each write-through cache in the system always monitors the Nanobus for bus write operations. Write operations to main memory can create cache consistency problems since they modify the contents of main memory without explicitly modifying the potentially distributed cached copies of a location. When a main memory write is monitored to a location that is cached, the monitoring cache must update or invalidate its local copy of the data. In the Multimax design, the decision was made to invalidate the local copy, because this minimized the interference with the local CPU. Updating instead of invalidating would require the bus interface to obtain access to the CPU cache at the expense of the CPU. Simply invalidating the location requires access only to the Valid bit of the cache tag store, a feat which is much easier to accomplish because the Valid bit is just one bit. The net result of the Multimax implementation of write-through is that at any point in time there may be many copies of a memory location distributed among the many caches in the system. However, once a modification to that memory location occurs, all but two copies of the data anywhere in the system will be invalidated - the copy in main memory, and possibly a copy in the cache of the CPU performing the write.

The Nanobus also supports caches implementing a non-write through, or write-deferred algorithm. A write-deferred algorithm allows write operations to be performed only in the cache. Such an algorithm enhances the CPU and system level performance by filtering write operations from the system bus and main memory. However, a write-deferred algorithm in a multiprocessing system aggravates the problem of data consistency across the system. When a write-deferred cache allows a write operation to take place only in the cache, the cached content of the memory location now no longer agrees with the main memory location, or with any other copies of this location in other CPU caches.

This data consistency problem can be solved through implementing a concept of data ownership. One method of implementing data ownership was discussed by Dr. Goodman, and one commercial application of data ownership was implemented by the Synapse Computer Corporation in their N+1 computer system. Past solutions to the data ownership problem have been usually been complex. As an example, the Synapse N+1's implementation required extensive additional hardware. The concept of ownership was implemented with extra data bits in memory and in the caches.

The scheme planned for the Multimax involves using a "Public Read" to satisfy all cache misses. When a write operation is attempted, a "Private Read" occurs to main memory. All other caches monitoring a "Private Read" will invalidate their local copies. By performing a "Private Read", the non-write through cache now "owns" the data. Ownership is maintained through a familiar mechanism (as seen in write-through caches) - monitoring the bus for addresses that have been privately read. If a bus read is attempted by any other device in the system on a privately owned location, the monitoring cache signals the memory to ignore the pending read request.

The write-deferred cache will now access the locally modified (and therefore owned), cached data, and write it back to main memory, updating the system and ensuring consistency. On write-back to main memory, the address on the Address Bus always specifies the memory location that is being updated. During a write operation which does not transfer data ownership, the "Requester I.D.", described previously, that is supplied with the write data, is not set to match any card in the system. The data is therefore transferred only to main memory. However, during a write operation which does transfers data ownership, the "Requester I.D." is set to match the I.D. of the card whose pended read operation started the whole write back process. In this manner, the data is simultaneously, in one bus operation, returned to both the memory and the new "owner", another cache.

Bus Access Arbitration

The Nanobus consists of three primary buses - the Address Bus, the Data Bus, and the Vector Bus. Access arbitration for each of these three buses bears examining, as each presents a unique set of problems.

Access to the Address Bus by CPUs and I/O devices determines the basic "fairness" of sharing the bottlenecked resource - the memory - because data transfers to or from the memory are always stimulated by addresses. To insure true fairness in Address Bus access, the Address Bus arbiter implements a dynamically modified round robin scheme.

Once a module has been granted access to the Nanobus Address Bus, it then becomes the lowest priority module in the round robin. However, the Nanobus protocol specifies that four bus clock ticks after an address is transferred, notification of its acceptance or rejection is returned by the addressed memory. At this time, if the address has been rejected, the Address Bus arbiter dynamically re-assigns the rejected module as the highest priority module at the same time the rejected module retries its request to the Nanobus during the next bus cycle. Therefore, the module's access to the bus is guaranteed.

Once a module has been retried at least once, its asserts a signal labeled "Priority". This halts new entries to the bus request round robin until all outstanding requests have been satisfied. The priority signal ensures that modules being retried are never caught in a potential lock out situation where it is continually retried and never gets access to the memory due to other modules interference between its retrys.

The implementation of the address arbiter is also unique. It is a combination of both a centralized and a distributed arbiter. The centralized section of the arbiter decides which module is to be granted the bus according to the retry rules previously described. The distributed part of the arbiter, which exists on each module, determines when and if a module will retry its request, or whether to holdoff due to the priority signal being asserted, as well as other conditions. This split arbiter allows the advantages of both centralized and distributed arbitration to be combined. The centralized arbiter avoids the need for every module to know the external state of every other module in the system, i.e. whether any other module in the system was requesting the bus. This would have required a large number of signal lines and redundant hardware on each module. The distributed arbiter allows local decisions to be made based on the state of the local module without the need to communicate this information around the system. The resulting implementation is both fast and simple.

The Vector Bus arbiter implements the same dynamically modified round robin algorithm as the Address Bus arbiter. However, since there is no concept of a "busy memory" on the Vector Bus, the retry mechanism is meaningless and not implemented. Once granted the Vector Bus, a transfer occurs only once.

The Data Bus arbiter is much simpler than the Address Bus arbiter. The Data Bus is used by memories wanting to return read data, or by CPU's or I/O devices sending write data. Memories must request access to the Data Bus through the Data Bus arbiter. This arbiter is very simple and grants access based on fixed priority due to a module's physical position in the backplane. The requests to the Data Bus, because they are stimulated by the "fair" addresses, tend to be fair themselves, and therefore a truly fair arbiter for the Data Bus is unnecessary.

However, there are some perturbations to the fairness of the Data Bus. Nanobus protocol requires that a write address on the Address Bus be followed during the very next bus cycle by the write data on the Data Bus. It is therefore possible that a memory might be granted access to the Data Bus to return read data just as a CPU or I/O device is transferring write data on the Data Bus. Clashes on the Data Bus are avoided by having the memories monitor the type of address that is transferred on the Address Bus. If a write address is detected, a memory will abort an attempt to return read data during the next Data Bus cycle, even if it has been granted the Data Bus by the Data Bus arbiter, and thereby avoiding a clash. Another perturbation to the fairness of the Data Bus happens because of refresh conflicts at the memories, which may delay the return of read data from some memory modules.

These perturbations to fairness can result in pathological conditions whereby a low priority module (due only to physical location) can be locked out from returning its read data for an extended period of time. To overcome this case, an UNJAM signal is available to all memories. When a memory has not been able to return requested read data for several bus clock cycles following its request, it asserts UNJAM until it is able to return its data. Asserting UNJAM causes the Address Bus arbiter to suspend granting the Address Bus (this eliminates write data traffic), and all memories not already requesting the Data Bus hold off posting new requests (eliminating more read data traffic). These changes to the arbiters allow the Data Bus bottleneck to unjam and the system to clear itself.

Atomic Operations

Due to the MIMD architecture of the Multimax, all system resources are shared and can be dynamically re-assigned by the operating system. Therefore, ownership identification of every system resource must always be clear and unambiguous, and arbitration for a shared resource must always have one and only one winner. Traditional methods for ensuring only one winner is through the use of indivisible flags or software locks in main memory. An indivisible, or atomic operation on a flag location typically requires a test-and-set function.

The atomic test-and-set function allows the CPU to read the state of a lock and then set it, while guaranteeing that there are no intervening operations by other devices between the read and the set, or write, operation. If such an operation by another device were to occur, the lock could be corrupted.

In the past, there have been several basic approaches to implementing atomic operations. The simplest has been to lock the entire system, bus and memory, between the read and the write of the atomic operation. While this may be adequate on a non-pended bus, or in a system of only moderate performance, this was neither possible on the Nanobus due to its pended design, nor desired because of the performance implications of locking the entire system bus for a period of time. Other typical solutions involve creating special hardware, either in the memory system or the I/O space, that either allows artificially partitioning memory into blocks and locking various blocks of memory for the atomic operation (Synapse N+1), or allows the creation of "special" dedicated semaphores that are not in memory (Sequent's Balance). None of these solutions were deemed acceptable because they imposed software restrictions on the architecture of the operating system.

To create semaphores in the Multimax, Nanobus memories incorporate a design feature to implement atomic test and set operations <u>at the memory chip level</u>. Since the memory chip is the least divisible memory element in the memory subsystem, atomic operations at this level will by definition always be indivisible. When an address is received by a memory with an indication that the addressed byte is to be atomicly read and set, the memory conducts a read operation and returns the data to the requester. However, before it completes the memory chip cycle, it performs a write cycle and sets the contents of that byte to all ones. This performs the atomic bit test and set function without having ever locked the bus or the memory controller, and without restrictions on the software as to the number, location, or use of semaphores. The only limitations are on the composition of the semaphores, which must be an entire byte, rather than (potentially) one bit, and that the "set" condition of the semaphore be all ones.

Resetting a semaphore is by its nature atomic, since if the set technique is successful in only ever allowing one owner, only one CPU will ever clear the semaphore.

Vector Bus Vector Arbitration

The Vector Bus vector arbitration on the Nanobus is unique in that it supports two types of vector transfers. In directed vector mode, interrupt vectors are addressed to one specific module, and therefore they pass directly from one module to another. In this mode, the Vector Bus functions as a standard data transfer bus. Directed vectors are very useful for an application typified by real time requirements, where the user may want to dedicate one processor to only execute an interrupt driven task or set of tasks to minimize execution latency.

The Vector Bus also supports a second mode of operation whereby the vectors <u>themselves</u> are arbitrated for. This mode allows the broadcast of a vector to one of three software controlled classes of modules, and then allows the hardware of all modules within the selected class to arbitrate among themselves as to which module should receive the vector. This mode is very useful for interrupt services that want to be spread across more than one processor to maximize the compute power that can be applied to an event, and to allow a variable amount of compute power to be applied dynamically as needed. An example of one such operation is the handling of terminal traffic. In a large system such as the Multimax, which can support hundreds of terminals, one would want to apply varying amounts of compute power to terminal service depending on the dynamics of the load.

Arbitrating for a vector is an important feature of the Multimax system because it causes the interrupt vector always to be directed by the hardware to the most interruptible module within a software selected class. All modules capable of receiving vectors have vector FIFOs for storing received vectors until they can be processed by the CPU. The vector arbitration allows the leveling of vectors in the FIFOs among modules, i.e. the module with the least number of interrupts in its FIFO and the lowest software set priority is the candidate for the interrupt vector.

This method for arbitrating among a variable number of modules for interrupt vectors is made possible because the Vector Bus is an open-collector bus. In order to arbitrate for a vector, all modules of the selected class assert on the Vector Bus a binary number representing the number of vectors in their FIFOs, and their physical slot number. All selected modules asynchronously compare each received data line from the Vector Bus with the data that it itself is driving onto the bus. The comparison starts with the most significant bit and works towards the least significant bit.

When a mismatch of any bit is found, the module disqualifies itself from the arbitration and stops driving the Vector Bus. Only one module in the system will walk down all the Vector Bus lines with a successful comparison. (Including the physical slot number in the arbitration process breaks any ties between modules in identical classes with the same number of vectors in their FIFOs.) This comparison technique is asynchronous, and results in a settling out between all of the modules in a class, leaving one and only one winner. Since the Nanobus is a synchronous bus, this asynchronous event is allowed to occur over multiple Vector Bus cycles, and during this time the synchronous state machines controlling the Vector Bus simply wait for a winner.

This Vector Bus design yields a very high performance system capable of supporting 1.2 million directed interrupts per second, and 500,000 arbitrated vectors per second.

Conclusions

The observed behavior of the Multimax system, and the Nanobus, have to date verified the analytical model that has been presented, though so far only write-through caches have been used in the system. The current system with 20 National Semiconductor 32032 CPUs uses an average of 11% of the available bus bandwidth. The hardware and software have required little tuning to achieve nearly linear increases in performance for each additional CPU added because of the simplicity and no restrictions guidelines around which the system was designed. A twenty processor system yields over nineteen processors worth of performance. The system has been proved capable of maintaining the 100 Mb throughput that it was designed for. The architectural concept of integrating the memory and interconnect design has proven very successful.

When designing a new interconnect system, one is sometimes tempted to concentrate the solution on only the problem immediately at hand, i.e. not designing for significant future expansion. Designing a point solution will almost always lead to a simpler and cheaper interconnect, but the penalty will be readily apparent when a second or third generation product is attempted and the foundation that has been laid is found to be inadequate. While the Nanobus design goals resulted in a system whose foundation is capable of delivering nearly ten times the throughput that the current system requires, and consequently results in slightly higher cost and longer development time, the next several generations of products now have a solid foundation on which to grow. This should yield computer systems which can expand significantly in performance, yet which can be developed within a short development cycle.

Annex, Multimax, and Nanobus are trademarks of Encore Computer Corporation.

Acknowledgement

The author wishes to acknowledge the invaluable contributions of the Multimax hardware development team to this project. They include:

John Bartlett - Designer of the Annex
Steve Corbin - Designer of the System Control Card.
Trevor Creary - Designer of the Shared Memory Card.
David Ford - Co-designer of the Dual Processor Card.
Brian Gill - Designer of the Ethernet and Mass Storage Card.
Russ Moore - V.P. Hardware Engineering
Charlie Namias - Co-designer of the Dual Processor Card.
Mark Natale and Dave Zopf, engineering assistants.

Bibliography

1. M. Ajomone, et. al. "Modeling Bus Contention and Memory Interference in a Multiprocessor System", *IEEE Transactions on Computers*, January 1983.
2. Bell, Mudge and McNamara, *Computer Engineering - A DEC View of Hardware Systems Design*, Bedford, Ma: Digital Press, 1978.
3. Kenneth I. Cohen, "Multiprocessing Architecture Tunes in to Transaction Processing",*Electronics*, January 27, 1983.
4. Encore Computer Corporation, *Multimax Technical Summary*, Marlboro, Ma.: First Printing, 1985.
5. Eli T. Fathi and Moshe Kreiger, "Multiple Microprocessor Systems: What, Why, and When",*IEEE Computer*, March 1983.
6. Steven Frank, "Tightly Coupled Multiprocessor System Speeds Memory Access Times", *Electronics*, January 12, 1984, pp. 164-169.
7. James Goodman, "Using Cache Memory to Reduce Processor - Memory Traffic", *Proc. 10th Annual Int. Symp. on Computer Architecture*, June 1983, pp. 124-131.
8. L.H. Holley et. al., "VM/370 Asymmetric Multiprocessing", *IBM Systems Journal*, Volume 18, number 1, 1979.
9. Dave Rodgers and Gary Fielland, "32-bit Computer System Shares Load Equally Among Up To 12 Processors", *Electronic Design*, September, 1984. pp.153-168.
10. Stone et. al., *Introduction to Computer Architecture - Second Edition*, Chicago, Ill.: SRA, 1980.
11. Alan Jay Smith "CPU Cache Memories", Draft, April 24, 1984, to appear as a chapter in *"The Handbook for Computer Designers"*, Ed. M. Flynn and G. Rossman.

About the Author

David J. Schanin received a B.S. degree in Computer Science from The City College, CUNY, in 1973, and an M.S. in Systems Engineering from Boston University in 1976. He has spent eleven years in industry, including over seven years at Digital Equipment Corp. Mr. Schanin was the founder and chief systems architect of Hydra Computer Systems, the company that was acquired by Encore Computer Corp. to build the Multimax computer system. He is currently president of Infinity Systems Inc., a consulting company specializing in in the design of multiprocessor computer systems, and high speed buses.

Mr. Schanin is an Adjunct Associate Professor at the Boston University College of Engineering, presently on leave. He is a member of the IEEE.

Optoelectronic Devices for Computing

By

Fred J. Leonberger

United Technologies Research Center
Silver Lane
East Hartford, Connecticut 06108

ABSTRACT

The status of a variety of optoelectronic devices for optical computing is reviewed. Most of these devices are useful for optical interconnects at the chip, intraboard and interboard level. Emphasis in the review will be on optical sources, detectors and switches. Materials and device technology that lead to integration with electronic devices will also be described. Finally, a brief overview of optoelectronic spatial light modulators useful for processing two-dimensional optical inputs is reviewed.

Optical techniques have a promising future in computing, with initial impact in the interconnection area.[1] This stems from problems using conventional electronics ranging from those at the chip and board level (pin-out limitations, wiring-dominated speed, loading, etc.) to the back plane (ground loops, terminating resistors, etc.). The rapid progress of optoelectronics and fiber optics can be brought to bear on these issues. Many computer companies are already installing fiber links, optical isolators, etc., and fiber optic local area networks will soon follow. Here, we will focus on recent research and development results to give an indication of future trends.

Illustrative data distribution examples from the areas of free space, guided wave and fiber optic technology are presented to show the range of optoelectronic technologies that could impact the interconnect problem. Figure 1 illustrates a free space/holographic approach. Here GaAs source arrays are hybridized with Si chips that incorporate photodetectors. This approach leads to large fan out and the possibility of dynamic interconnects if the hologram were replaced with a reflective spatial light modulator. Initial work on hologram requirements[2] as well as independent work on surface emitting lasers[3,4] has been reported. A second concept is the use of planar waveguide technology to achieve fixed but global interconnects for such problems as the perfect shuffle as shown in Figure 2. Here, source and detector arrays allow interconnections on Si wafers. Progress in this area include low-loss glass waveguides[5] and the demonstration of laser and detector arrays.[6] A third type of interconnect utilizes the high speed of GaAs electronic and optoelectronic components to time multiplex signals from an array of Si circuits, as shown in Figure 3. This method is well suited for fiber communications across large boards and between boards.

In all of the above examples, the rapid development progress of efficient optoelectronic components makes the approaches worthy of investigation. For example, lasers with efficiencies of ~ 30 percent with power dissipation of a few mW can lead to signals of 100 mW from photodiode-transimpedance amplifiers (detector quantum efficiency ~ 70%, f < 100 MHz). Only a few transitors are required for the detector circuit and subsequent amplification of logic levels, so it is not unreasonable to consider integrating such circuits where necessary in VLSI wafers and boards. These high performance devices can be combined with advanced hybridization techniques that minimize bond-lead parasitic effects. Further enhancement is possible with advances in GaAs monolithic optoelectronic integrated circuits.[7] An exciting possibility for the longer term is the monolithic integration of GaAs and Si devices on a common substrate. Much progress has been made in the past year in GaAs/Si heteroepitaxy as well as a demonstration of GaAs lasers on Si[8] and negligible degradation of Si CMOS transistors following the growth of GaAs layers and formulation of FETs on a common substrate.[9]

Integrated optic devices for switching fiber optic signals are also under development and have numerous computing applications. These switches are formed monolithically on wafers of electrooptic material using microelectronic technology. They are useful where fast (multi-GHz) low voltage (< 10 V) switching between fibers is required for single or small switch array applications. A schematic of a

CH2345-7/86/0000/0419$01.00 © 1986 IEEE

2 x 2 directional-coupler switch is shown in Figure 4. $LiNbO_3$ switches and modulators have received the most attention in recent years and are therefore more highly developed than their semiconductor parts. Drive voltage for a single device is generally < 10 V. Low insertion loss is an important parameter in designing any practical integrated optic device, and fiber pigtailed modulators with < 2 dB insertion loss (fiber-chip-fiber) have been achieved. Multiple devices integrated in the form of arrays of 4 x 4 directional coupler switches have been reported by several groups and recently and 8 x 8 switching has been demonstrated.[10] A 4 x 4 device has been pigtailed[11] and exhibited crosstalk of -25 dB, excess loss of 5 dB and switching voltages of ~ 30 V. In another area, arrays of Mach-Zahnder interferometers have been used to demonstrate analog-to-digital conversion at 4 bits and 1 Gigasample/sec, which is faster than currently achievable by electronic means.[12]

In GaAs switches, much recent progress has been made in developing low loss channel guides (~ 0.2 dB/cm at 1.3 μm) with low fiber coupling loss (up to 70% coupling efficiency). GaAs directional coupler switches with 30 dB extinction and Mach-Zehnder modulators with 3 GHz bandwidth have been demonstrated. Typical switching voltages are ~ 20V. In principal such devices can be integrated with electronic devices on a common substrate. Recently the use of GaAs/GaAlAs quantum-well devices has led to the potential of high efficiency electrooptical waveguide modulators as well as the prospect of practical all-optical switching devices. The latter would be particularly attractive in computing applications provided adequate switching contrast ratio and fan-out can be demonstrated.

The last device to be described is the spatial light modulator. This in general is a two dimensional device that allows programmed control (modulation, transmission or reflection) of each pixel. Devices under development utilize optical or electrical control. Here, we will focus on devices that use CCDs as part of the structure. Such devices are of special interest in optical processing because they allow lines of data to be scrolled (line transfer) which is attractive for a number of algorithms. Devices which are under development include those utilizing Si CCD/deformable membranes[13] and GaAs CCD/electroabsorption.[14] Figure 5 shows a schematic of one such device.

In summary, a large variety of optoelectronic devices are under development that can have important roles in computing. Initial applications are in interconnection, but the future impact will be in actual computation.

References

1. J. W. Goodman, F. J. Leonberger, S. Y. Kung and R. A. Athale, Proc. IEEE 72, 850 (1984.)

2. R. K. Kostak, J. W. Goodman and L. Hesselink, 1984 OSA Annual Mtg, San Diego, CA, Paper ThP2.

3. Z. L. Liau and J. N. Walpole, Appl. Phys. Lett., 467, 115 (1985); J. J. Yang et al, CLEO'86 Digest, Paper PD-ThT7 (OSA, 1986).

4. K. Iga, S. Ishikawa, S. Ohkouchi and T. Nishimura, Appl. Phys. Lett. 45, 348 (1984).

5. J. T. Boyd, R. W. Wu, D. E. Zelmon and A Nauman, Optical Engineer 24, 230 (1985).

6. See, for example, P. P. Deimal, et al 1985 IEEE/OSA Optical Fiber Communication Conf., San Diego, CA, Paper ThC4.

7. See, for example O. Wada, H. Hamgauchi, S. Miura, M. Makinchi, K. Nakai, H. Hermiatsu and T. Sakurai, Appl. Phys. Lett. 46, 981 (1985).

8. T. H,. Windhorn and G. M. Metze, Appl. Phys. Lett. 47, 1031 (1985).

9. R. Fischer et al, Appl. Phys. Lett. 47, 983 (1985).

10. P. Granestrand et al, Digest OFC'86 p. 4 IOSA, 1986.

11. G. A. Bogert, E. J. Murphy and R. T. Ku, Digets IGWO'86, paper PDP3 (OSA, 1986)

12. R. A. Becker, C. E. Woodwand, F. J. Leonberger and R. C. Williamson, Proc. IEEE 72, 802 (1984).

13. D. R. Pape, Proc. SPIE 465, 17 (1984)

14. R. H. Kingston, B. E. Burke, K. B. Nichols and F. J. Leonberger, ibid, 9.

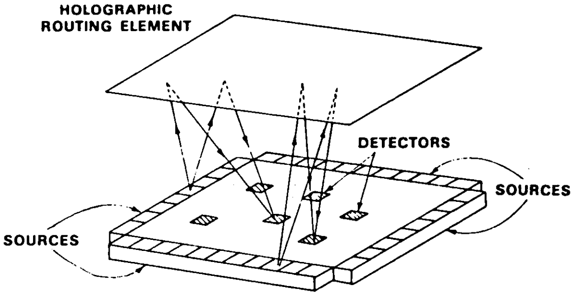


Fig. 1. Hybrid Ga/As Si free-space approach to data communication.

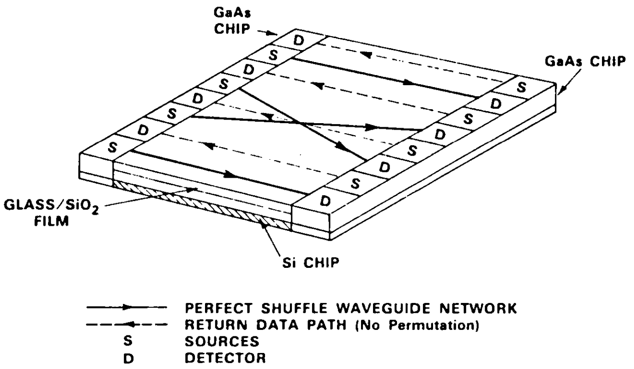

GaAs CHIP
GaAs CHIP
GLASS/SiO$_2$ FILM
Si CHIP
S
D
PERFECT SHUFFLE WAVEGUIDE NETWORK
RETURN DATA PATH (No Permutation)
S SOURCES
D DETECTOR

Fig. 2. Optical perfect shuffle network.

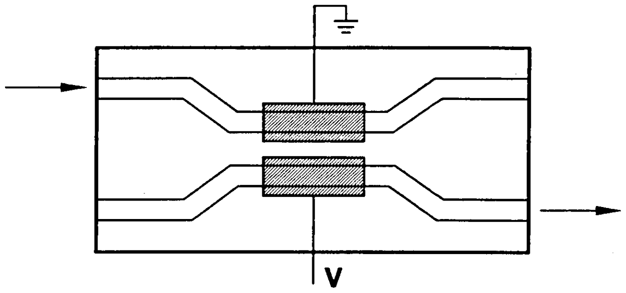


Fig. 4. Directional' coupler switch.

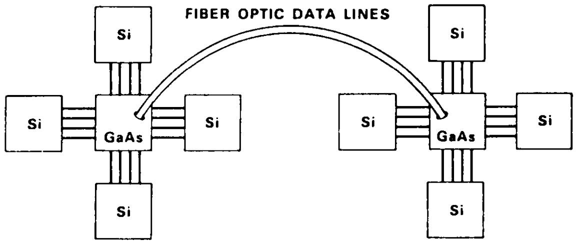


Fig. 3. Modules of Si chips surrounding GaAs chips, and with communication between GaAs chips via optical fibers.

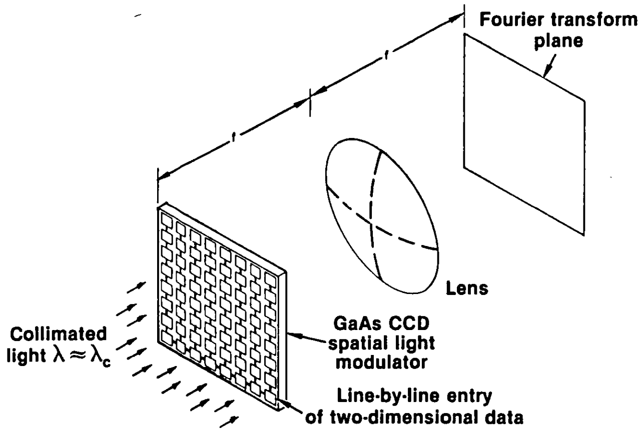

Fourier transform plane
f
f
Lens
Collimated light $\lambda \approx \lambda_c$
GaAs CCD spatial light modulator
Line-by-line entry of two-dimensional data

Fig. 5. Two-dimensional optical signal processing using GaAs spatial light modulator.

ARCHITECTURES FOR OPTICAL MATRIX MULTIPLIERS

Ravindra A.Athale

The BDM Corp.
7915 Jones Branch Drive
McLean, VA.22102

ABSTRACT

The operation of multiplying two matrices is one of the fundamental operations that is frequently encountered in signal processing, image processing and numerical computations. This operation can be performed by optical systems employing parallelism. This paper discusses different architectures for optical processors performing this operation. The architectures are divided into several classes according to the degree of parallelism entailed and the type of interconnections employed.

INTRODUCTION

The operation of multiplying two matrices is one of the most common operations encountered in signal processing, image processing, and numerical computation involving solutions of a system of linear simultaneous equations. This operation is mathematically defined in equation 1 below:

$$C_{ij} = \sum_{k} A_{ik} B_{kj} \qquad [1]$$

For i=1 to N and j=1 to N

In FORTRTAN , this operation can be coded by the following program:

```
    DIMENSION  A(N,N),  B(N,N),   C(N,N)
    DO 10 I=1,N
    DO 10 J=1,N
    DO 10 K=1,N
    C(I,J)  = C(I,J)  + A(I,K)  *  B(K,J)
10  CONTINUE
```

It should be noted that there are three DO loops in the program with N passes per loop. Since the operations performed in the loops are multiplication and addition, this program involves N^3 multiplications and additions. The indices I and J orrespond to the array index of the output matrix C, and the index K is the common array index for the input matrices A and B, over which summation is performed

Optical processing systems are capable of performing the operations of analog multiplication and addition in parallel between one- or two-dimensional array of positive real numbers and achieve global communication between them. This property of optics has been used in the past for signal processing and image processing operations primarily based on Fourier transforms. Recently, optical processing systems have been investigated for performing matrix operations in parallel, which will make them more widely applicable. The large variety of optical architectures reported in the literature can be classified into three different categories according to the level of parallelism:

(i) one-dimensional systems performing N operations in parallel
(ii) two-dimensional systems performing N^2 operations in parallel
(iii) two-dimensional systems employing multiplexing performing N^3 operations in parallel.

The first two categories can be further subdivided according to which of the DO loops in the program for matrix multiplication are implemented in parallel. The optical processors in each subcategory can be implemented with different technologies, such as integrated optics, acoustooptics, electrooptics etc. A further classification results when one considers which parameters - e.g. time or space, temporal or spatial frequency - are used to multiplex the operations and which are used for summation/integration. A truely comprehensive study which includes a detailed analysis of all of these variations will indeed be massive. In this paper we will briefly discuss the different optical architectures in a generic way. We will place particular emphasis on the types interconnects involved in each of these archietctures since it will high light the special advantages offered by the use of optics.

CH2345-7/86/0000/0422$01.00 © 1986 IEEE

Other details can be found in the articles that are referenced at the end of the paper.

N-PARALLEL OPTICAL SYSTEMS:

The optical processor in this category perform N multiplications and/or additions in parallel. If we look at the program listing, we will get three choices for the DO loop index that we could implement in parallel with an optical system. The indices I and J correspond to the array indices of the output matrix C, and hence are equivalent for the purpose of this analysis. Hence there are two choices, (i)parallelizing the DO loop over index I (or J), and (ii)parallelizing the DO loop over index K.

(i) This choice of the DO loop index (I) leads to optically performing N multiplication in parallel and the summation over the K index and the DO loop over J sequentially. The resulting optical architecture is shown in Figure 1. This system contains a one-dimensional (1-D) spatial light modulator (SLM), a point modulator, a 1-D time-integrating detector array, and collimating optics (for signal broadcasting) and imaging optics. The basic operation that this processor performs in one clock cycle of the 1-D SLM is that of scalar-vector multiplication defined by equation 2:

$$\underline{c} \quad = \quad a\ \underline{b} \qquad [2]$$

It should be noted that this system uses an interconnection network for broadcasting in one dimension.

(ii) This choice of the DO loop index (K) leads to performin N multiplications and additions in parallel. The resulting architecture is shown in Figure 2. This system contains two 1-D SLM, a point detector, and focussing optics (for fan-in) and imaging optics. The basic operation that this processor performs in one clock cycle of the 1-D SLM is that of vector inner product defined by equation 3:

$$c \quad = \quad \underline{a}^T\ \underline{b} \qquad [3]$$

It should be noted that this system uses an interconnection network for fanning in N-data channels in one dimension.

Both of these optical architectures do not fully exploit the 2-D parallelism of optics. However, their 1-D nature allows for an integrated optic implementation that allows for higher speed operation (100 MHz) in a small and rugged package. It should also be noted that the operations described above are of interest in and by themselves for signal processing/symbolic processing operations and do not have to be considered in the context of matrix multiplications alone. The scalar-vector multiplication is is useful in calculating a weighted version of a given input and the vector-vector inner product can give a similarity measure between the two vectors to be compared.

N² -PARALLEL OPTICAL SYSTEMS

The optical processors in this category perform N² multiplications and/or additions in parallel and hence fully exploit the 2-D parallelism offered by optics. If we look at the program listing, we will get two distinct choices for the two DO loops that we can implement in parallel with an optical system. (i) this choice involves performing the DO loops over I and J index in parallel, (ii) the second choice involves performing the DO loops over I (J) and K in parallel.

(i) This choice of DO loop indices (I and J) leads to performing N² multiplications in parallel and the summation over the K index sequentially. The resulting optical architecture is shown in Figure 3. This system contains two 1-D SLMs arranged orthogonal to each other, optics for collimating and focussing simultaneously along orthogonal directions, and a 2-D time-integrating detector array. The basic operation that this processor performs in one clock cycle of the 1-D SLM is that of a vector-vector outer product defined in equation [4]:

$$C' \quad = \quad \underline{a}\ \underline{b}^T \qquad [4]$$

where C'is a rank one matrix that is NXN in dimension. It should be noted that this optical architecture uses two 1-D input arrays and yet calculates a 2-D output array. The interconnections utilized by this system are quite complex in that they involve broadcasting along one spatial dimension and fanning-in along orthogonal spatial direction. One element of the 1-D SLM encoding one element of the vector $\underline{b}$ is simultaneously accessed by N elements of the other input vector $\underline{a}$ without contention. The data path for an element of vector $\underline{a}$ retains its identity. So after being multiplied by an element of vector $\underline{b}$, it is directed to a specific location on the 2-D time-integrating detector array as shown in Figure 3. This system can be considered to be a spatially multiplexed version of the 1-D scalar-vector multiplier depicted in Figure 1.

(ii) This choice of DO loop indices (I and K) leads to performing N^2multiplications and additions in parallel and performing the DO loop over the remaining index (J) sequentially. The resulting optical architecture is shown in Figure 4. This system contains one 1-D SLM, one 2-D SLM, one 1-D non-integrating detector array, and optics for imaging/collimating on the input side and for imaging/focussing on the output side. The basic operation that this processor performs is that multiplying a column vector of matrix B by matrix A defined in equation [5]:

$$\underline{c} = A\,\underline{b} \qquad [5]$$

The interconnections utilized by this system involve broadcasting the vector $\underline{b}$ to all rows of matrix A on the input side and fanning in the product of a row of A and vector $\underline{b}$ on to a specific detector element on the output side. This system can be considered to be a spatially multiplexed version of the 1-D vector-vector inner product processor depicted in Figure 2.

The optical processors in this category do utilize the 2-D parallelism of optical systems by employing the spatial variables for multiplexing and/or for integration. These two operations are also useful in and by themselves in signal and image processing. The operation of outer product is critical in synthesizing a complicated matrix (e.g. an image) from several simple "primitive" images corresponding to rank one matrices. The vector-matrix multiplication implements a generalized linear transformation on a 1-D input. The optics utilized by these systems contains off-the-shelf components like spherical and cylindrical lenses to give different properties along the two orthogonal directions.

N^3-PARALLEL OPTICAL SYSTEM:

The optical processors in this category perform N^3 multiplications and additions in parallel. Since the matrix-matrix multiplication involves N^3 multiplications/additions this class of optical processors will perform the operation in one clock cycle of the active devices involved. Since we are performing all the DO loops in parallel, there is only one way of formulating the processor mathematically. Different architectures result, however, when designing such a system optically.

The schematic diagram of the N^3-parallel optical system is shown in Figure 5. This system uses two 2-D SLMs for inputing matrices A and B and a 2-D detector arrayfor detecting the output matrix C. The optics involved is complicated and has to be implemented via computer generated holography.

One intriguing feature of this architecture is that it employs only N^2 active elements and still performs N^3 operations in parallel. Therefore the efficiency of this architecture as a parallel processor (computational speedup / number of processing elements) is N which can be much larger than 1! In most other designs for parallel processors the goal is to achieve an efficiency of 1, indicating a linear speedup with the number of processors. This indeed represents a unique feature of optics in that it utilizes the parallel, contention free, multiple-access communication capability of optics to add an extra dimension to the computational power of a paralle processor. Since the operation of matrix-matrix multiplication is a well structured operation with little interdependance between the calculations of the elements of the output matrix, this feature of optical systems can be exploited to fullest extent to achieve superlinear speedup in a parallel architecture.

The N^3-parallel optical processor can be viewed as a multiplexed version of the N^2-parallel optical architectures. Since in the earlier section we discussed two different optical systems for those architectures, we can view the N^3-parallel architecture as multiplexed versions of either the outer product optical processor or the vector-matrix optical processor. It should be emphasized that this division is strictly for conceptual clarity and is not indicative of a deeper category.

Figure 6(a) shows the schematic diagram of an N^3-parallel optical processor viewed as N outer product optical processors that are spatially multiplexed. In the system depicted in Figure 3 the addition of the outer product matrices for calculating the output matrix C was performed by integration in time on the 2-D detector array. In Figure 6(a), all the outer products are simultaneously calculated and are therefore added by integration of N terms in space by fanning in the apropriate signals emerging from the elements of the second matrix B. Each column of matrix A is encoded by a light beam traveling at an apropriate angle so as to interact with the correct row of matrix B as indicated in Figure 3. On the output detector array, the results of the different outer products performed in parallel converge thus performing the final integration. Figure 6(a) shows the optical paths for only two outer products for clarity.

Figure 6(b) shows the schematic diagram of the N^3-parallel optical processor viewed as a spatially multiplexed version of the optical vector-matrix multiplier shown in Figure 4. In that system, the operation of vector-matrix multiplication was performed in one cycle generating one row of the output matrix C. The full answer was calculated by generating different rows of matrix C in a time-sequential fashion. In the system depicted in Figure 6(b), all rows of matrix A are available simultaneously while the matrix B is used in N different vector-matrix products simultaneously. The resultant N rows of the output matrix C are spatially separated and are detected by the rows of the output detector array. In this processor, each row of matrix A is encoded by a light beam traveling at an appropriate angle. At matrix B, all rows of A are simultaneously accesing the elements of B while keeping their distinct identity. The unique encoding of the light beam for each row of A causes the output to be separated on the detector array thus providing all rows of matrix C in parallel.

CONCLUSION

Optical processors offer the unique features of two-dimensional parallelism in the basic arithmetic operations of analog multiplications and additions and global, parallel, and contention-free communication between the two-dimensional array of simple processing elements. These features can be exploited to build optical processors for performing the general operation of matrix-matrix multiplication. In this paper, we outlined several different architectures of parallel optical processors for performing this operation with varying degree of parallelism. In each category the type of communication involved were emphasized. A unique optical architecture was described that uses the contention-free communication offered by optics to obtain a speedup of N^3 with a system containing only N^2 active processing elements.

REFERENCES

GENERAL OPTICAL PROCESSING REFERNCES:

J.W.Goodman, "Introduction to Fourier Optics", McGraw-Hill, 1968.

S.H.Lee ed. "Optical Information Processing: Fundamentals", New York: Springer-Verlag, 1981.

Proceedings of IEEE Special Issue on Optical Computing, Volume 65, No.1, January 1977.

Proceedings of IEEE Special Issue on Optical Computing, Volume 72, 1984.

GENERAL REVIEW ARTICLES ON OPTICAL MATRIX PROCESSORS:

R.A.Athale, 10th International Optical Computing Conference, April 6-8, 1983, Cambridge, Mass. IEEE Catalog Number 83CH1880-4.

D.Casasent, Proc. IEEE, Vol.72, p.831,1984.

W.T.Rhodes and P.S.Guilfoyle, Proc. IEEE, Vol.72, p.820, 1984.

1-D INTEGRATED OPTICAL ARCHITECTURES:

C.Verber, Proc. IEEE, Vol.72, p.942, 1984.

OPTICAL VECTOR-MATRIX MULTIPLIERS:

M.A.Monahan, K.Bromley,and R.P.Bocker, Proceedings of IEEE, Vol.65, p.121, 1977.

J.W.Goodman, A.R.Dias, and L.M.Woody, Optics Letters, Vol.2,p.1, 1978

H.J.Caulfield, W.T.Rhodes, M.J.Foster, and S.Horwitz, Optics Comm., Vol.40, p.80, 1981

OPTICAL OUTER PRODUCT PROCESSORS:

R.A.Athale and W.C.Collins, Applied Optics, Vol.21, p.2088, 1982.

R.A.Athale and J.N.Lee, Proc. IEEE, Vol.72, p.931, 1984.

R.P.Bocker, H.J.Caulfield, and K.Bromley, Applied Optics, Vol.22, p.804, 1983.

OPTICAL FULLY PARALLEL PARALLEL MATRIX-MATRIX MULTIPLIERS:

R.A.Heinz, J.O.Artman, and S.H.Lee, Applied Optics, Vol.9, p.2161, 1970.

P.N.Tamura and J.C.Wyant, Proceedings of SPIE, Vol.83, 1975.

A.R.Dias, in "Optical Information Processing for Aerospace Applications", NASA Conference Proceedings 2207, (NTIS, Springfield, VA.).

Y-Z Liang and H.K.Liu, Optics Letters, Vol.9, p.322, 1984.

H.Nakano and K.Hotate, Applied Optics, Vol.24, p.4238, 1985.

OPTICAL MATRIX PROCESSORS: ALGORITHMS AND HIGHER ORDER OPERATIONS

H.J.Caulfield, D.Dvore, J.W.Goodman, and W.T.Rhodes, Applied Optics, Vol.20, p.2263, 1981.

D.Casasent and M.Carlotto, Applied Optics, Vol.21, p.147, 1982.

R.A.Athale and J.N.Lee, Optics Letters, Vol.8, p.590, 1983.

D.Casasent, C.Neuman, and J.Lycas, Applied Optics, Vol.23, p.1960, 1984.

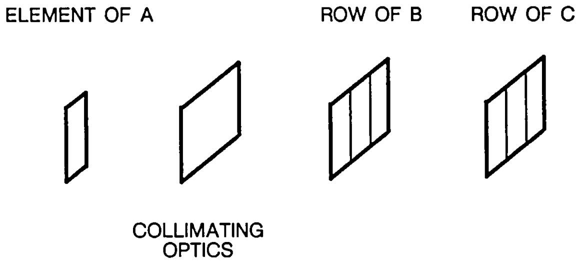


Figure 1. The N-parallel scalar-vector product optical processor. (Imaging optics ommitted.)

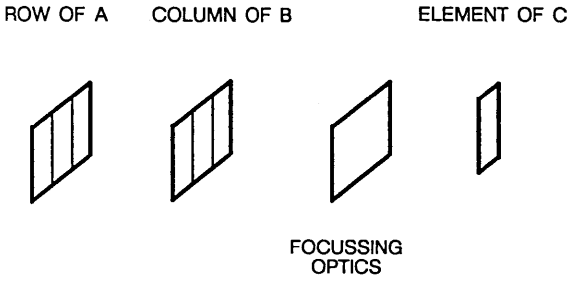


Figure 2. The N-parallel inner product optical processor. (Imaging optics ommitted.)

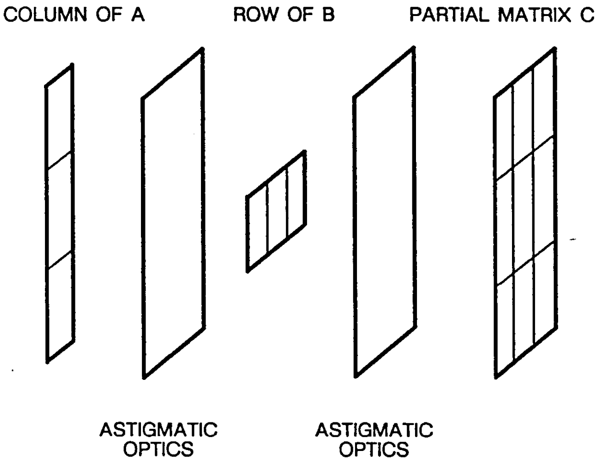


Figure 3. The N^2-parallel outer product optical processor.

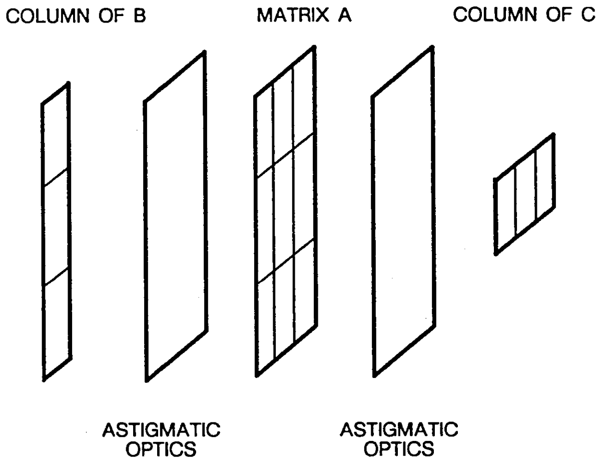


Figure 4. The N^2-parallel vector-matrix optical processor

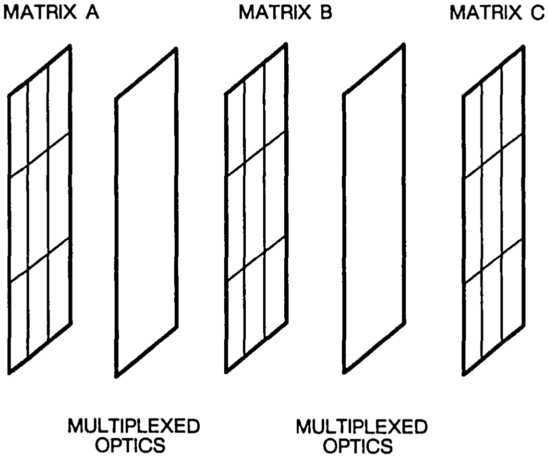


Figure 5. The N^3-parallel optical processor.

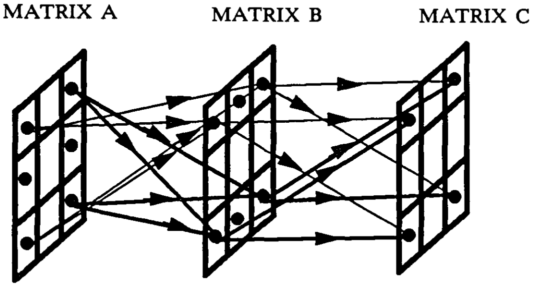


Figure 6(a). The N^3-parallel optical processor viewed as a multiplexed outer product processor.

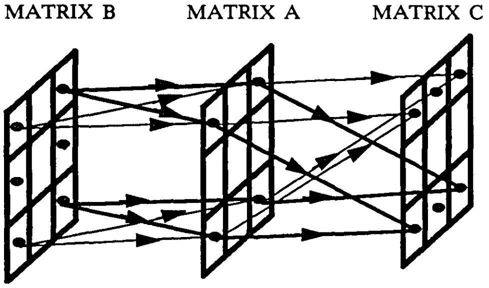


Figure 6(b). The N^3-parallel optical processor viewed as a multiplexed vector-matrix processor.

OPTICAL REALIZATIONS OF NEURAL NETWORK MODELS

Demetri Psaltis

California Institute of Technology
Department of Electrical Engineering
Pasadena, CA, 91125

INTRODUCTION

The optical implementation of computing systems whose structure and function are motivated by natural intelligence systems is a subject that involves optical computing and neural network models for computation. These are two areas that have individually received attention in recent years and they share the common property that they promise to provide solutions for fundamental problems in computation. In the case of optical computers the limitation that is being addressed is communication. With optics it is possible to have large arrays of processing elements communicating with each other without connecting a wire between each pair. The need to provide wires for communication in an electronic circuit is perceived as a major technological limitation of VLSI [1]. The primary justification for optical computing is therefore to extend the communication capability in computers [2,3]. It is not clear however precisely how this global communication capability can be put to good use. Through free space interconnects and volume holograms we can have thousands of computing elements all talking to each other at the same time. Is it possible to do useful computation in such a system? We look at neural network models for an answer to this question.

NEURAL NETWORK MODELS OF COMPUTATION

Natural systems of intelligence are being examined as a possible source of inspiration for building computers, partially because of the persistent difficulty we have had in performing tasks such as simple pattern recognition problems. Such problems appear trivial for humans but very cumbersome for conventional computers. Moreover these problems have not become substantially easier to solve as the processing power of computers has increased and allowed them to tackle problems that are well beyond the capability of a human. There is a suspicion therefore that the way in which these problems are tackled by neural networks is fundamentally different from the way we approach these problems in conventional computers. If we understand what these differences are this will help us design better computers to solve these problems.

An electronic computer and a neural network have some morphological similarities. They both consist of a large number of simple processing elements (neurons versus gates) that are connected to each other. In both systems the individual computing elements perform simple operations but the organization of the simple elements into a large system allows complicated tasks to be performed. The most striking **difference** between a neural network and an electronic circuit in a computer is the degree of connectivity. Each neuron is connected to thousands of other neurons whereas in an electronic computer typically each gate receives inputs from few (two or three) other gates. The type of connectivity that exists in a neural network allows thousands of neurons to collectively and simultaneously influence the state of each neuron. The role of this extensive and complex communication network must play a prominent role, indeed the dominant role, in providing neural networks their computing power. Another basic difference is the role of time. In conventional computing time is paramount. The number of steps required for the execution of a program is the primary measure of complexity. The speed-up that one obtains by breaking down a problem to many smaller tasks that are executed in parallel is the driving force for most of the developments in computer architecture. Certain computations are done quickly with neural networks because they process information collectively. We draw the distinction here between parallel and collective processing. In a parallel computer a problem is decomposed into small pieces that are individually worked upon. In the computations that we envision being performed by networks, a set of neurons collectively work on the same problem. The global communications allow each node (neuron) to change its state in response to the state of the entire network. The transitions from one state to another are determined in part by the operation of the neurons themselves but principally by the connections amongst them. The **program** is stored in the communication network. Neural nets modify their structure to fit the requirements of the problem. The distinction between software development and hardware organization is almost entirely removed.

CH2345-7/86/0000/0428$01.00 © 1986 IEEE

The highly abstract neural network models which are of primary interest to us share two properties. The first is the Hebbian [4] hypothesis which states that information is stored by modifying the strength of the connections between neurons. The second is that the nonlinear operation performed by the neurons is relatively simple, often assumed to be a thresholding operation [5]. Accordingly an optical realization of a neural network model requires two main components: an array of optical nonlinear elements to simulate the action of the neurons and a system to specify the optical interconnections among the processing elements (neurons); since the connections in this case constitute the memory, they must be modifiable. A diagram of a generic optical implementation of a neural network is shown in Fig.1. We envision that the processor array will be fabricated using planar technology (most likely semiconductor technology) and thus the array of neurons is arranged on a two dimensional planar configuration. The merit of the optical approach is that the interconnections are specified externally to this plane. The three dimensional construction of the optical system is the key feature that allows us to a) fully utilize the real estate on the processor plane for the simulation of neurons and b) possibly use the volume of a nonlinear crystal to record a hologram that specifies the interconnections. In what follows we will discuss candidate emerging optical technologies for simulating neurons and their interconnections and review optical implementations of associative memories.

OPTICAL REALIZATIONS

There are three candidate device technologies for optically simulating the nonlinear mapping performed by the neurons: spatial light modulators [6], integrated optoelectronics [7], and arrays of nonlinear optical switches [8]. Spatial light modulators have been investigated for many years primarily for optical image processors. A practically useful device is not yet available, but several decvices suitable for laboratory experiments exist. Most of the SLM's can be operated in a high contrast mode, producing a two dimensional light modulation that is proportional to a thresholded version of an image, thereby simulating the action of neurons in network models that are based on threshold logic. These devices are playing an important role at this stage, because they allow us to perform meaningful experiments. They will continue to play this role in the foreseeable future but in the long term optoelectronics and nonlinear optics appear more promising.

The optoelectronic approach to simulating an array of neurons involves the integration of two dimensional arrays of detectors and light emitters (LEDs) on a single (possibly hybrid) chip. The output of each detector can then be connected via a saturating amplifier, or an appropriate analog circuit that perfroms the required nonlinear mapping, to the corresponding LED. Thus the combination of the detector, the LED, and the intervening electrical circuit comprise the neuron. There are several advantageous features of this approach. First, it is technologically the most straightforward. The integration of optical devices and electronic circuits is currently being pursued for providing optical interconnections to VLSI circuits. The devices required for simulating an array of neurons are particularly simple and regular and therefore they ought to be relatively easy to manufacture. It is worth pointing out that the only electrical connections that would be required to such an optoelectronic chip are to provide power. The second interesting aspect is the relative flexibility there is in specifying the action of the neurons by designing the electronic circuit. Probably the most difficult problem in manufacturing practical devices of this type is power dissipation. Nevertheless, we believe that optoelectronics is the technology that is most likely to provide us with devices that are practically useful for implementing optical networks.

In recent years we have seen dramatic improvements in the area of nonlinear optics that has led to numerous demonstrations of optical switching including two dimensional arrays of swithches. The major problem with these arrays at present is the very high power required to switch each element making large arrays impractical. These optical switches are intended either for optical communications or digital optical computing. Therefore most of the work in the literature deals with using these devices to implement simple Boolean gates. With these applications in mind, most of the research effort has been directed towards increasing the switching speed and this high speed operation results in exceedingly high power requirements. For a neural network simulation, speed is not essential. The time response of neurons in the brain is in the millisecond range. What is required is ever larger arrays that switch slower and thus have reduced power requirements. Therefore, with the thrust of the research in nonlinear optical switches being almost orthogonal to what is needed for the netrwork implementations, we feel that nonlinear optics will not play a significant role in this area in the near future, unless some of the research is appropriately redirected.

We now turn our attention to interconnections, the second major component of an optical network. A crucial consideration here is that we must be able to simulate plasticity, i.e. modify the interconnections. In neural net models, the strength of the interconnections is used to store the "program" and the data; it is the memory of the system. Moreover, a particularly interesting feature of many of the models is supervised learning (as opposed to software development) as well as self organization (a form of unsupervised learning). These features emerge as a result of continuous dynamic modification of the interconnections and thus optical implementations of such models also require interconnections that can be modified continuously. The second important considerations in selecting such a medium is its storage capacity, i.e. how many resolvable spots can be recorded in the area or the volume of the material. This directly determines the number of interconnections that are being simulated which in turn determines the amount of information that can be stored in the network. For the networks we envision, we would like to be able to arbitarily interconnect at least 10^4 neurons which translates to 10^8 connections. This consideration eliminates several candidates, such as spatial light modulators, from being considered as a variable dynamic interconnection medium for neural network simulations. We will discuss two separate optical technologies that we believe are the most promising for this purpose: magnetooptic memory disks and photorefractive crystals.

Write-only optical memories are already being used for video and audio recording and increasingly in computers. Magnetooptic disks, which can be erased and rerecorded, are not yet commercially available however this should happen in the immediate future. We can have billions of resolvable spots on a single optical disk and we can reconfigure the recording magnetically. Presently such optical disks are used as mass storage devices in electronic computers. Information is accessed from these memories serially by focusing a single laser onto a single bit (out of several billion that may be recorded on the same surface) and measuring the reflected light on a single detector. By mechanically spinning the disk and moving the laser head we get access to any one bit. The potential exists for accessing information stored on the disk much faster and much more effectively if the same disk is used to store information such that the connections between several thousand lasers and detectors are specified. In the following section we will discuss a specific model of associative memory and an optical implementation that can be implemented using optical disks.

Historically, holography is the first link between optics and neural network models. Van Heerden [10] and Gabor [11] both discussed holography as an associative memory and mentioned the possibility that there are analogies between holography and the way information is stored in the brain. In holography a pattern is reconstructed by illuminating the hologram with the reference beam that was used during recording. Thus a hologram is a form of associative memory. It is also known that a reconstuction can be obtained from only a part of a hologram a property that is shared by many types of associative memory. More recently holography is being considered as a method for providing optical interconnections [3,9] in an optical or optoelectronic computer. Such connections can be made programmable by recording the hologram on photorefractive (PR) crystals [6]. When a PR crystal is exposed to light, free charges are photogenerated and eventually retrapped. This process results in the redistribution of the charge within this crystal in a pattern similar to the pattern of the illuminating light intensity. The presence of spatially varying charge density induces internal fields in the crystal which in turn modify the index of refraction. Thus a phase hologram is recorded which can be reconstrued by illuminating the crystal with a light beam. PR crystals represent the second major candidate for realizing programmable optical interconnects. The advantages of PR crystals over magnetooptic disks are the optical rather than magnetic recording and the potential for volume storage. The number of optical connections that can be specified with a volume hologram is proportional to the volume of the crystal divided by the wavelength cubed. The principal advantage of magnetooptic disks on the other hand is the relatively advanced state of the art of the technology, which will continue to mature since there is a well identified commercial application for this technology.

The device technologies that are needed for the optical realization of a neural network that were outlined above all require further development before they can be used in practical systems. However the progress being made in these device technologies breeds sufficient optimism for the future outlook and this has led to designs and laboratory demonstrations of optical architectures [12-18] to simulate neural network models of associative memory [19]. In the following section we focus on a particular auto-associative memory model [20-22] and present experimental demonstrations of it.

OPTICAL ASSOCIATIVE MEMORIES

Let $v_i^{(m)}$ $i = 1, ...N$ $m = 1, ...M$ be binary (+1,-1) valued vectors. Then a matrix T_{ij} is formed by summing the outer products of all the vectors:

$$T_{ij} = \sum_{m}^{M} v_i^{(m)} v_j^{(m)} \quad , \quad T_{ii} = 0 \qquad (1)$$

Multiplying the matrix T_{ij} with any one of the stored vectors results in the reproduction of a noisy estimate of the same vector:

$$\sum_{i}^{N} T_{ij} v_i^{(1)} = (N-1) v_i^{(1)} + \sum_{m \neq 1}^{M} \sum_{i \neq j}^{N} v_i^{(m)} v_j^{(m)} v_i^{(1)} \qquad (2)$$

The first term in the above sum is the "correct" term amplified by N whereas the second term is unwanted interference whose value on the average is approximately $\sqrt{(N-1)(M-1)}$. Thus if M is sufficiently larger than N the first term will dominate and the output will look like the input. If this is the case then with high probability thresholding will eliminate the cross talk term and reproduce the input vector.

In the biological interpretation of the model the matrix T_{ij} represents the strengths of the connections among neurons and the thresholding describes the action of the neurons. The final characteristic of this model is feedback.

The output of each neuron (threshold element) becomes one of the inputs to all the other neurons in a fully connected network. We model this by making the binary vector that is produced by the thresholding operation the input vector to the matrix T_{ij} for the next iteration thereby making the correctly stored vectors (i.e. input vectors that are reproduced at the output) stable states of the network. A block diagram of this model is shown in Figure 2. The network is initialized by setting the state of the neurons according to an external input and allowing the network to reach a stable state. The system usually chooses the stable state that is most similar to the initial vector. Thus initial vectors that are distorted or partial versions of one of the stored vectors result in the stable state to be reached that is equal to this stored vector.

The most straightforward optical implementation of the Hopfield memory is by adding nonlinear feedback to an optical vector-matrix multiplier [23]. The system is shown in Figure 3. A linear array of detectors/saturating electronic amplifiers/light emmiting diodes is used to simulate the array threshold units. The strength of the interconnections between neurons is specified by the tranmittance of a two dimensional mask placed between the LEDs and the detectors. The details of this implementation and its experimental demonstration can be found in references [12] and [13]. In this paper we discuss a holographic implementation that is a modification of an architecture that is presented in reference [12] and was recently demonstrated experimentally by Paek and Psaltis [24].

Examination of equation (2) reveals that the read-out process in this associative memory can be decomposed as a cascade of two steps: An inner product between the input and all the stored vectors (the summation over i in Eq.(2)) followed by a summation of all the stored vectors each weighted by the corresponding inner product. An optical system that implements this associative memory is shown in Figure 4. An input image illuminates the system through a beamsplitter. The array of optical thresholding elements is placed immediately following the beamsplitter. The combination of the Fourier transforming lens L_1, the first hologram placed in plane P_2 and transforming lens L_2 comprise an optical correlator [25] that is used to form inner products between the thresholded input image and a library of stored references. The Fourier trasnsforms of the reference images are stored in the first hologram each on a separate spatial frequency carrier such that the inner products appear spatially separated at plane P_3. A pinhole array is placed at P_3 with each pinhole located at the position where one of the inner products forms. The light transmitted through each pinhole is collimated by lens L_3 and it illuminates a second hologram in plane P_4. The Fourier transforms of the same set of reference images are also stored in the second hologram. The arrangement of the pinholes with respect to the spatial frequency encoding that is used in the second hologram results in the reconstruction of all the stored images overlapping at plane P_1, and each weighted by the appropriate inner product. After thresholding the image at P_1 the light is fed back into the loop until a stable state is reached. This final state can be probed on the other side of the beam splitter as shown in Figure 4.

ACKNOWLEDGEMENTS

Acknowledgements are due to Y. Abu-Mostafa, N. Farhat, J. Hong, E. Paek, C. Park, and S. Venkatesh who are the authors co-workers in this area. This work is funded by DARPA.

REFERENCES

1. I. Sutherland and C. Mead, *Scientific American*, vol. 237, p. 210, 1977.
2. A. Huang, *Proc. IEEE*, vol.72, 1984.
3. J. Goodman et.al., *Proc. IEEE*, vol. 72, 1984.
4. D. Hebb, *Organization of Behavior*,Wiley, New York, 1941.
5. W. S. McCulloch and W. Pitts, *Bull. Math. Biophysiol.*, vol. 5, p. 115, 1943.
6. A. Tanguay Jr., "Spatial Light Modulators for Real Time Optical processing", in *Future Directions for Optical Information Processing*, Lubbock, Texas, Texas Tech. Univ. 1981.
7. J. Neff, this volume.
8. H. Gibbs, S. McCall, T. Venkatesan, *SPIE*, vol. 269, p. 75, 1981.
9. A. Sawchuk and T. Strand, *Proc. IEEE*, vol. 72, 1984.
10. P. van Heerden, *Appl. Opt.*, vol. 2, p. 387, 1963.
11. D. Gabor, *IBM J. Res. Dev.* ,vol. 13, p. 156, 1969.
12. D. Psaltis and N. Farhat, *Opt. Lett.*, vol. 10, pp. 98-100, Jan. 1985.
13. N. Farhat, D. Psaltis, A. Prata, E.G. Paek, *Applied Optics*, vol. 24, p. 1469, 1985.
14. A. Fischer and C. L. Giles, *Proceedings of the IEEE 1985 Comcon Conference*, IEEE cat. no CH2135-2/85,342(1985).
15. D. Anderson, *Opt. Lett.*, vol. 11, p. 56, 1986.
16. M. Cohen, *Proc. SPIE*, vol. 625, 1986.
17. B. Soffer, G. Dunning, Y. Owechko, and E. Marom, *Opt. Lett.*, vol. 11, p. 118, 1986.
18. A. Yariv and S. K. Kwong, *Opt. Lett.*, vol. 11, p. 56, 1986.
19. T. Kohonen, *Associative Memory: A System Theoretic Approach*, Springer-Verlag, Berlin, 1977.
20. J. Anderson, *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-13, p. 799, 1983.
21. J.J. Hopfield, *Proc. Nat. Acad. of Sciences*, vol. 79, p. 2554, 1982.
22. K. Nakano, *IEEE Trans. on Systems, Man, and Cybernetics*, vol. SMC-2, p. 380, 1972.
23. J. Goodman, R. Dias, L. Woody, *Opt. Lett.*, vol. 2, p. 1, 1978.
24. E.G. Paek and D. Psaltis, to be published.
25. A. Vander Lugt, *IEEE Trans. Info. Theory*, IT-10:2, 1964.

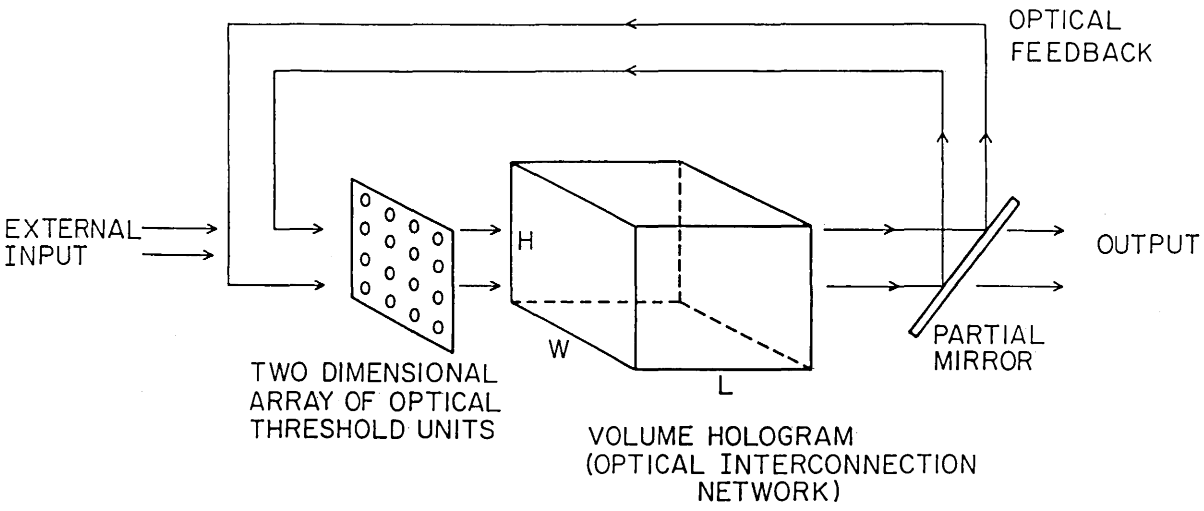


Figure 1. Optical Realization of a Neural Net Using a Volume Hologram.

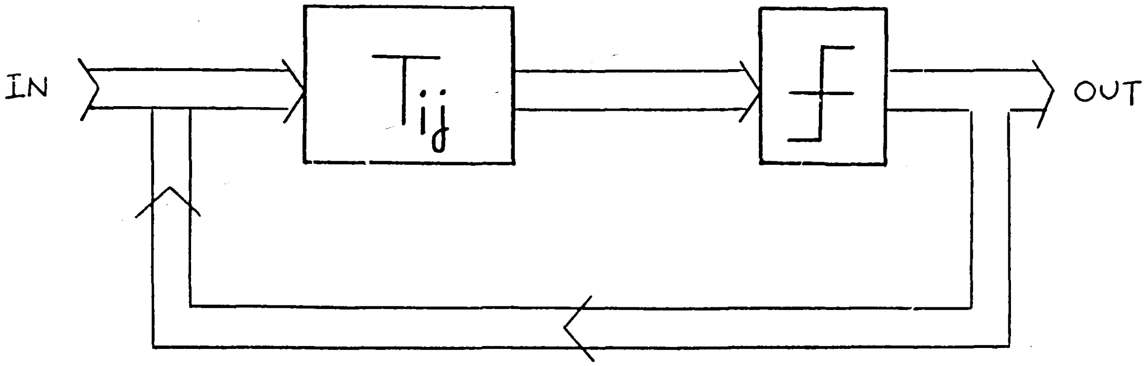


Figure 2. Neural Network Model of Associative Memory.

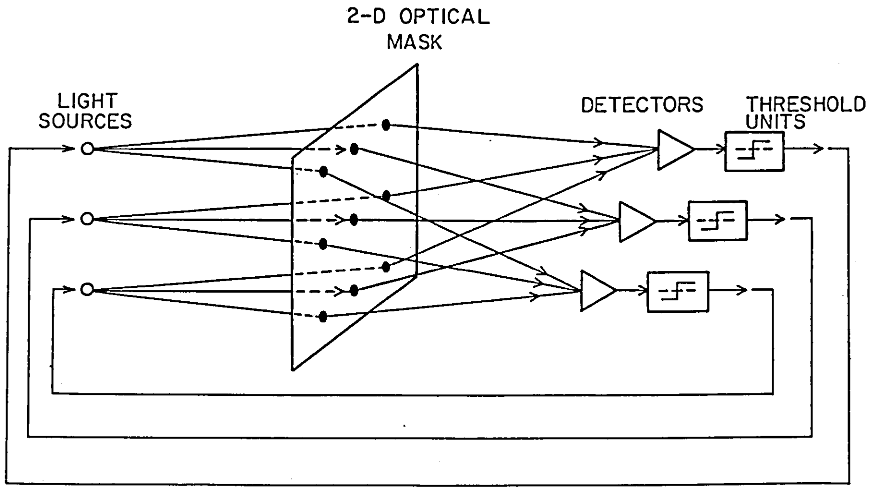


Figure 3. Optical Realization of Associative Memory using a Planar Transparency.

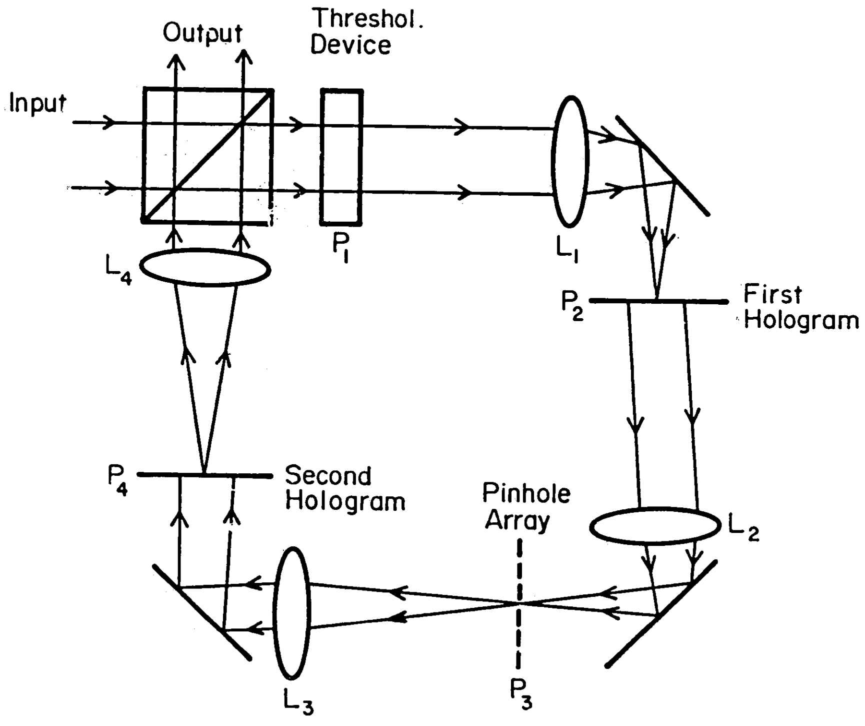


Figure 4. A Holographic Implementation of Associative Memory.

OPTICAL SYMBOLIC COMPUTING

Brian G. Kushner* - John A. Neff**

* The BDM Corporation, 7915 Jones Branch Drive, McLean, VA 22102-3396
** Defense Advanced Research Projects Agency, 1400 Wilson Boulevard
Arlington, VA 22209-2308

ABSTRACT

The need to drastically increase the processing rate of artificial intelligence (AI) systems, coupled with the limitations of current uniprocessor architectures, has resulted in a major research impetus to develop a new generation of parallel systems. Similar to a number of electronic symbolic computers being developed, optical computing systems can be viewed as a representative of the class of fine-grained, tightly-coupled architectures. This paper addresses potential roles of optical systems in symbolic computing and suggests future optical implementations of these architectures.

INTRODUCTION

The need to drastically increase the processing rate of artificial intelligence (AI) systems, coupled with the limitations of current uniprocessor architectures, has resulted in a major research impetus to explore parallelism in modern computing systems. Applications of these AI systems are placing stringent demands upon the underlying computer architectures. To meet these challenges, a new class of parallel computer architectures are emerging, structures which seek to optimize the processing and retrievel of symbolic information.

One promising group of AI-driven architectures can be described as tightly-coupled, fine-grained multiprocessor systems, and are characterized by both a dependence on complex interprocessor communications and flexibility in the interconnect topology.[1] This means that the architectures are composed of a large number of similar, relatively noncomplex processors,[2] which are coupled in such a manner that given processor can communicate with any other. The driving features behind this new generation of computer architectures are equally applicable to optical computing systems, which are also characterized by tightly coupled, finegrained elements with a high degree of communications flexibility. Optical systems also provide a form of two-dimensional parallelism which appears are attractive for symbolic processing operations. At the present time, we believe that optical systems can be adapted to the types of operations and data structures encounted in AI, and offer the promise of enhanced computational throughput to overcome bottlenecks in existing AI systems.

This paper will first describe some of the aspects of symbolic computing which drive the need for these new architectures. As a case in point, opto-electronic interconnnects are already being investigated for interprocessor communications in computer architectures. Potential roles for optics in multiprocessor systems will then be addressed. The paper will conclude with a discussion of current efforts to utilize optics in symbolic computing, both at the interconnect and processor levels, along with proposed optical implementations which look promising for general purpose symbolic processing.

MULTIPROCESSOR SYSTEMS AND SYMBOLIC COMPUTING

Symbolic processing is typically refers to on going research and development in four functional areas: speech recognition, vision or image understanding, natural language understanding, and expert systems. All four disciplines are characterized by the following three attributes: symbolic representations of knowledge; a high degree of interaction with a stored grouping of symbolic knowledge; and some level of reasoning capability, which draws conclusions by comparing inputs to the system with elements retrieved from the knowledge base. At the present time, each of these factors, retrieval of knowledge, representation of knowledge and reasoning, limits the computational throughput rate of AI systems. The remainder of this section will use expert systems as an example of a symbolic processing domain.

An expert system is a machine with mimics or emulates the thought and reasoning processes

CH2345-7/86/0000/0434$01.00 © 1986 IEEE

of human expert. It seeks to utilize the solution techniques which a human expert in a given discipline would use to solve particular problem in that domain. Knowledge appropriate to the discipline is placed into the machine, forming what is known as the knowledge base, enabling the system to understand the problem. Besides this knowledge base, the expert system consists of the "reasoner" or "inference engine", which manipulates the symbolic information stored in the knowledge base so as to "infer" new knowledge on its road to deriving a conclusion to a particular problem.

Retrieval in expert systems may be considered at four hierarchical levels: search of unorganized data, search of organized data, content addressing, and heuristic searching. The amount of symbolic information stored in a practical knowledge base is too large to even consider an exhaustive search of unorganized data. Considerable improvement can be realized by organizing the data to form data bases, and current expert systems make extensive use of data base management techniques to facilitate the retrieval process. But this technology is still based on von Neumann architectures which greatly hinder both the management and access to data bases of the size that will be needed for expert systems of the future. This problem has led to considerable interest in content addressable memory implementations of artificial neural systems and in logic-enhanced, or smart, memories for accomplishing heuristic searches.

Both the artificial neural systems and the logicenhanced memories fall into the category of tightly-coupled, fine-grained architectures. That is, they consist of thousands (or even millions) of switching or processing nodes (finegrained) which are interconnected to a high degree (tightly-coupled). The nodes of the neural network are just memory elements, but these elements are interconnected in a global fashion, so that processing associated with data retrieval may be distributed over all of the interconnected nodes. Although such networks will likely see use for symbolic computing, especially for associative recall of knowledge base information, neither the individual nodes nor the networks themselves have the processing power needed for advanced expert systems. Architectures with actual processing elements at each node represent the latest thinking of computer scientists dealing with artificial intelligence. These systems have been labeled with such names as logic-enhanced memories, smart memories, and connection machines.

Logic-enhanced memories avoid the von Neumann bottleneck by intermixing the processing and the memory funtions. This can be viewed either as distributing the memory among a large number of tightly-coupled processors, or as providing some processing capability to each element of a memory (hence the name logic-enhanced memory). This permits such powerful functions as interconnect reconfiguration and internodal relationship designation (e.g., for semantic network representations - to be discussed). These memories have such a broad processing power that they may be categorized as fine-grained, tightly-coupled multiprocessors.

Both the neural networks and the logic-enhanced memories are parallel processors; that is, any number of nodes or any of the interconnects can be active at any given time. It should also be noted that several optical implementations of neural networks and logic-enhanced memories have been developed, including associative processors[3,4] and memories incorporating a feature known as attention.[5] These systems all take advantage of the fine-grained nature of the optical devices and the types of global communications operations which are possible in optical computing systems.

A classification of parallel systems has been developed by Seitz,[6] a modified version of which is shown in figure 1. It provides an interesting categorization based on the number of processors and the relative degreee of processor complexity. Conventional uniprocessor architectures are plotted as a point of reference representing high complexity in a single processor. As one moves up to more than one processor, the trend is toward reduced complexity within each processor, a trend that is driven by total system cost and reliability, on the one hand, and by an escalating overall system complexity on the other hand.

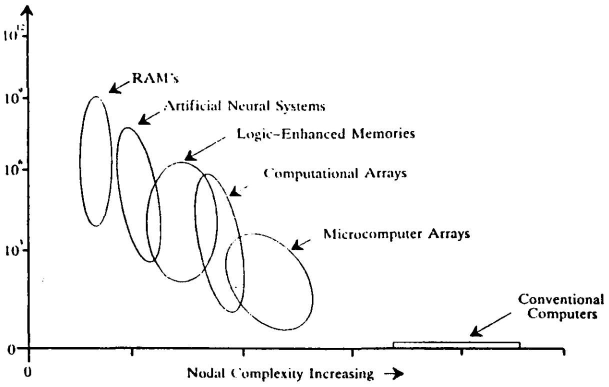


Figure 1
Classification of Parallel Processors by Nodal Complexity

Microcomputer arrays are basically a set of computers that send messages to one another via a communication network. Such systems are usually loosely-coupled; that is, the individual computers do not share main memory and I/O devices, although one computer can always draw upon another's resources through the communication network. The application of such systems in symbolic computing will likely be in solving problems that involve interactions between more than than one knowledge base. Each processor can work on a given part of the problem in such a way as to minimize the need for interprocessor communications.

Computational arrays are systems whose processing elements have been designed for tasks of comparable complexity to floating-point operations. Systolic arrays, for which the processors are connected in regular patterns that match the flow of data in the computations, compromise most of the architectures in this category.

As mentioned above, the architectures of most interest to knowledge base retrieval are the more fine-grained, tightly-coupled systems, such as logic-enhanced memories and neural networks.

Random access memories are shown in Figure 1 as a point of reference on the fine-grained ends just as uniprocessors were shown as a reference for nodal complexity.

To this point the discussion has centered on the knowledge base retrieval process so fundamental to expert system operation. The second major aspect of expert systems, as mentioned above,is knowledge representation. Figure 2 illustrates one popular method for representing knowledge, known as a semantic network. A semantic net may be characterized as a graphical representation scheme in which the graph nodes represent objects or concepts and the links represent inference procedures that relate the nodes. The importance and complexity of connectivity in these knowledge representation schemes has resulted in serious consideration of the tightly-coupled multiprocessor architectures for symbolic computing and expert systems implementations. The processing power at each node can, for example, be used to define the internodal relationships of the semantic network.

The similarity of these highly connected architectures to neurological systems lends credence to their importance in symbolic processing. We are very aware of the power of the brain in performing intelligent operations such as reasoning and pattern recognition, yet the brain consists of relatively slow switching elements. The biological switch is the neuron, and it operates in the millisecond range - about a milllion times slower than current electronic switching speeds. The difference lies in the large degreee of connectivity between biological switches, leading to a high degree of parallel processing. Neurons in the brain can have upwards of 10,000 synapses (biological connectors), whereas electronic switches in today's computers typically have only a few connections to other switches. There is strong evidence to suggest that the processing power of the brain is related to the high degree of connectivity between the neurons, permitting parallel processing and thereby compensating for the slow switching speeds.

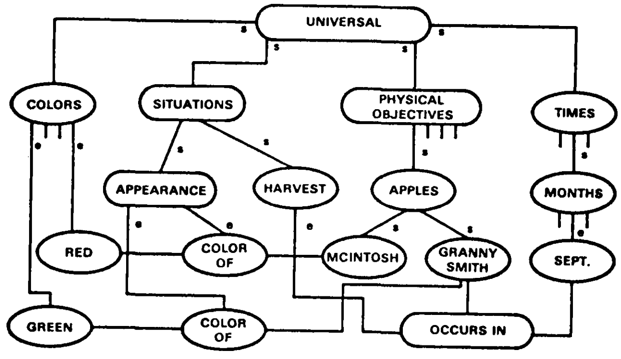


Figure 2
Semantic Networks

The next section will focus on potential roles for optics in multiprocessor systems. This will lead to a discussion of current efforts to couple optical computing with AI, and will conclude with two proposals for optical architectures which could significantly enhance the computational throughtput of fine-grained, tightlycoupled, multiprocessor systems. Such opticallybased systems, if realized, could open up a whole new field of opto-electronic computing directed toward artificial intelligence applications.

OPTICS AND MULTIPROCESSORS

Any approach to optical architectures must give serious consideration to what can be accomplished with existing technologies, namely VLSI electronics.
Optical computing will not seriously threaten electronic computing unless it can offer several orders of magnitude improvement in some critical measurement criterion, such as the power-speed-cost product, in a given problem domain. Therefore, a good starting point in addressing the application of optics to symbolic computing is to identify problem areas for electronics.

It is not surprising that the relative weaknesses and strengths of electronics and optics are traceable in one way or another to the fundamental physics of inter-electon and inter-photon interactions. Relatively speaking, the interaction between photons is weak; hence, electrons are good for the switching operations so fundamental to computing and photons are good for the interswitch communications, providing links which are free from detrimental coupling effects that lead to crosstalk and capacitive loading. Subscribing to such reasoning, however, is impractical due to the quantum losses which accompany both the electron-to-photon and the photon-to electron conversions. There is research and development underway to replace some of the longer interconnect links within computers with optical channels because it is the longer interconnects that create severe power, speed, and space problems for electronics[7]. But such a capability stops far short of using optics to its full advantage in multiprocessor architectures appropriate for symbolic computing.

Consider the electronic-switching/optical-communications positions as representing one of the four corners of the square shown in Figure 3. The sides of the square represent a continuum of combinations between the extremes of the corners. The upper left corner represents all-electronic systems while the bottom right represents all-optical. Since movement toward the bottom left corner not technically practical, the focus is along the upper and right sides. Upon considering computing systems for which switching is the predominant function, the trade-off between optics and electronics is seen to fall somewhere along the upper edge; that is, all electronic switching with some optical links. However, symbolic processing places a strong emphasis on connectively as was discussed above. The de-emphasis on switching (fine-grained architectures with low nodal complexity) and the emphasis on communications (tightly-coupled systems) leads one to consider architectures for which the communcations is optics and only some of the switching is done with electronics. It is this category of electronic/optical hybrid architectures that can have a significant impact on symbolic computing.

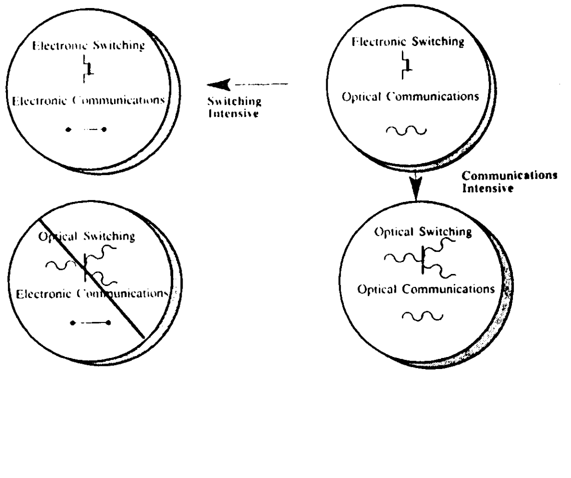


Figure 3
Electronic Versus Optical Computing

The upper right hand corner of figure 3 refers to the set of architectures which utilize optical switching for interconnect recongifuration and eletronic switching for logic operations.[7] Optics proves to be especially valuable in providing both the longer and the more global interconnects in processing, due to the combined power-speed-space-crosstalk penalties associated with electronic interconnects. Several examples of the use of optical interconnects for these in advanced architectures are currently under development, both in coarse and fine-grained computer structures. The interest in optics arises from the high bandwidth communications requirements between the individual nodes and the number of parallel channels which can effectively be multiplexed over a single optical link. The Texas Reconfigurable Array Computer (TRAC),[8] for example,is studying fiber optics for internodal board to board communications at bandwidths between 100 and 500 Mbps, and the WARP system[9] is investigating the use of optical interconnects for intercell communications at a rate of 0.5 1.0 Gbps and 32:1 multiplexing. Another example, that of a finegrained electronic symbolic computer under development, is the Connection Machine.[2] It is composed of 65,536 individual processing elements (PEs), organized as a large array of printed circuit boards, each of which contain 512 PEs equally divided between 32 chips. Both guided and unguided optical interconnects are being studied for overcoming challenges to this architecture, such as clock skew, 1.0 - 3.0 Gbps communication rates over 32:1 multiplexed links, and interprocessor broadcast.

Over the longer term, architectures such as that shown in Figure 4 could be developed to effectively integrate opto-electronic components in computer systems. Such a hybrid structure would use optics for most communications requirements and electronics for processing element functions. For the sake of simplicity, the illustration shows only two of the many possible boards and only four chips/board. If this were a fine-grained processor, each chip could contain many PEs.

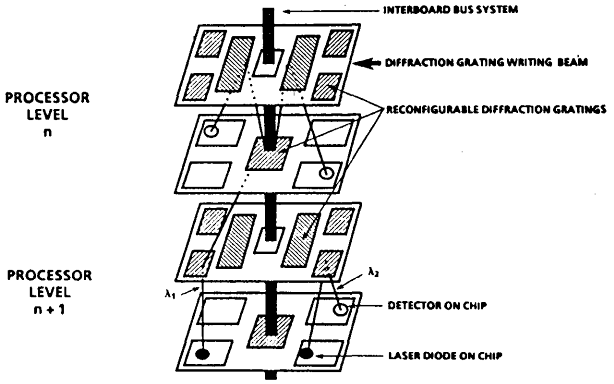


Figure 4
Hybrid Optical/Electronic
Multiprocessor Architecture

Each board in Figure 4 contrains four optoelectronic chips and one frequency selective filter (hologram). In between each board is a planar array of reconfigurable diffraction gratings, which perfom the majority of the switching operations involved in the interconnection process. This particular architecture employs wavelength division multiplexing (WDM) to direct optical bit streams to the appropriate board. The beam labeled illustrates this operation. The hologram directly above of the transmitting chip directs the beam to the center of the next board, where it is superimposed on the main beam which travels to all of the systems boards. Upon reaching the intended board, the frequency selective filter diffracts the beam to a bus-to-board hologram which directs the beam to its final destination.

The intra-board and intra-chip interconnects would be handled by the plane of holograms above the board, as illustrated by beam . The logistics of handling a large number of muliplexed beams will not be discussed here, other than to say that the optical switching most likely will be achieved through nonlinear wave mixing. For example, four wave mixing may be used to generate holograms[10] which can be rapibly varied to permit interconnect reconfiguration. Diffraction grating writing beams would contain the desired information for changing the holographic gratings. Note that some of the switching actions of such an architecture are being performed optically rather than electronically.

As one moves toward the bottom right corner of the classification scheme presented in Figure 3, the percentage of optical implementation increases until an all-optical architecture is achieved. While a number of efforts are underway to develop such all-optical structures, the research is currently directed at defining the appropriate computational primitives for optical symbolic processing. Thus the present focus is constrained to identifying and protoyping systems which can process the fundamental operations associated with symbolic computing--namely, the performance of correlation, searching, and pattern matching operations on symbolic data.

Typical of a class of research applications are the optical inference machines.[11,12] Here, the emphasis is on developing optoelectronic architectures which can perform the fundamental matching and logic operations encountered in retrieving symbolic data from a database. These sets of operations are both language and representation specific, so that architectures are being analyzed for several of the major different AI programming paradigms. Examples of other representations being investigated by optical computing researchers include semantic networks, graph theoretical representations, symbolic substitution for binary data,[13] and shadowcasting structures.[14]

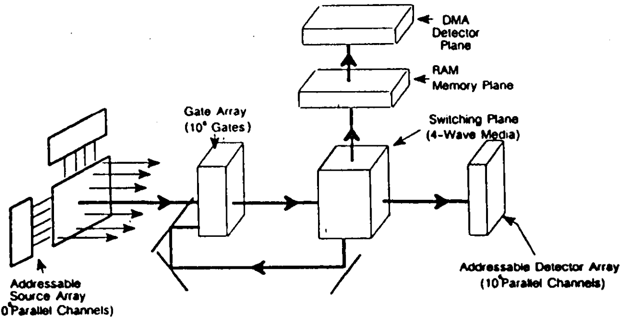

DMA Detector Plane
RAM Memory Plane
Gate Array (10^6 Gates)
Switching Plane (4-Wave Media)
Addressable Source Array (10^6 Parallel Channels)
Addressable Detector Array (10^6 Parallel Channels)

Figure 5
All-Optical Multiprocessor Architecture

An example of a fine-grained, tightly-coupled optical symbolic computer of the future is shown schematically in figure 5.[7] Although no one has built such a computer, it is technically believable to achieve such a system consisting of 1 million parallel channels. This does not mean that the system would be configured necessarily with 1 million nodes, since such this implies that the planar array of logic elements (designated as the gate array) would have just one logic element per channel. Instead, several logic elements would usually be interconnected via the interconnect media to form a processing element. For example a square array of n x n logic elements (gates) may comprise an arithmetic logic unit, several registers, and possibly some cache memory. An example of this type of structure is shown in figure 6, where individual elements in a 2-D SLM have been assigned the necessary functions to comprise a computational processing element. Taking an n of 5 (25 logic elements/processor) would lead to a machine with 40,000 nodes large enough to be practical as a symbolic computer.

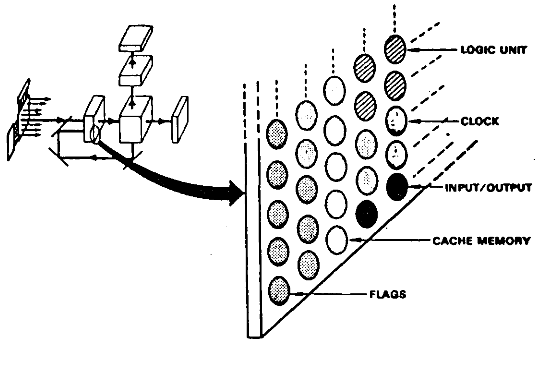


Figure 6
An All-Optical Processing Element
(Detail of Figure 5)

Input to the optical computer could be via either an array of independently addressable laser diodes or a two-dimensional spatial light modulator (2D SLM). The diode array would be capable of much higher modulation speeds, but would involve more complex circuitry, especially if operation requires uniformity over the complete array. If the input already exists as a two dimensional light pattern, such as might be output from a vision processor, an input device may not be needed (depending on the compatiblility of the two processors).

The logic element array could be either a 2D SLM exhibiting a nonlinear response or an array of optical bistable switches. The latter device will ultimately lead to much higher switching speeds, but current realizations of optical bistable switches require impractical power levels. Improved nonlinear optical materials are needed to achieve widely utilized optical bistable devices.

The interconnect element will likely employ wave mixing in a nonlinear optical medium, similar in operation to that mentioned previously fo the hybrid architecture. However, due to the much larger number of channels that must be handled, the switching may be done in a multistage fashion, in which multiple parallel planes of real-time hologram arrays would be exercised.

The detector will be a major technological challenge. In the most general case, one would like a one million channel device, with each channel operating around 1 MHz (projected speed for 2D SLMs). However, the requirements will be much less for most practical processor designs. If the problem domain were to require, say, 100 iterations or more (e.g., semantic network searches to depths of at least 100), an output would be required only once every 100 microseconds. This reduces the throughput rate of the detector to 10^{10}, a number more in line with projections for GaAs microelectronics. Another example would be where each processor consists of a block of n x n channels as discussed above. Assuming an n equal to 4 and that each processor has just one output channel, the throughput requirement of the detector would be 6.25 x 10^{10}. Some combination of these two designs should yield a detector requirement that would be well within near technical feasibility.

The last major component of this all-optical architecture is the memory. The practice in electronics of co-locating some of the memory with the logic elements cannot necessarily be transferred to the optical computing domain because of the greatly reduced communications delays. Thus, Figure 5 shows the main memory as the single block, equally shared by all of the processors.

CONCLUSION

The ultimate objective of artificial intelligence is to expand the power and reasoning processes in computing machines, allowing these machines to emulate and achieve capabilities typically associated with intelligent behavior in humans. However, efforts to achieve these goals have been severely constrained by today's serial architectures and by the separation of the processing and memory functions. Fine-grained, tightly-coupled multiprocessors appear to be a viable class of architectures for symbolic processing. Optical techniques, in the form of opto-electronic interconnects and optical computing, are being investigated as a means of alleviating computational bottlenecks in these systems.

Opto-electronics may play a major role in making these systems a reality, either as a supplement to an existing architecture or as part of an integrated opto-electronic computer. Research to date has demonstrated the viability of optical interconnects, and a number of efforts are underway to incorporate opto-electronics into existing architectures. At another level, the potential exists for developing all-optical symbolic processors as part of a larger scale computational environment. This appears particularly promising, since the 2D SLMs are in fact fine-grained PEs in a tightly-coupled environment. Such an all-optical system could serve as a co-processor in symbolic processing systems, and research is underway to identify the appropriate computational primitives and compatible representations.

Interestingly enough, at the present time, the numeric "supercomputers" execute AI functions at a greater rate than dedicated AI machines.[15] This may imply that raw speed is an important component in overcoming existing AI computational bottlenecks. It also may indicate that architectures designed to improve numeric throughput may also be useful in symbolic computation, and vice versa. Regardless of which of these paths are developed, the utilization of optics in symbolic processing represents an exciting new direction for optical computing.

REFERENCES

(1) A. Gupta, C. Forgy, A. Newell, and R. Wedig, "Parallel Algorithms and Architectures for Rule Based Systems," Proceedings of the 13th Annual International Symposium on Computer Architecture, (1986)

(2) W. D. Hillis, The Connection Machine, MIT Press, Cambridge, MA (1985)

(3) A. D. Fisher, C. L. Giles, and J. N. Lee, "Associative Processing Architectures for Optical Computing," J.Opt.Soc.Am.A. 1 (1984) 133

(4) D. Psaltis and N. Farhat, "Optical Information Processing Models of Neural Networks with Thresholding and Feedback," Optics Letters Vol 10 (1985) 98

(5) R. A. Athale, C. B. Friedlander, and B. G. Kushner, "Attentive Associative Architectures and Their Implications to Optical Computing," Proceedings of the SPIE, Vol. 625, (1986)

(6) C. L. Seitz, "Concurrent VLSI Architectures," IEEE Transactions on Computers, Vol. C-33.12 (1984)

(7) J. A. Neff and B. G. Kushner, "Optics and Symbolic Computing," in Optical and Symbolic Computing, R. Arrathoon, Ed., Marcel-Dekker, New York (to be published January 1987)

(8) G. J. Lipovsky, Texas Reconfigurable Array Computer, University of Texas Technical Report, (1984)

(9) E. Anould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, K. Sarocky, and J. A. Webb, "WARP Architecture and Implementation," Proceedings of the 13th Annual Interantional Symposium on Computer Architecture, (May 1986)

(10) D. M. Pepper, "Nonlinear Optical Phase Conjugation," Optical Engineering, 21, (1982) 156

(11) G. Eichmann and H. J. Caulfield, "Optical Learning (Inference) Machines," Applied Optics, Vol. 24.14 (1985) 2051

(12) C. Warde and J. Kottas, "Hybrid Optical Inference Machines: Architectural Considerations," Applied Optics, Vol. 25.6 (1985) 940

(13) K. Brenneer and A. Huang, "An Optical Processor Based on Symbolic Subtitution, " Technical Digest of OSA Topical Meeting on Optical Computing, (March 1985)

(14) J. Tanida and Y. Ichioka, "Optical Logic Array Processor," Proceeding of the 10th International Optical Computing Conference, (1983)

(15) R. P. Gabriel, Performance and Evaluation of LISP Systems, MIT Press, Cambridge, MA(1985)

An extendable optically interconnected parallel computer.

Alastair D. McAulay

Texas Instruments, Computer Science Center,
P.O Box 226015, MS 238, Dallas, TX 75266.

ABSTRACT

Desired features for a high performance machine are listed and include extendability, flexibility, and reliability. These features are hard to satisfy simultaneously with almost all existing and proposed parallel machines. A previously proposed optical crossbar interconnected processor is reviewed. It has properties that enable extension to larger systems by the addition of more processors, crossbar switches and exchange switches. These properties include: fine granularity, high communciation bandwidths, and reconfigurability. A method of doubling the size of the system without losing performance is described. It would appear that doubling may be repeated recursively until physical or other constraints arise. The performance of the resulting machine and the degree to which it satisfies the desired features are discussed.

INTRODUCTION

Desired features for multiprocessor

A high performance multiprocessor is desired that is extendable to more processors with a corresponding increase in performance. It must also be flexible and reliable. Extendability implies that more processors may be added to the multiprocessor together with corresponding interconnections without requiring new software and with performance approaching linear improvements with increasing number of processors. This will enable the same architecture and associated software to apply to a wide product range and provide longer life for customers and products. This implies that high performance versions of the machine must be built initially and that the machine must be capable of further extension to satisfy demand for many decades.

High performance involves considerations of throughput and latency. High throughput is suitable for many large problems because repetitive computations are often required. However, minimum latency is also required because of those situations where results are needed before subsequent computations can be performed. Flexibility implies that a wide range of algorithms must run efficiently. Also, new algorithms and as yet undiscovered algorithms must be easily entered into the machine and run efficiently. Reliability is required in any complex system.

Difficulties of achieving desired features

Parallelism is required to meet the requirements for future computer performance, even with the fastest technology components[17]. The number of parallel processors that may be used efficiently is limited in today's prototype and proposed systems by the communication delay and interconnection complexity. These systems are not generally extendable. I showed earlier, by analysis of a 3-D finite element computation, that attaching processors to a bus is straightforward but not extendable. Efficiency fell to 7% for 32 processors connected to a 37 MHz bus, where each processor has a computation rate of 10 Mflops[16].

Flexibility to efficiently run a wide range of algorithms may be achieved by reconfigurability[3],[20]. Reconfigurability also enables software to readily adjust to an extended system. However, reconfigurability introduces large delays and high control overhead in most proposed systems. This severely restricts the number of processors that may operate efficiently. Also, the throughput of the system is reduced and the latency increased.

Systolic configurations use nearest neighbor connections and prearrange dataflow so that input and output occurs at the edges of the processor array at each cycle[5]. Latency is increased relative to serial machines and this causes difficulties for general purpose numeric and symbolic computing. Also signal processing often involves situations that are adaptive or require rapid response. The number of elements in an array may be increased without altering the bandwidth at an input or output connection. The nearest neighbor connections also provide fast communication. This permits large numbers of processors to be efficiently used in parallel, thus providing high throughput and extendablity. However, the nearest neighbor connections limits the flexibility or range of algorithms that may be implemented efficiently. For example, many fast algorithms use doubling which results in

CH2345-7/86/0000/0441$01.00 © 1986 IEEE

complex non nearest neighbor communications, e.g. FFT, recursive doubling.

Extendability to larger high performance systems increases the likelihood of interference because of longer cables. This reduces the reliability. Reliability is often accomplished by means of redundancy in software, hardware, time and/or space. This is detrimental to satisfying performance for a given cost. It also limits extendability.

REVIEW OF SINGLE CROSSBAR SYSTEM

Prior research

An optically interconnected system was selected after examining a number of different architectures[17], including systems performing optical computation[11]. I previously showed how to efficiently implement basic signal processing algorithms such as an FFT, systolic filter, and matrix-vector multiplier on the proposed optically interconnected processor[12]. The crossbar switch permits fast algorithms, such as those involving doubling, to be implemented more efficiently than on a nearest neighbor or systolic array[5]. The improved performance justifies the cost of the switch. Previous investigations illustrated the implementation of Levinson's algorithm for optimal least square filtering[6], the implementation of Levinson-Durbin's, Burg's and Schur's algorithms[10], and texture classification[7], and the implementation of conjugate-gradients[9]. The crossbar switch makes possible close coupling between symbolic and numeric processing[8].

System

Figure 1 shows a preliminary organizational structure for an optical crossbar signal processor[12],[15]. Hundreds of processing elements, P_i, $i = 1$ *to* $N/4$, are connected to an optical switch of size $N/2$ by $N/2$ by means of commercially available fiber optic links of bandwidth 160 MHz or more. The processors perform elementary operations such as multiply or add and therefore have two input connections for the two operands. This fine granularity permits the maximum amount of parallelism to be extracted from algorithms. The processing element output is converted from parallel to serial in a shift register and used to drive a laser diode. A second output is provided for convenience. The laser diodes are connected via optical fiber links to the optical crossbar switch. Fibers returning from the switch connect to light sensors at the processor inputs. A second fiber optic loop between processors and main memory banks provides input/output. The logic permitting input/output, marked **w** and **v**, may be mounted alongside the processor.

Optical switch and interconnections

Optics is used for interconnections because photons are inactive, having non interfering characteristics. Also, optics provides sufficient bandwidth to support fine granularity parallel processors so that high levels of algorithm parallelism may be extracted.

Each intersection in a crossbar switch, figure 2a, has a switch permitting a horizontal input line to be coupled with a vertical output one. One output receives information from one input but one input may broadcast to several outputs in a *generalized* crossbar switch. Figure 2b shows a diagrammatic crossbar switch implemented with a spatial light modulator (SLM) and dots indicate transparent regions consistent with the closed switch settings marked by dots in Figure 2a. An optical lens system is

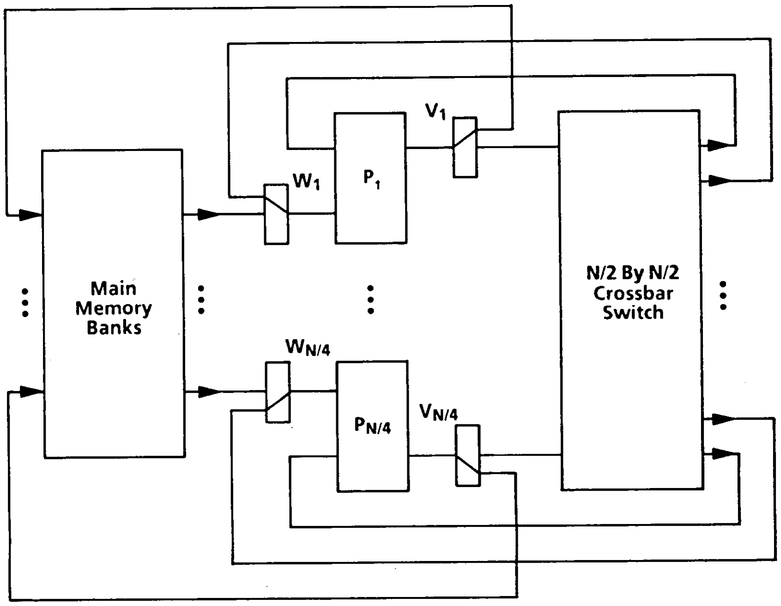


Figure 1: **Single crossbar system**

used to spread the light from the input sources horizontally without spreading the light vertically. Light passing through the spatial light modulator is collapsed onto receiving diodes by means of a lens system which focusses vertically without spreading horizontally. In addition to being a directional generalized crossbar switch, the optical switch permits *inclusive OR* operations by allowing different inputs to feed an output.

A reflecting membrane covers the surface of an array of transistors on a silicon chip, figure 3, to form a deformable mirror device (DMD). Figure 2b showed a transmissive spatial light modulator. The DMD is reflective and requires a beam splitter to separate the reflected light from the incident light. The mirrors are drawn as though they are transmissive light modulators for clarity. Activation of a transistor causes the membrane to dip above the transistor. A collimated beam of light striking the DMD will result in concentrations of light just above each of the activated transistors. This array of light and dark pixels is imaged to another plane for viewing or further operation. A Schlieren system uses a stop in the frequency domain to remove low spatial frequency light arising from reflection from regions between mirror deflection pixels. Imaging and performing spectral analysis with a Texas Instrument's DMD of size 128 by 128 has been published[18]. It takes one microsecond per row to load the DMD while still using the previous deflections. A few microseconds are required to switch to the new settings.

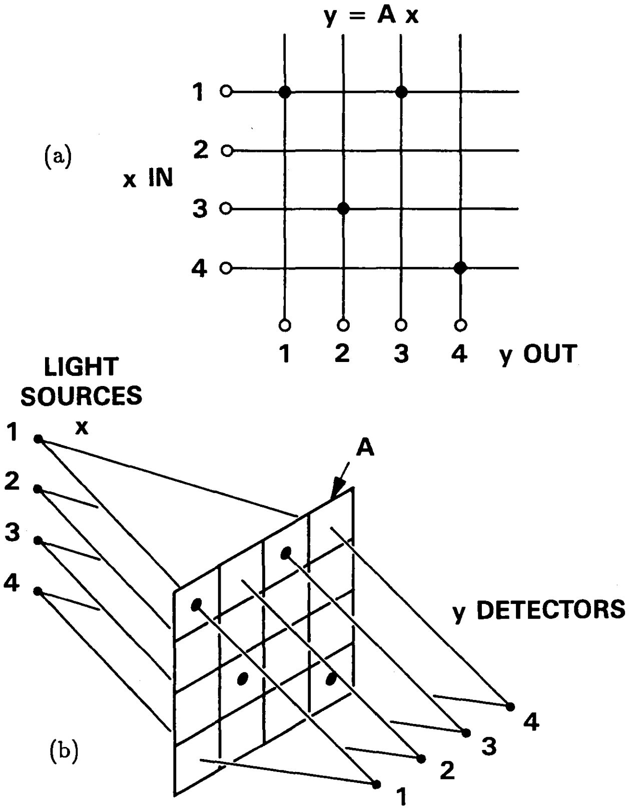


Figure 2: **Diagrammatic optical crossbar switch**

(a) Switch settings
(b) Spatial light modulator crossbar switch

Commercially available fiber optic links are available at the transmission rates required to keep up with available processing elements. It is anticipated that faster links will be commercially available as the processing element speeds increase. Optical links increase the immunity to interference relative to using cables.

Software and programming approach

The flow of data is prearranged so as to minimize run time overhead[7]. A static dataflow approach is used to minimize run time overhead for signal processing algorithms. The use of a high throughput crossbar switch will make it possible to map algorithm graphs onto the system more easily and more efficiently than onto constrained interconnection networks such as hypercube[19] and multistage networks[4]. This suggests that automatic mapping will be feasible for our machine. The throughput for a wide range of algorithms is also expected to be higher than for other architectures, including current prototype message passing dataflow machines[1].

Figure 4 illustrates an executable flow graph for the Levinson-Durbin algorithm unrolled for pipelining[7]. Nodes marked with subtraction imply the subtraction of the right hand input from the left hand input. Flow is down the page. The triangular arrows indicate negation or unary minus which may be accomplished at the input to the appropriate node rather than with an extra node. Identity instructions marked "Ident" support the fanout of operands so that an input connects to a processing el-

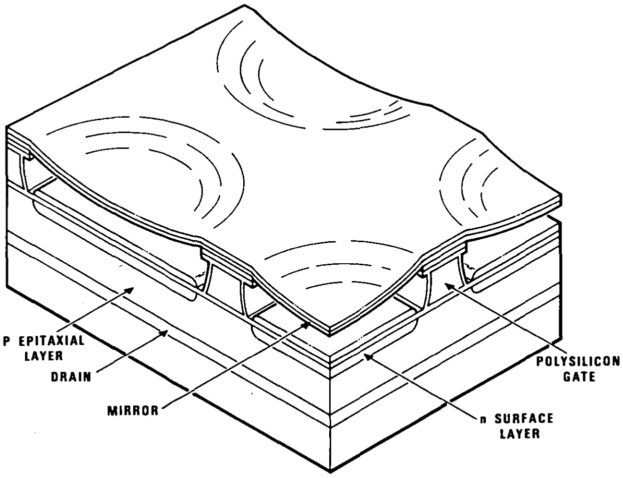


Figure 3: **Deformable mirror device**

ement before going through the crossbar. Each operator must forward its results on every clock. Consequently, unit delays are inserted on any edge of the graph which "crosses a level" without being used. The number of unit delays is a number at the input to an instruction indicating the queue at an existing node. Parallelism is evident in the flow graph by the number of nodes that occur side by side. A 100% efficiency is achieved once the pipeline is filled[7].

The flow graph is mapped directly to the machine without coding in a higher level language. However, as many algorithms exist in coded form, it is desirable to be able to create flow graphs from the code and display the graph. The level of parallelism may be observed and algorithm changes made to increase the parallelism[9]. Nodes in the flow graph are assigned to processing elements in the system and links in the flow graph to settings of the crossbar switch. A compiler is being developed for this mapping. Data flows into the switch during operation and is routed to the appropriate processor. A processor will perform the operation for which it is programmed on the next clock cycle after receiving its operands. The output is routed via the switch to the next processor.

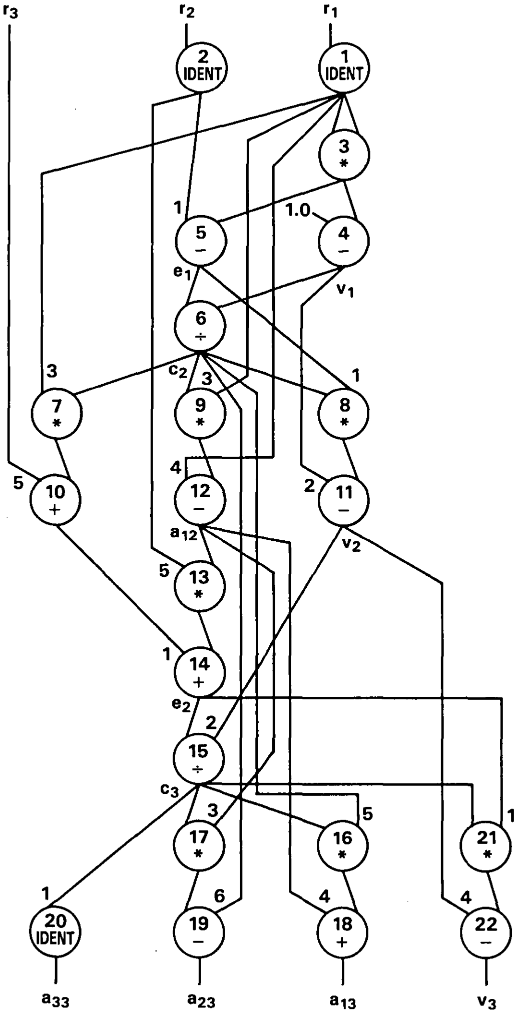


Figure 4: **Flow graph illustration**

EXTENDED CROSSBAR SYSTEM

Illustration of a double-crossbar system

A specific approach is illustrated for extending the single crossbar system described previously to double the size. Alternatives are considered in the next section. A second identical single-crossbar system is placed next to the first, figure 5. The optical fiber connections into and out of the crossbar switches are opened and exchange switches **s** and **u** inserted respectively.

The switches that implement the exchange switch at the input to the crossbar are marked s_1 through $s_{N/2}$. The second part of each switch is marked s_1' through $s_{N/2}'$. Consider the activity between the input to a switch, say s_1 and s_1', and the input to the crossbar switches. The effect is that of a 2 by 2 switch for which the two inputs are either connected directly to the two outputs or in an exchanged configuration. An example is provided by the top output for processor P_1 and the top output of processor $P_{N/4+1}$. If P_1 is connected to the top input of the top crossbar switch then $P_{N/4+1}$ is connected to the top input of the lower crossbar switch. In the exchanged position (as shown in figure 5) P_1 is connected to the top input of the lower crossbar switch and $P_{N/4+1}$ is connected to the top input of the upper crossbar switch.

An exchange switch, u_1 through $u_{N/2}$, is similarly incorporated between the output of the crossbar switches and the input to the processing elements. An example is provided when the top output from the upper crossbar switch is connected to output 1 at the right, then the top output of the lower crossbar switch is connected to output $N/2 + 1$ at the right side. In the exchange position, the top output of the upper crossbar switch is connected to output $N/2 + 1$ at the right and the top output of the lower crossbar switch is connected to output 1 at the right.

An electro-optic switch may be used to implement the exchange switch **s** by directing light into one of two fiber channels. A combining coupler is used at the input to the crossbar switch to permit light to pass from either channel. The switches **v** may be implemented with similar electro-optic cells. Similarly, the output exchanges switches **u** may be implemented using a reverse electro-optic cell. Only one of the input light channels is permitted to generate an electronic signal. The electronic signal is then converted to light for transmission back to the processor.

Interconnection alternatives for extendablility

Alternative approaches to extendability are considered. Four crossbar switches of size $N/2$ by $N/2$ may be used to construct a double size N by N crossbar switch system. $N/2$ inputs are connected in parallel into crossbar switches one and two and the other $N/2$ inputs are connected in parallel into switches three and four. The outputs of switches one and three are connected in parallel to produce $N/2$ outputs and the outputs of switches two and four are connected in parallel to produce the other $N/2$ outputs. OR capability is maintained.

An alternative is to use only two $N/2$ by $N/2$ switches together with two sets of $N/2$ switches[2]. This increases control complexity but uses less switches than the four crossbar switch system. If two by two optical switches, such as DMDs, are used, for which OR operations are possible, all the original features of the optical switch are maintained for the larger size. Hence, from a software point of view the system is identical to the previous single-crossbar system, except for doubling in size. The resulting system will lack OR capability if generalized two by two switches are used with no OR capability. Futher, some broadcast capability is lost if exchange switches are used permitting only permutations and not broadcasting. For example, the system now permits inputs to broadcast to only half the outputs. This case was described in the last section. Only one control bit rather than two is required for control of the two by two switches. The crossbar switch is still complete or nonblocking and provides N^2 well defined mappings from input to output.

If we omit the exchange switches at the crossbar output, **u**, it is still possible to reach any output from any input. However, blocking may now occur and the total available mappings are only N! This is illustrated using Figure 5. Suppose it is desired to connect via the crossbar

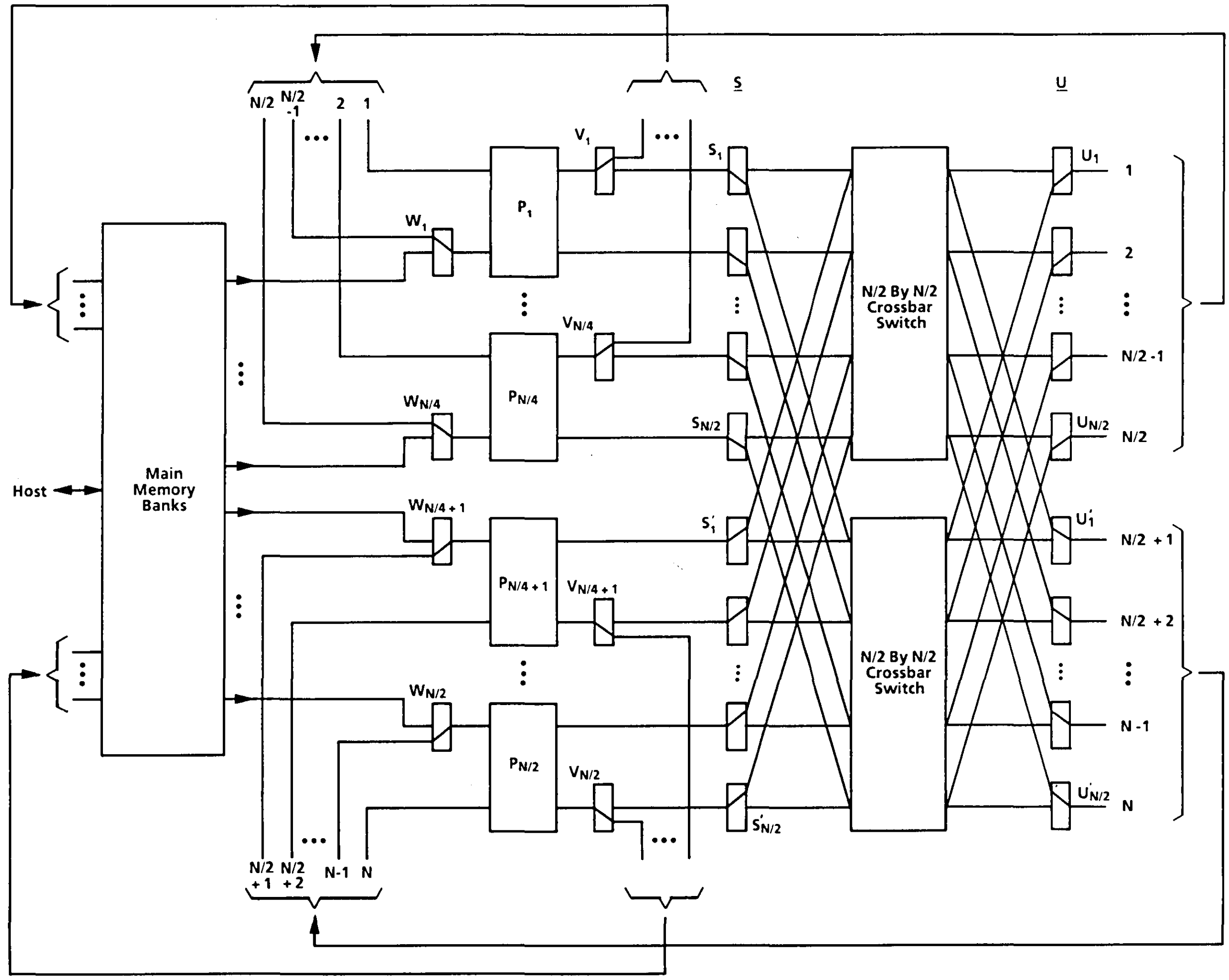


Figure 5: **Double crossbar system**

switches the top output of processor P_1 to the input of processor $P_{N/2}$ and at the same time the top output of processor $P_{N/4+1}$ to the input of processor $P_{N/4+1}$. The first connection may by accomplished be setting switch s_1 to the downward switch position (as shown in Figure 5) and $u'_{N/2}$ to the downward switch position (opposite to that shown in Figure 5). Consequently, the other half of the first switch s'_1 is set upward. In order to accomplish the second connection u'_1 must be set upward. Absence of this latter switch would result in blocking of this second connection by the first.

The system may be doubled in size again by placing a second system like that in figure 5 beside the first. The inputs to switches **s** and outputs from exchange switches **u** are now broken and further exchange switches inserted. Doubling may be applied recursively until physical constraints arise. Depending on the properties of the two by two switches some features relative to a single larger DMD optical switch may be increasingly lost as the amount of switching in the two by two switches increases relative to that in the optical DMD switches. The complete or nonblocking feature is maintained.

Finding, isolating and correcting faults

The use of multiple crossbar switches permits a degree of fault tolerance. Diagnostic programs and data may be run through the system periodically to verify correct operation. Failure to respond correctly to diagnostics is followed by testing each crossbar separately with the same diagnostics. Testing with the exchange switches interchanged will further determine whether processors or switches are at fault. In the former case reconfiguration will be made with the offending processing element disconnected. In the latter case, the system performance will be reduced if the switch is no longer permitted during reconfiguration. Repair of offending processors or switches should be possible without stopping the system.

PERFORMANCE ASSESSMENTS

Did the system meet desired features?

Sections of a computation often involve global communication or are not divisible into sufficiently large parallel pieces for efficient computation on a parallel machine. This becomes more of a problem as the number of processors is increased because the parallelizable parts are performed faster and the difficult parts become more of a bottleneck. Fine granularity and the broadcast capability of the crossbar switch permit these difficult parts to run efficiently on the proposed optically interconnected system. Consequently, the crossbar switch system may be extended to more processors without this difficulty.

Optical interconnections are needed to provide the high bandwidth required for fine granularity parallelism. Photons are inert compared to electrons, therefore the proposed sysem has a higher immunity to interference than electronic systems. This assists in extendability where distances to new processors and memory are likely to be further than for a smaller number of initial processors. Greater immunity to interference is important for enhancing reliability. Packaging is simplified when components need not be very close.

The optical crossbar provides reconfigurability, in fact N^2 well defined links, between processing elements. Reconfigurability enables a wide range of algorithms to be efficiently implemented as is required to meet the flexibility requirement. The large number of alternative paths assists reliability because failure of a processing element degrades performance only slightly after reconfiguration. The processing element or switch may even be repaired without stopping the system. The system may be extended by adding more crossbar and tow by two switches to emulate larger crossbar switches. Consequently, the same software may be used after extension. The optical switch has a higher throughput than conventional electronic crossbar switches and does not require N^2 interconnections. It is easier to map algorithms onto a full crossbar switch than onto a constrained or reduced switch, thus making automatic mapping of algorithms by means of a compiler feasible. Otherwise a directed graph for an algorithm must be mapped onto a directed graph for a processor interconnection system.

Static dataflow permits prior arrangement of data flows. This significantly reduces run time overhead. Conventional architectures require the computation of addresses for the operands for an operation and then the finding of these addresses and the fetching of the data. An instruction must then be decoded, the operation performed and an address computed for storing the solution.

Comments on computing performance of extended processor

Previous research showed that some algorithms, such as conjugate gradients[13],[14] for solving large sets of linear equations, may be implemented with over 90% efficiency even when there is a mismatch between the number of processing elements and the dimension of the problem[9]. This is still true for an extended processor having more crossbar switches and processing elements because both communication and computation are increased in the same proportions. Rapid solution is not critical for problems much smaller than the machine unless there are a large number of them. In this case the processor may be reconfigured to provided pipelining or parallel computation of the small problems and will achieve very high efficiencies.

Algorithms, such as Levinson's[6],[10], that are iterative with increasing dimension and the number of iterations are determined during computation, are difficult to implement efficiently on parallel machines. The extended machine will still speed up computation in proportion to the incease in processors, no matter what the efficiency. Unrolling loops of the iteration provides faster speeds because control for these loops is no longer required at run time. This may be beneficial even if the occasional problem does not require all the loops. Tags are used in the architecture to indicate when processing elements should not operate on data for reasons such as this. Algorithms, such as Levinson's, may be implemented with 100% efficiency by pipelining when there are many cases of known dimension[7].

CONCLUSIONS

Desirable features for a high performance processor include extendablility, flexibility and reliability. These features are difficult to achieve at the same time as indicated. An optical crossbar interconnected processor proposed earlier was shown to be extendable by recursively doubling the system. Processing elements, switches and crossbar switches are added in the same proportions. The advantages of the system are carried through to the new system and proportional increases in performance with increasing number of processors is anticipated. Extendablility would not be possible without the specific features of the machine. Fine granularity enables parallelism to be extracted where other machines could not. Therefore, for such cases, the other machines would exhibit severe degradation in performance with increased numbers of processors. The high bandwidth optical interconnections make it possible to have fine granularity. Reconfigurability of the crossbar permits the extended system to operate with the same software. It also provides algorithm flexibility for adjusting to the change in the relative dimensions between problem and machine because of the changing number of processors. Reconfigurability, high levels of parallelism, multiple crossbar switches and optical links provide the opportunity for high reliability. This architecture is considered promising for future computers.

Acknowledgments

Office of Naval Research and DARPA support under contract N00014-85-C-0755 is gratefully acknowledged. I wish to thank Jeff Sampsell and Don Oxley of Texas Instruments for valuable discussions.

References

1 Arvind and R.A. Iannucci, " Two fundamental issues in multiprocessing: the dataflow solution," MIT Report, MIT/LCS/TM-241, 1983.

2 Benes V., Mathematical theory of connecting networks and telephone traffic, New York, Academic Press. 1965.

3 Browne J.C., "Parallel architectures for computer systems," Physics Today, Vol. **37**, No 5, May 1984.

4 D.Y. Cheng, " A floating point coprocessor for the Butterfly Multiprocessor System," SRC Technical report No. 059, Univ. Cal. Berkeley, 1984.

5 Kung H.T., "Why systolic architectures?," Computer, Vol. **15**, pp. 37-46. 1982.

6 McAulay A.D., "Optimal Least Square Filtering with a Digital Optically Interconnected Processor," SPIE Technical Symposium SE, Advances in Optical Information Processing II, Vol. **639**, 1986.

7 – "Image texture classification with an optical crossbar interconnected processor", Future Directions in Computer Architecture and Software, Army Research Office Workshop, 1986.

8 – "Optical Interconnections for Real Time Symbolic and Numeric Processing," Invited Chapter in Book on "Optical Computing: Digital and Symbolic", Marcel Dekker. NY. 1987.

9 – "Conjugate Gradients on Optical Crossbar Interconnected Multiprocessor," Submitted, Journal of Parallel and Distrubuted Processing. 1986.

10 – "Parallel AR Computation with a Reconfigurable Signal Processor," IEEE Internat. Conf. on Acoustics, Speech and Signal Processing, 86CH2243-4, Vol. **2**, 1986.

11 – " Deformable mirror nearest neighbor optical computer," Optical Eng., Vol. **25**, No. 1, 1986.

12 – "Optical Crossbar Interconnected Signal Processor with Basic Algorithms," Opt. Eng., Vol. **25**, pp. 82-90. 1986.

13 – " Plane-Layer Prestack Inversion in the Presence of Surface Reverberation," Geophysics, Vol. **51**, No. 9. 1986.

14 – " Prestack Inversion with Plane Layer Point Source Modeling," Geophysics, Vol. **50**, No. 1, pp. 77-89, 1985.

15 – "Optical Crossbar Signal Processor," Proceedings of SPIE, Real Time Signal Processing VIII, Vol. **564**, pp. 131-138. 1985.

16 – " Finite element computation on nearest neighbor connected machines," NASA Symposium on Advances and Trends in Structures and Dynamics, NASA Publn. **2335**, pp. 15-29, 1984.

17 – " The role of parallel computing," IEEE Region 5 Conf., IEEE Publin. 85CH2123-8, 1985.

18 D.R. Pape, and L.J. Hornbeck, "Characteristics of the deformable mirror device for optical information processing," Opt. Eng., Vol. **22**, pp. 675-681. 1983.

19 C.L. Seitz, " The Cosmic cube," Communications of the ACM, Vol. **28**, No. 1, pp. 22-23, 1985.

20 Siegel H.J., "Interconnection networks for large scale parallel processing, theory and case studies", Lexington Books, 1984.

OPTICAL INTERCONNECT TECHNOLOGY DEVELOPMENTS
(Invited Paper)

L. D. Hutcheson

CyberOptics Corporation
2331 University Avenue SE
Minneapolis, MN 55414

Abstract

Conventional interconnect and switching technology is rapidly becoming a critical issue in the realization of systems using high speed silicon and GaAs-based technologies. Optical interconnect technology promises to enhance performance, provide relief from the pinout problem, decrease implementation complexity, and provide improvements to the flexibility of systems by allowing real time reconfiguration of these systems. In recent years, rapid progress has been made in VLSI/VHSIC technology that improves on-chip density and speed while packaging these high speed chips is becoming extremely difficult and in some cases limiting system performance. By releasing the bandwidth constraints on interconnects and packaging, the full processing speed capabilities of silicon and GaAs logic can be exploited to dramatically improve system throughput. A number of University, governmental and industrial laboratories have been developing technology for on-chip/on-wafer, chip-to-chip and board-to-board high speed optical communication. Both guided wave and free space communication media are being developed. In this paper, a review of the technological developments, current status and future projections for optical interconnects will be presented.

Introduction

There is a growing demand to increase the throughput of high-speed processors and computers. To meet this demand, denser, higher-speed IC's and new computing architectures are being developed. Electrical interconnects and switching have been identified as bottlenecks to the throughput of computing systems. Two trends brought on by the need for faster computing systems have pushed the requirements on various levels of interconnects to the edge of what is possible with conventional electrical interconnects. The first trend is the development of higher speed and denser switching devices in silicon and GaAs. Switching speeds of logic devices are now exceeding speeds of 1.0 Gb/s, and high density integration has resulted in the need for interconnect technologies to handle hundreds of output pins. The second trend is the development of new architectures for increasing the parallelism, and hence, the throughput of a computing system.

A representation of processor and interconnect complexity for present and proposed computing architectures[1] is shown in Figure 1. The dimension along the axis is the number of processors required for the architecture. On the left end of the axis is the Von Neumann type architecture which has very few processors, but the processors tend to be very complex. Progressing to the right, the number of processors per system increases until it reaches a neural network requiring millions of processors but the processors are much less complex than the Von Neumann case. From looking at Figure 1, it becomes apparent that as the number of processors increases, the number and complexity of the interconnects within the system increases dramatically. In fact, at the far right of this scale, the interconnects become an integral part of the computing architecture, and the boundary between the processors and the interconnects becomes blurred.

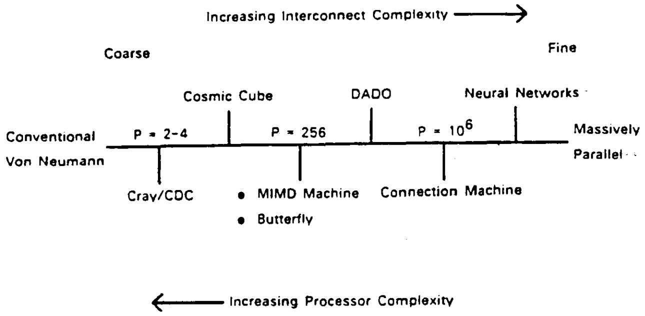


Figure 1 Granularity of Processing [Ref. 1].

Optical interconnects have long provided the telecommunications industry with a high speed, low weight, low volume, long distance transmission media[2]. Optical interconnects can also be applied, at least conceptually, at many levels within a computing architecture. On a physically large scale, optical interconnect technology has been used in local area networks to connect computers. At the next level down in physical size is the interconnection between systems within a computer such as a memory to processor connection. This is classified as a board-to-board or backplane interconnection. Next there are the interconnections between individual chips on a single board or between chips in a multi-chip package. Finally, there are the interconnections between devices within a single chip or on-chip clock distribution.

CH2345-7/86/0000/0448$01.00©1986 IEEE

A number of researchers have been developing the technology for optical routing techniques, packaging of optical and electronic components, high speed opto-electronic devices and optical clock distribution leading to optical interconnect demonstrations. A discussion of these technology developments, as well as some optical interconnect system demonstrations, will be the subject for the remainder of this paper.

Optical Routing Techniques

There are basically two types of routing techniques being explored - guided and free space. The guided interconnect technique could use optical fibers for distributing the optical signals or optical waveguides integrated on a substrate. Fibers for optical input/output (I/O) shows promise because of its well established technology, low cost and ease of coupling to the fibers. Since the distances to be travelled by the optical signals for applications discussed in this paper are short (<1 km), multi-mode fibers can be used even for data rates of several Gb/s. This will significantly ease the fiber to chip alignment and packaging constraints. A general scheme for using optical fibers[3] for I/O is shown in Figure 2. Fibers are sufficiently flexible and low volume to provide random routing with little crosstalk for backplane communications. For chip-to-chip interconnects on a single board or even intrachip communication, fibers become cumbersome. In this environment, fibers and their bending radius become large in comparison with typical electrical interconnects and the size of the devices. An alternative is to use integrated waveguides in the packaging to increase density and reliability.

The fiber optic routing scheme shown in Figure 2 shows two possible configurations. The first is a point-to-point communication link with a single optical source linked to a single receiver having a fanout of one. The second shows a multiple fanout with one source communicating to several detectors.

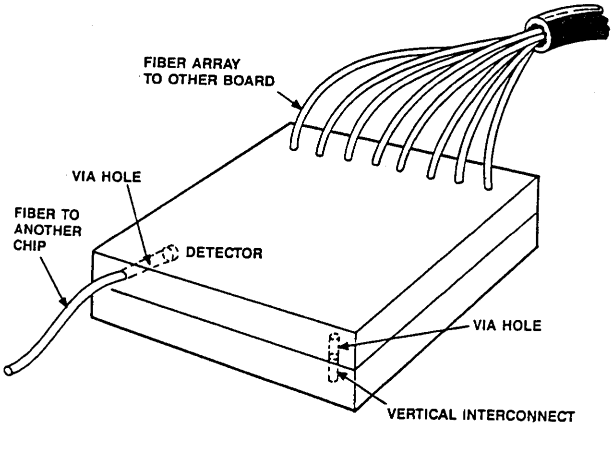


Figure 2 Scheme for using optical fibers for I/O showing a fanout (bottom) of one and multiple fanout (top) [Ref. 3].

In the area of high-speed communication busses, there is a practical limit for speed and fanout in implementing high speed electrical bus structures. These busses span printed wiring boards, between boards and between systems. The difficulty arises when an electrical interconnect is required to fanout to multiple distributed loads. As the number of fanouts increase, several parameters effect the electrical interconnect. The two most notable effects of fanout[4] are the lowering of the effective characteristic impedance and the increase of the propagation delay. While the increase in characteristic impedance is not fundamentally limiting since drivers can be designed to provide the required power, it does increase the required on chip power and decrease available chip real-estate, both of which are undesirable. The effect of the decrease in propagation delay decreases the critical line length. The critical line length is the maximum length that an electrical interconnect can be without termination, so that ringing on the circuit is minimized.

Using optical fibers as the transmission media, the effect on fanout is quite different. The number of fanouts for optical interconnects are limited primarily by the available power to the detectors. The actual power available to the detectors is determined by the power of the source and losses throughout the distribution system. Optical fanout is achieved by power splitting of the optical channel as shown in Figure 2, which can be achieved by star, tree or tap networks. The maximum fanout for an optical channel can be determined from the minimum power required by the detector (for a given bit error rate), optical source power and losses throughout the system. Figure 3 shows a comparison of optical and electrical fanouts versus clock frequency[4] for a bussing structure.

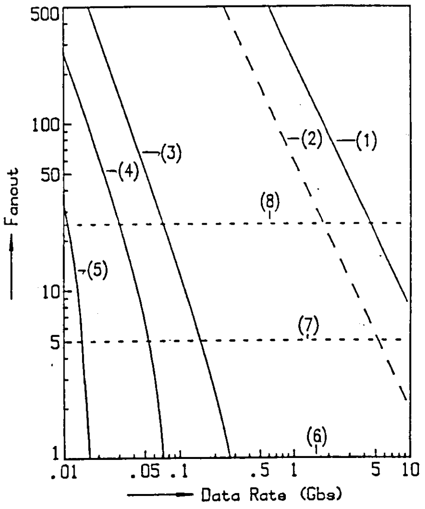


Figure 3 Optical and electrical fanout limitations versus data rate for various conditions (1) maximum theoretical optical fanout (2) practical limitation to optical interconnects assuming a 6dB design margin (3), (4) and (5) represent 2, 10 and 50 cm unterminated electrical lines respectively and (6), (7) and (8) represent 2, 10 and 50 cm terminated electrical lines respectively [Ref. 4].

Using un-guided or free-space techniques for directing the light to its destination is a method that shows a lot of promise. The use of free space allows the density of the interconnects to increase to the fundamental diffraction limit of optics with minimal crosstalk. One can use either unfocused or focused free space interconnects. The unfocused technique simply broadcasts the optical signals carrying the information over the entire electronic chip. However, this type of interconnect is inefficient since only a small portion of the optical energy falls on the photodetectors and the rest is wasted. Also, optical energy incident on portions of the chip where it is not wanted may induce stray electronic signals that cause errors and improper operation.

Using holographic techniques as shown in Figure 4, the optical source can be imaged onto a multitude of detector sites simultaneously[5]. This should significantly increase efficiency and reduce stray electronic signals, provided the hologram can be made with large diffraction efficiency. The primary disadvantage of using focused interconnect techniques is the high alignment precision that is needed to assure that the focused spots are striking the appropriate places on the chip[6]. A number of holographic imaging techniques for optical interconnects has been investigated by Goodman[7] et al. The configuration that appears to have the most flexibility is shown in Figure 5, since it can have a large number of independent channels and spatially variable fanout[7]. In this proposed technique, each facet is illuminated sequentially with a number of diverging object beams and a converging reference wave. The positions of the object beams can be moved automatically with a computer controlled stepper motor drive and the beam ratios can be adjusted to provide optimized diffraction efficiency.

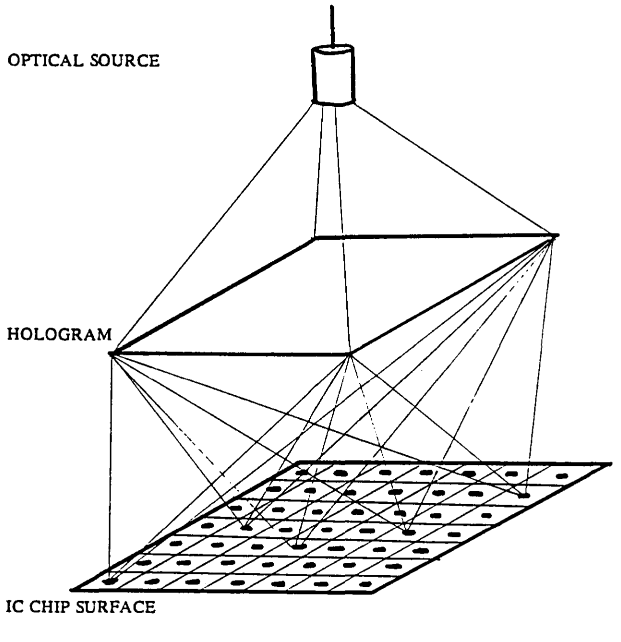


Figure 4 Free space holographic optical element for distribution of the clock [Ref. 5].

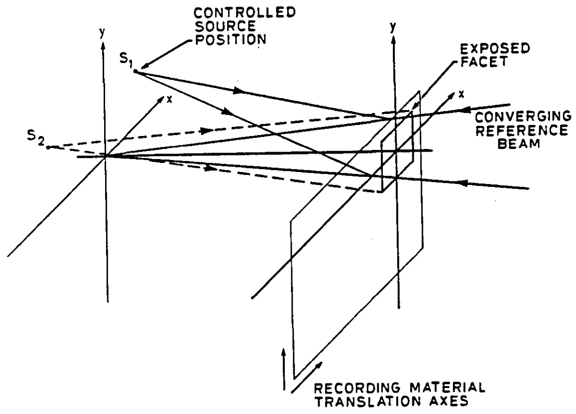

CONTROLLED SOURCE POSITION
y
S1
S2
x
y
x
EXPOSED FACET
CONVERGING REFERENCE BEAM
RECORDING MATERIAL TRANSLATION AXES

Figure 5 Multi-facet hologram formed with selective object source points. Source points are encoded in sequential fashion [Ref. 7].

Bergman[8] et al have been looking at reflection holograms fabricated as a surface relief in silicon. The holographic pattern is recorded in photographic film and then contact printed onto a layer of photoresist on a silicon substrate. After developing the photoresist, the silicon is chemically etched and then coated with aluminum. To date, they have achieved 20 percent diffraction efficiency with a promise of higher efficiency in the future.

Free space optical interconnects provides the potential of implementing reconfigurable interconnects. Figure 6 shows one possible implementation of a nonblocking crossbar network[9]. The input LED's or lasers represent the inputs of "n" different channels, while the detector array is the output to "n" channels. The spatial light modulator which may be electro-optic, magneto-optic or acousto-optic, can be programmed to connect any one of the inputs to one or any combination of outputs.

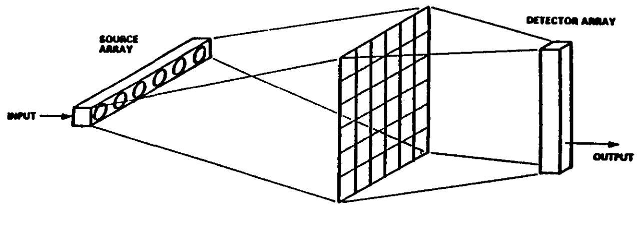


Figure 6 Free space optical interconnect implementation of a crossbar network [Ref. 9].

Packaging Techniques

For optics to be used in computing systems, packaging methods must be developed to allow high-speed silicon and GaAs digital components to be packaged with opto-electronic components and waveguides[9,10]. Unique packaging problems arise when optical interconnects are used with high speed electronics. The coupling between opto-electronic components and waveguides requires critical alignment tolerances of less than a micron. In addition, thermal conditions on the chip carriers must be tightly controlled to keep the opto-electronic devices from drifting.

A technique has been developed and demonstrated at Honeywell[4] to align several fibers to a multiple detector array. Figure 7 shows the optical coupler which was developed to align and attach multiple fibers. V-grooves were etched in silicon[11] to provide an excellent alignment fixture since the spacing between the V-grooves can be delineated exactly to match the detector spacing. As shown in Figure 7, the exit end of the fiber is polished at an angle such that the light strikes the end of the fiber and is total internally reflected through the bottom of the fiber. Since the fiber used in these experiments was multimode (50 um core diameter), the end of the fiber must be polished at 58° so that all of the modes are total internally reflected. It is interesting to note that since the fiber has a cylindrical geometry, the fiber acts as a lens to focus the light into an elliptical spot in the detector plane. This allows the detector size to be smaller than the fiber core diameter which could be important for very high speed systems.

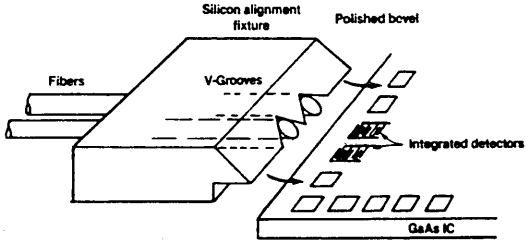


Figure 7 Fiber to detector alignment technique utilizing silicon V-grooves [Ref. 9].

Maximum efficiency is limited to about 70% using this technique, due to Fresnel reflection from the surface of GaAs. This could be reduced by introducing an index matching fluid or depositing an anti-reflection coating on the surface of the detector. Without using index matching fluid, anti-reflection coatings or taking any special precautions, an efficiency of 60% has been demonstrated[9].

A different fiber to chip technique shown in Figure 8 has been demonstrated[12] that allows vertical bonding of the fiber to the chip. A high aspect ratio hole is etched in the silicon substrate by using a frequency-doubled argon-ion laser at a wavelength of 257 nm for laser-assisted etching in a 5% aqueous solution of HF. The diameter of the hole is controlled (to first order) by the diameter of the laser beam. In this case, the diameter of the hole was 12 um. Due to the nature of light guiding during the etching process, the side walls of the etch are nearly vertical.

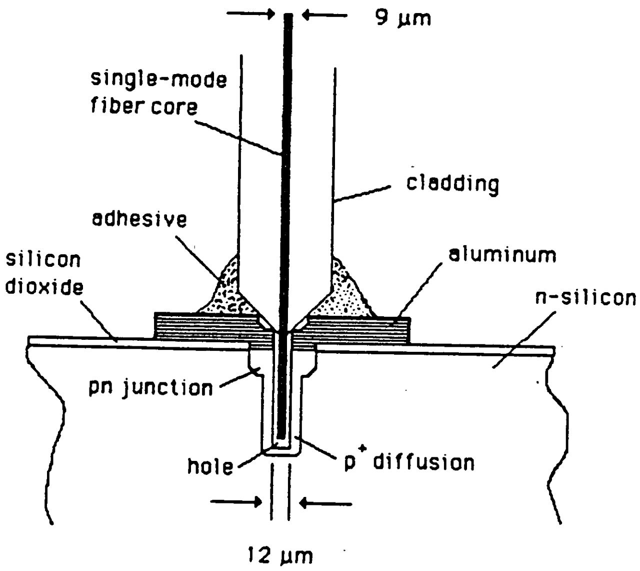


Figure 8 Schematic cross-section of a vertical mounted fiber optic coupler [Ref. 12].

Once the hole has been etched, a pn junction detector is fabricated in the hole using standard semiconductor device fabrication techniques. A single mode fiber is chemically etched down to its 9 um diameter core by immersion in buffered solution of HF for approximately 2 hours. The etched fiber is then inserted into the detector well and attached to the chip using epoxy. This system was tested by coupling a He-Ne laser emitting at a wavelength of 6328nm into the fiber using a microscope objective. By measuring the IV characteristic of the photodiode, the responsivity was determined to be 0.13 A/W corresponding to a quantum efficiency of approximately 25%.

The advantage to using this technique is the flexibility to put the optical detectors at any position on the chip. This allows the flexibility to receive information at any position. This technique is not planar, therefore, an unconventional package needs to be implemented to use this technique. This particular development was demonstrated in silicon, however, it could also be used for GaAs circuits.

A packaging technology based upon multichip integration is being developed at MIT Lincoln Laboratories. Individual die or chips are imbedded in a potting compound and cast[13] as shown in Figure 9. By using standard photolithographic processes, defining connections between the chips becomes fairly routine. Choosing the potting compound is crucial to ensure chemical and thermal compatibility with photolithographic processing steps. A breadboard demonstration was built which consisted of a commercial GaAs shift register (HMD-11141), a mass-transported GaInAsP laser diode[14], a 6-dB attenuator, and an outer-conductor DC block. The laser had a threshold current of 18.5mA and a 3dB frequency of 6GHz at a current of 50mA. Using a clock rate of 1.4GHz to the input of the GaAs shift register, a chip risetime of 150 ps and a falltime of 120 ps was measured.

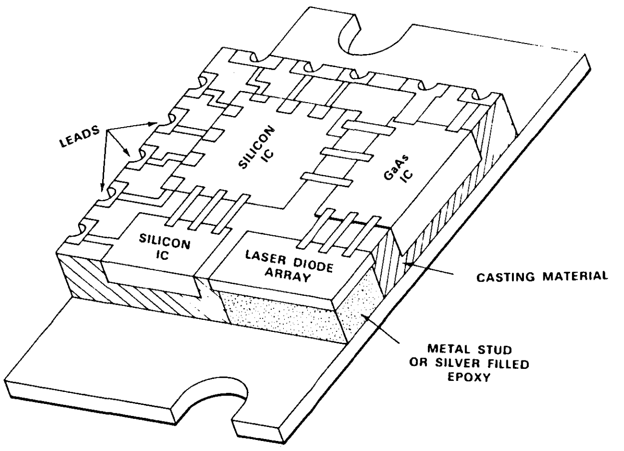


Figure 9 Multichip packaging technique of silicon, gallium arsenide, and opto-electronic components [Ref 13].

Opto-Electronic Device Development

In addition to optical routing, the effective use of optical interconnects also requires high performance, low power opto-electronic transmitters and receivers. The use of discrete components for the opto-electronic transmitter and receiver suffers from two disadvantages[15]; one is the difficulty of controlling parasitic reactances and the second is the high cost of manually fine tuning each module to create reproducible performance at high frequencies.

It is anticipated that in order to realize the full performance of an optical interconnect system, there would be a distinct advantage if the opto-electronic components were monolithically integrated with the high speed electronics. The excess capacitance and inductance with the bonding pads and wires would be eliminated. This translates to higher speed, less power and lower noise than its hybrid counterpart. GaAs provides an attractive material for monolithic integration since both GaAs digital integrated circuits and AlGaAs/GaAs opto-electronic devices are being developed at numerous laboratories. There have been a number of demonstrations of laser diodes integrated with either a single or a few GaAs IC components [16-20]. Although these devices demonstrated the feasibility of the monolithic integration of lasers with a small number of transistors, these techniques have not been suitable for integrating a laser with a large GaAs IC.

For the past several years, Honeywell and Rockwell have been developing the technology to integrate opto-electronic devices with complex GaAs circuits[15,21-24]. One structure that has been demonstrated[15] is shown in Figure 10. The components in the structure are a transverse junction stripe (TJS) laser, a field effect transistor (FET) driver, and a 4:1 multiplexer (MUX). The mux and driver active regions are formed by selective ion implantation into the semi-insulating GaAs substrate while the TJS laser is fabricated by liquid phase epitaxial growth in a well that is etched into the substrate. The rear laser mirror facet is formed by using a microcleave process developed at the California Institute of Technology[25]. A cross sectional view of the microcleave process is shown in Figure 11. The small horseshoe shaped wing at the end of the laser is formed by chemically etching under the AlGaAs. The wing is broken off to form a smooth cleaved mirror facet. Measurements have shown that lasers with undercut mirrors have operating characteristics (laser threshold ~ 37mA) identical to similar lasers having two cleaved mirrors. The opto-electronic chip having a size of 1.8 mm x 1.8 mm operated at speeds up to 150 MHz[26].

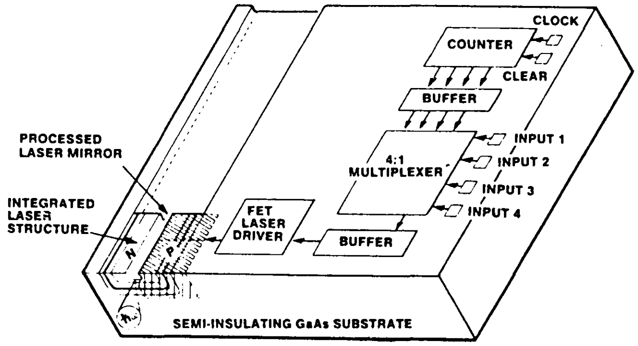


Figure 10 Block diagram of the monolithic opto-electronic transmitter consisting of an integratable TJS laser, laser driver and 4:1 multiplexer [Ref. 15].

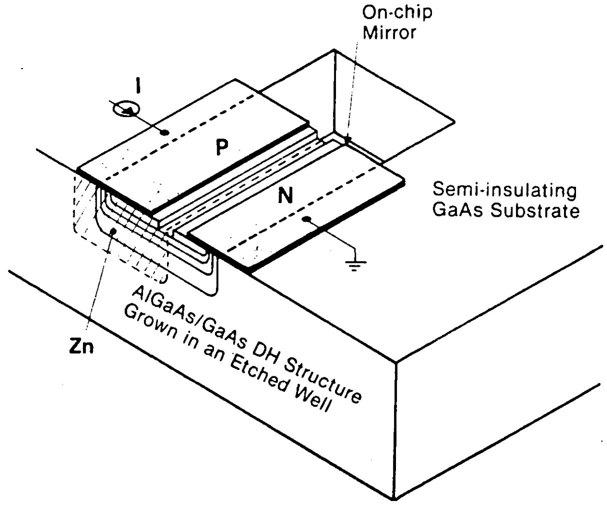

On-chip Mirror
I
P
N
Semi-insulating GaAs Substrate
Zn
AlGaAs/GaAs DH Structure Grown in an Etched Well

Figure 11 Microcleave process to form a smooth cleaved mirror facet for monolithic integration [Ref. 15].

More recently, a 1 Gbit/sec opto-electronic receiver chip was demonstrated[27]. The receiver chip was designed to digitally multiplex four high speed input signals. Two of the inputs are electrical and two are optical. The receiver consisted of two back-to-back Schottky diode detectors, two amplifiers, a 4:1 multiplexer and a laser driver. All of the components are fabricated on a semi-insulating GaAs substrate using direction implantation MESFET technology.

The photodector was fabricated directly on the semi-insulating substrate, thus eliminating the need for epitaxial growth. A pulse response of the back-to-back Schottky photodiode[28] is shown in Figure 12 for a reverse bias voltage of -15V. The rise and fall times of the photodetector response observed for a range of bias voltages between -10V and -15V were under 100 ps, which is sufficient for a 1 Gbit/sec operation of the receiver. The responsivity of the detector was measured to be 0.2 A/W for a wavelength of 0.84 um. The output of the detector is fed into a three-stage amplifier: a preamplifer, a gain stage and a buffer stage. The preamplifier translates the current into a voltage and amplifies to the proper GaAs logic levels by the gain stage and the buffer stage is used as a line driver to drive the input of the 4:1 multiplexer. The preamplifier had a gain of 20dB with a feedback around the second stage to improve its frequency response.

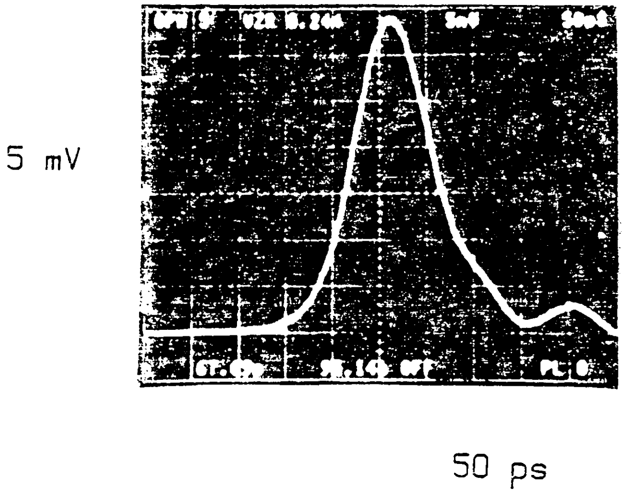


Figure 12 Pulse response of back-to-back Schottky photodiode for bias voltage of -15V [Ref. 26].

A similar effort on monolithic integration using a multiple quantum well (MQW) laser with a ridge waveguide structure[24] in shown in Figure 13. The MQW laser structure was grown by MOCVD and consisted of five 100 A GaAs wells separated by four 40 A $Al_{0.2}$ $Ga_{0.8}$ As barriers. The ridge waveguide was ion milled having a width of 5 um. These integrated devices have shown room temperature operation with 15 mA threshholds and differential quantum efficiences of 60%. For this particular integration scheme, the laser will be implemented as a surface emitting laser as shown in Figure 14. The laser light is coupled from the laser to a passive AlGaAs waveguide. The light in the waveguide is coupled from surface either with an etched V-groove or a grating as shown in Figure 14. This technique allows the user to fabricate lasers at any position on the chip rather than only at the edge of the chip. This method is more difficult to implement if fibers are to be used as the transmission medium. However, it does provide the flexibility for utilizing the third dimension and directing the optical signals via holographic techniques.

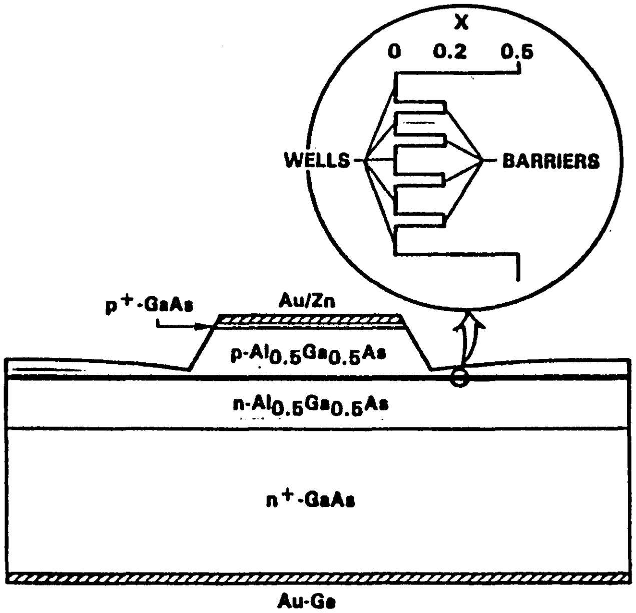


Figure 13 Cross-section and energy band diagram for a ridge waveguide MQW laser [Ref. 24].

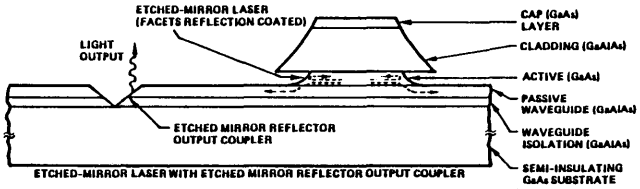

ETCHED-MIRROR LASER (FACETS REFLECTION COATED)
LIGHT OUTPUT
CAP (GaAs) LAYER
CLADDING (GaAlAs)
ACTIVE (GaAs)
PASSIVE WAVEGUIDE (GaAlAs)
WAVEGUIDE ISOLATION (GaAlAs)
SEMI-INSULATING GaAs SUBSTRATE
ETCHED MIRROR REFLECTOR OUTPUT COUPLER
ETCHED-MIRROR LASER WITH ETCHED MIRROR REFLECTOR OUTPUT COUPLER
LIGHT OUTPUT
GRATING OUTPUT COUPLER
ETCHED-MIRROR LASER WITH GRATING OUTPUT COUPLER

Figure 14 Surface emitting laser using (a) etched V-groove and (b) grating [from Rockwell].

The photodetector used in this development is a PIN photodector grown by MOCVD. The structure is a 6 layer stack consisting of GaAs and AlGaAs layers. Initial tests on this device have shown a dark current [24] of less than 10pA and a breakdown voltage of 80 V. A current gain of 10 was demonstrated at a voltage of 95% of breakdown.

Optical Interconnect Demonstration

There are a number of university, government and industrial laboratories developing analytical tools, technology, packaging techniques and system demonstrations. One system demonstration nearing completion is a hybrid model designed for point-to-point communications between chips and boards. A block diagram of the interconnect demonstration is shown in Figure 15. The board consists of four ECL Fairchild 100K series shift registers, each of which provides an 8-bit psuedo-random word at 250Mb/sec. The outputs of the four high speed ECL chips are connected to a GaAs 4:1 multiplexer which has an output of 1 Gbit/sec. Two of the ECL chips are packaged with an AlGaAs laser, laser drivers and an optical fiber. The other two word generators are packaged alone such that the GaAs chip has two optical inputs and two electrical inputs. The GaAs 4:1 multiplexer package consists of two optical fiber inputs, two electrical inputs, the GaAs chip with 4:1 multiplexer, detectors, preamplifier, laser driver and a separate laser coupled to an optical fiber. The optics was designed to have a minimum impact on the existing high-speed package, but still provide rigid and efficient coupling to the opto-electronic components.

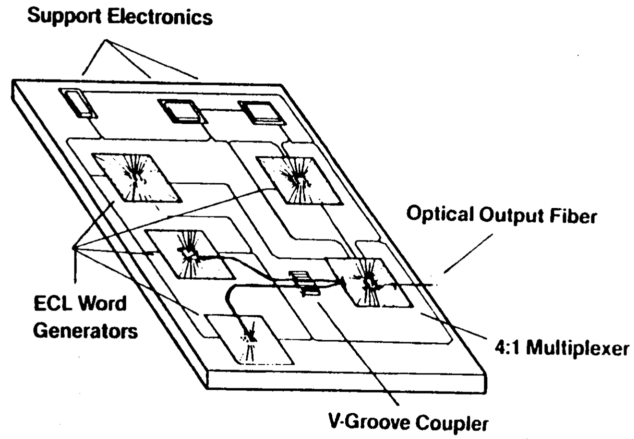


Figure 15 Layout of chip-to-chip and board-to-board Optical Interconnect Demonstrations [Ref. 1].

The full system test has not been completed; however, key components in the system have been tested. The GaAs receiver chip was tested and found to be fully functional. An optical fiber was positioned over one of the two detectors, and one of the four data channels was successfully modulated with an optical input. The 4:1 multiplexer on the GaAs chip was implemented using a complementary clocked dual shift register architecture. This architecture makes it possible to have an output data rate (NRZ) that is twice the clock speed. All of the shift registers in the multiplexer were designed using a differential current-mode-logic structure to minimize the delay and device count in the shift registers[27]. The complete GaAs chip consists of 807 depletion mode transistors and 325 level shifting diodes.

A test fixture was developed to test the high speed GaAs receiver. Tests were performed to determine the maximum data rate that the 4:1 multiplexer was capable of handling. The bandwidth of the test equipment was the limiting factor in determining the maximum data rate of the chip. The maximum data rate measured on the receiver chip was 2 Gbit/s (NRZ). The size of the GaAs chip is 2.82 mm x 2.04 mm and it dissipated 3.5 watts of electrical power at 1 Gbit/sec. It is anticipated that test data for the full optical interconnect demonstration will be presented at the conference.

Another optical interconnect demonstration is being developed at the Naval Ocean Systems Center[29]. Figure 16 shows the concept where an optoelectronic switch will be used in a time division muliplexing (TDM) scheme. This is being investigated as an output from a VLSI chip where the timing comes from adjusting the length of the fiber. A side view of the optoelectronic switch is shown in Figure 17. The metal interconnect stripline has a small discontinuity such that the signals cannot be transmitted to the edge of the chip. The connection is made by shining a high speed laser pulse at the position of discontinuity. This increases the conductivity of the semiconductor sufficiently to short the discontinuity. By adjusting the length of the fiber several of these microstrip lines can be multiplexed to a high speed serial output. This technique has been demonstrated in silicon but could also be implemented in GaAs.

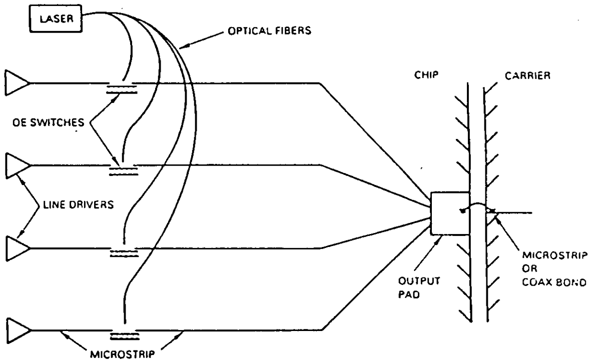


Figure 16 Optoelectronic time division multiplexing scheme for VLSI chip output [from NOSC].

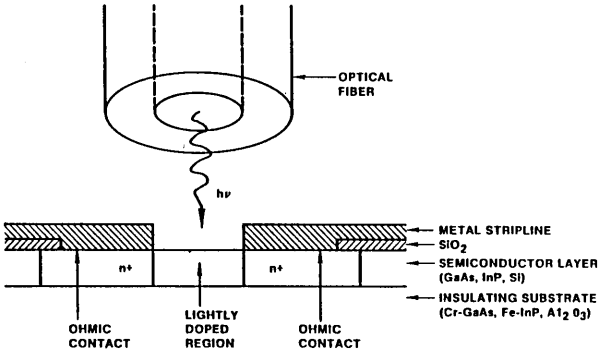


Figure 17 Side view of optoelectronic switch [from NOSC].

Conclusions

As can be seen by the discussion presented in this paper, there are numerous researchers developing the concepts and technology for advanced optical interconnects for high speed computing applications. A lot of attention is being given to packaging high speed silicon VLSI devices with high speed GaAs integrated electronics, opto-electronics and optics. The optimum method for routing the optical signals has yet to be determined. However, fiber optics for the board-to-board and local area network applications and free-space holographic techniques for the intrachip and chip-to-chip communications appears to have the most promise. The monolithic integration of opto-electronic components with electronics is making rapid progress, yet a number of technical hurdles such as ultra-low threshold lasers, reliability, yield and low power, need to be solved before these devices can be implemented in computing applications.

The results to date are encouraging for optics to become an integral part of future high throughput computing systems. For the near term, we can expect several demonstrations of optical interconnects using hybrid components. As the packaging of electrical and optical components becomes more common, we will learn which applications and which high speed circuits will benefit from this technology.

Acknowledgements

I would like to acknowledge the help of numerous people who provided information for this paper. They include John Neff of DARPA, Paul Haugen and Sankar Ray of Honeywell, Paul Prucnal of Columbia University, Joe Goodman of Stanford University, Dean Tsang of MIT Lincoln Labs, Alan Johnston of Jet Propulsion Laboratory, Kevin Kilcoyne of Rockwell and Ron Reedy of Naval Ocean Systems Center.

References

1. L. D. Hutcheson, P. R. Haugen and A. Husain, "Gigabit per Second Optical Chip-to-Chip Interconnects," SPIE Proceedings, Cannes, France, November, 1985.

2. J. E. Midwinter, "Current Status of Optical Communications Technology," J. of Light. Tech., LT-3, 927 (1985).

3. J. Fried, "Optical I/O for High-Speed CMOS Systems," to be published in Optical Engineering, October, 1986.

4. P. R. Haugen, S. Rychnovsky, L. D. Hutcheson and A. Husain, "Optical Interconnects for High Speed Computers," to be published in Optical Engineering, October, 1986.

5. B. D. Clymer and J. W. Goodman, "Optical Clock Distribution to Silicon Chips," to be published in Optical Engineering, October, 1986.

6. J. W. Goodman, F. J. Leonberger, S. Y. Kung and R. A. Athale, "Optical Interconnections for VLSI Systems," Proc. IEEE, 72, 850 (1984).

7. R. K. Kostuk, J. W. Goodman and L. Hesselink, "Optical Imaging Applied to Microelectronic Chip-to-Chip Interconnections," Appl. Opt., 24, 2851 (1985).

8. L. Bergman, W. H. Wu, A Johnston, R. Nixon, S. Esener, C. Guest, P. Yu, T. Drabik, M. Feldman and S. H. Lee, "Holographic Optical Interconnects for VLSI," to be published in Optical Engineering, October, 1986.

9. P. R. Haugen, A. Husain and L. D. Hutcheson, "Directions and Development in Optical Interconnect Technology," SPIE Proceedings Vol. 625, P. 110, January, 1986.

10. D. H. Hartman, M. K. Grace and F. V. Richard, "An Effective Lateral Fiber-optic Electronic Coupling and Packaging Technique Suitable for VHSIC Applications," J. Light. Tech., LT-4, 73 (1986).

11. E. J. Murphy and T. C. Rice, "Low-Loss Coupling of Multiple Fiber Arrays to Single-Mode Waveguides," J. Light. Tech, LT-1, 470 (1983).

12. P. R. Prucnal, E. R. Fossum and R. M. Osgood, "Integrated Fiber-Optic Coupler for Very Large Scale Integration Interconnects," Optics Letters, 11, 109 (1986).

13. D. Z. Tsang, D. L. Smythe, A. Chu and J. J. Lambert, "A Technology for Optical Interconnections Based on Multichip Integration," to be published in Optical Engineering, October, 1986.

14. Z. L. Liau, J. N. Walpole and D. Z. Tsang, "Fabrication, Characterization, and Analysis of Mass-Transported GaInAsP/InP Buried-Heterostructure Lasers," IEEE J. Quant. Elect., QE-20, 855 (1984).

15. J. Carney, M. Helix, R. Kolbas, S. Jamison and S. Ray, "Integrated Optoelectronic Transmitter," SPIE Proceedings, Vol. 408, April, 1983.

16. I. Ury, S. Margalit, M. M. Yust, and A. Yariv, "Monolithic Integration of an Injection Laser and a Metal Semiconductor Field Effect Transistor," Appl. Phys. Lett., 34, 430 (1979).

17. T. Fukuzawa, N. Nakamura, M. Hirao, T. Kuroda, and J. Umeda, "Monolithic Integration of a GaAlAs Injection Laser with a Schottky-gate Field Effect Transistor," Appl. Phys. Lett., 36, 181 (1979).

18. J. Katz, N. Bar-Chaim, P. C. Chen, S. Margalit, I. Ury, D. Wilt, M. Yust and A. Yariv, "A Monolithic Integration of GaAs/AlGaAs Bipolar Transistor and Heterostructure Laser," Appl. Phys. Lett., 37, 211 (1980).

19. I. Ury, K. Lau, N. Bar-Chaim and A. Yariv, "Very High Frequency GaAlAs Laser Field-Effect Transistor Monolithic Integrated Circuit," Appl. Phys. Lett., 41, 126 (1982).

20. M. Kim, C. Hong, D. Kasemset and R. Milano, "GaAlAs/GaAs Integrated Optoelectronic Transmitter using selective MOCVD Epitaxy and Planar Ion Implantation," Proceedings IEEE GaAs IC Symposium, October, 1983.

21. R. Kolbas, J. Carney, J. Abrokwak, E. Kalweit and M. Hitchell, "Planar Optical Sources and Detectors for Monolithic Integration with GaAs Metal Semiconductor Field-Effect Transistor (MESFET) Electronics," Proceedings SPIE, Vol. 321, January, 1982.

22. J. K. Carney, M. J. Helix, R. M. Kolbas, S. A. Jamison and S. Ray, "Monolithic Optoelectronic/Electronic Circuits," Proceedings IEEE GaAs IC Symposium, October, 1982.

23. M. Kim, C. Hong, D. Kasemset and R. Milano, "GaAlAs/GaAs Integrated Optoelectronic Transmitter Using Selective MOCVD Epitaxy and Planar Ion Implantation," Proceedings IEEE GaAs IC Symposium, October, 1983.

24. M. K. Kilcoyne, D. Kasemset, R. Asatourian, S. Beccue, "Optical Data Transmission Between High Speed Digital Integrated Circuit Chips," SPIE Proceedings, Vol. 625, P. 127 (1986).

25. H. Blauvelt, N. Bar-Chaim, D. Fekete, S. Margalit and A. Yariv, "AlGaAs Lasers with Micro-Cleaved Mirrors Suitable for Monolithic Integration," Appl. Phys. Lett., 40, 2891 (1982).

26. J. Carney, M. Helix and R. M. Kolbas, "Gigabit Optoelectronic Transmitter," Proceedings IEEE GaAs IC Symposium, October, 1983.

27. M. P. Walton, P. R. Haugen and S. L. Palmquist, "A 1 Gbit/s Optical/Electrical Input Monolithic GaAs Transmitter IC," Proceedings IEEE MTT-S International Microwave Symposium, Baltimore, Maryland, June, 1986.

28. S. Ray and M. P. Walton, "Monolithic Optoelectronic Receiver for Gbit Operation," Proceedings IEEE MTT-S International Microwave Symposium, Baltimore, Maryland, June, 1986.

29. Information on the optoelectronic switch received from Ron Reedy of the Naval Ocean Systems Center.

Optical Interconnection Systems for Digital Parallel Processors

Alexander A. Sawchuk
Signal and Image Processing Institute
University of Southern California
Mail Code 0272
Los Angeles, CA 90089
(213)743-5527

ABSTRACT

Optical techniques have great potential for applications in the dynamic interconnection of digital parallel processing systems. In this paper, we review some of the relevant advantages of optics, including high signal bandwidth, inherent parallelism, and low mutual interference. Specific optical realizations of dynamic generalized crossbars and other types of networks are presented. One way to implement crossbar networks is to utilize optical matrix-vector multiplier architectures; several designs are compared. The systems may use acousto-, electro-, or magneto-optic spatial light modulators as the active controlling element. Other optical digital processors that can realize crossbars and multistage networks are also described.

1.0 Introduction

This paper is concerned with the application of bulk (3-D) optical components and optical architectures to interconnection problems in digital systems. Efficient communication among the elements of these systems is recognized as the key to faster computing through the use of parallel processors and shared resources. This paper concentrates on dynamic, rapidly time-variable interconnections at the spatial scale size of processors (10^0-10^3 m) and boards (10^{-1}-10^0 m).

Several multiprocessor systems incorporating a large number of processors are in various stages of development [COMP 82], [COMP 85]. A crossbar network is very desirable for interconnections among these processors because every processor can communicate with every other processor with no conflicts. However, the electronic implementation of crossbar networks of large size is difficult due to constraints on VLSI technology. The parallel nature of optics, its high available signal bandwidth, together with its relative freedom from interference makes it attractive for implementing large crossbar and other networks that do not suffer from many of the limitations of their electronic counterparts.

1.1 Need for Parallel Interconnection Architectures

Any multiprocessor system that applies several processors to a job must be designed to allow efficient communication between processors, and between processors and memories, or else the advantage of multiprocessing is offset by the communication bottleneck. A crossbar interconnection network allows every processor to communicate with every other processor with no conflicts in the network (we use the term "crossbar" here to refer to networks which are strictly non-blocking, in which a unique path exists from every input to every output; the complexity of such networks grows as the square of the network size N) - however, the electronic implementation of large crossbar networks has been considered impractical due to technological constraints. This fact forced researchers to proceed in one of two directions: (i) design interconnection networks with less than $O(N^2)$ complexity, which are feasible to build and easy to control, and, at the same time, provide reasonable performance; or (ii) design new algorithms or modify existing algorithms to suit the limited interconnection structure, so as to utilize the available interconnection structure most efficiently. Due to their inherent 3-dimensional nature, low mutual interference between channels, and high bandwidth, optical switching systems offer great potential for multiplexing data from thousands of processing elements over a single optical channel without causing a speed bottleneck.

1.2 Interconnection Considerations

Dynamic reconfigurable interconnection networks are desirable for interconnecting a large number of processors (or PEs) to memories or to other processors. Connecting every PE to every other PE (or memory) with a fixed, dedicated line is not viable because of the number of lines and connections required. Networks range in complexity from a simple bus to a crossbar. In this section we define some important interconnection network characteristics and describe how they differ in electronic and optical implementations.

Number of Lines. This parameter, N, defines the number of input and output lines. Implementing moderately large crossbars (>100×100) in electronics is difficult due to bandwidth, power and reliability contraints.

Bandwidth. This refers to the bandwidth of each line in the network. In passive optical networks (which do not detect and regenerate optical signals internally), the bandwidth is limited by external sources and detectors, which can easily operate at 1 Gb/s rates or higher. Sources (laser diodes) and detectors are available at 20 GHz and 40 ps, respectively. 1-D arrays of these devices are presently limited to lower values; existing detector arrays with parallel outputs operate at approximately 50 Mb/s. In contrast, electronic systems typically operate at 10 Mb/s for each line.

Reconfiguration Time. Reconfiguration of optical networks is limited to approximately 1 μs for moderate or large networks, with existing or near-future technology. Optical device arrays presently have switching times of 1 μs for each switch at best. For an electronically controlled optical switching array, it is a matter of switching time *and* addressing schemes to control all of the switch states: Practical electronic switches can typically switch in 50-100 ns for each switch. Fast reconfiguration of the network is needed in an environment where the switching permutations change rapidly. However, slow switching could be tolerated in applications where each connection is followed by the transfer of large blocks of data.

CH2345-7/86/0000/0457$01.00 © 1986 IEEE

Broadcast Capability. Broadcasting is the ability for one processor to simultaneously send information to more than one other processor. If the interconnection network is capable of achieving broadcasts in one step, the overall time-complexity of many parallel algorithms can be reduced. In electronic implementations, the provision of broadcasting involves higher control complexity, a larger number of pins, and/or slower operation. In some optical systems, the addition of broadcast capability involves only very minor increases in complexity.

Data Format. The switching process as well as the transfer of data can be performed either synchronously or asynchronously. Synchronous operation requires strobing or capture of data at precise instants of time in the system; data buffers are generally required.

Type of Detector. Three major kinds of multichannel optical detectors, namely real-time, shift/add and time-integrating are required to implement these systems. The data bandwidths of systems with a vector array of real-time detectors is limited by the response time of each. Limitations on shift/add and time-integrating detectors are generally due to multiplexing electronics needed to provide the electronic output.

1.3 Electronic Crossbar Networks

We note very briefly that a few electronic crossbar switch units with as many as 512×512 switches have been designed and/or constructed for experimental special-purpose parallel processor systems [BURG 83], [DENN 82], [BROO 84], [SAWC 85], [SAWC 86]. These systems are generally physically large, dissipate a great amount of heat, and have questionable reliability.

2.0 Crossbar Interconnections with a Matrix-Vector or Matrix-Matrix Processor

Suppose that N serial data input lines (each one-bit wide) are to be connected to a set of N serial data output lines (each one-bit wide). The data bits on each line can be thought of as flowing synchronously, although, for some of the techniques described, this synchronism is not necessary. The data on the input lines at some instant of time is represented by the column vector $\vec{b}$ of length N. The data on the output line of the interconnection network is represented by the column vector $\vec{c}$ of length N, and both these vectors have only binary elements. The state of the switch is described by the $N \times N$ matrix A whose elements are also 0's and 1's. Matrix A is a generalized permutation matrix; an entry of 1 in row i and column j of A means that input j is connected to output line i. The overall operation of the interconnection network is represented by

$$\vec{c} \doteq A\ \vec{b} \qquad (2.0\text{-}1)$$

which is a matrix-vector multiplication. As an example of this we have (for $N=4$),

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1&0&0&0 \\ 0&0&1&0 \\ 0&1&0&0 \\ 0&0&0&1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \qquad (2.0\text{-}2)$$

Note that each row and column of A must have only a single "1" entry for the switch to act as an ideal one-to-one crossbar permutation network. The system can perform broadcasting (one input connected to two or more outputs) if there are two or more 1's in a column of A. However, two or more 1's in any row of A implies contention, as several inputs attempt to communicate with one output. Once the physical connection described by the matrix A is made, the data transfer can be synchronous or asynchronous.

This idea can be generalized for the interconnection of N parallel input lines (each M bits wide) to N parallel output lines (each M bits wide). In this case, the input lines are represented by an $N \times M$ matrix B, the output lines are represented by an $N \times M$ matrix C, and the interconnection matrix A is the same as before. All matrices (A, B and C) have binary entries as before, and the switching operation is

$$C = AB \qquad (2.0\text{-}3)$$

Thus, systems capable of performing matrix-vector or matrix-matrix multiplications with binary entries as given in Eq. (2.0-1) or Eq. (2.0-3) are suitable for use as a crossbar interconnection network. There are many techniques available for this purpose in optical signal processing. In some of these systems, the physical switch may be analog and passive (a switchable reflective or transmissive element); thus bit synchronism is not required and the data bandwidth is limited only by the optical sources and detectors used. In other systems, the matrix-vector multiplication is implemented by active electro-optical or acousto-optical components, implying the detection and regeneration of optical signals. In such systems the switch itself may limit the data bandwidth. In the remainder of this section we discuss some of these systems.

2.1 N^2-Parallel Inner Product Processor

Figure 2.1-1 schematically shows a system capable of performing optical matrix-vector multiplications of the form of Eq. (2.0-1) in an inner product format [ATHA 83]. The N input lines drive a 1-D array of N light emitting diodes (LEDs) or laser diodes with a binary signal, so that a binary 1 is represented by light of a fixed intensity, and a binary 0 is represented by a lower (or zero) intensity. An optical system to the right of the input vector spreads the light from each input source into a vertical column that illuminates the crossbar mask shown. The crossbar mask consists of an $N \times N$ array of windows representing the entries in the $N \times N$ permutation matrix A. An entry of 0 in A corresponds to zero light transmission, or an opaque window in the mask. An entry of 1 in A corresponds to full light transmission or an open window. In the systems we are investigating, the crossbar mask is an electrically-controllable shutter array that is time-variable through the application of external control signals. Several different technologies are available for this purpose, including mechanical, electro-optic, and magneto-optic.

Following the crossbar mask, the next set of optics collects the light transmitted by each row of the mask, and sums the mask output onto a 1-D vertical array of N photodetectors corresponding to the N output lines. Thus the system performs a parallel matrix-vector multiplication. There are many possible ways to actually implement this schematic design, some of which use discrete optics or optical waveguides.

This N^2-parallel inner product architecture has an optical light efficiency of $1/N$ at most, when used for the usual crossbar operation. This occurs because $(N-1)/N$ of the light from each input source does not pass through the crossbar mask. A matrix-vector operation is completed in just one clock cycle. A matrix-matrix multiply takes N clock cycles. Since it is a passive interconnection, once the mask is set, the data can flow through synchronously or asynchronously, can be analog

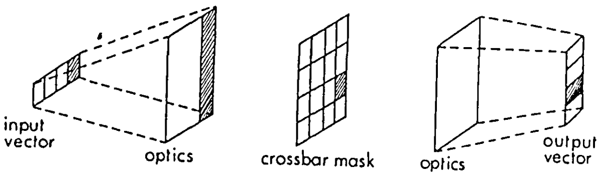


Fig. 2.1-1. N^2-parallel matrix-vector inner product processor.

or digital, and has a bandwidth limited only by the sources and detectors. The inputs and outputs can also be directly coupled to optical fibers; in this case the sources and detectors can be some distance from the system. Another advantage of this system is that no input or output buffers are required. Broadcasting is possible, providing a generalized crossbar interconnection. The control is external and could be optical or electronic; this architecture is useful primarily for circuit switching applications.

2.2 Systolic Architectures

Systolic architectures have the advantage of providing for rapid reconfiguration of the network, i.e. they can be reconfigured at the bit rate of the data (neglecting any overhead in calculating the needed states of the crossbar matrix elements). In systolic systems, though, the input data must be configured appropriately for the matrix multiplication to proceed; this will require electronics at the input that can buffer the signals and send them into the switch in the appropriate sequence and at appropriate times. In some cases, only minimal buffering is required, e.g. when the data is originally time-division multiplexed bit by bit.

An N-parallel systolic matrix-vector multiplier is shown in Fig. 2.2-1 and implements the matrix-vector multiplication of Eq. (2.0-1) [ATHA 83].

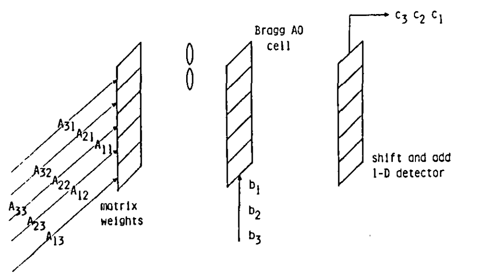


Fig. 2.2-1. N-parallel systolic matrix-vector multiplier.

The matrix elements A_{ij} conceptually flow in from the left. (Each pair of matrix elements along a line is actually separated by a 0.) They can be realized physically by a 1-D array of LED's. The vector elements b_j enter the end of the acousto-optic (AO) cell. (The b_j's are interlaced with 0's). The signals are synchronized so that A_{11} is emitted from the center LED when b_1 reaches the center of the AO cell. The system is arranged so that the light from each LED passes through the corresponding pixel of the AO cell, and results in the product $A_{ij}\ b_j$ of these elements at the corresponding element of the 1-D detector array. A shift/add detector (e.g., 1-D CCD) will accumulate the appropriate elements and output the resulting vector elements c_i serially. For example, in the first cycle $A_{11}\ b_1$ will appear at the middle detector. In the second cycle $A_{11}\ b_1$ will have been (electronically) shifted down one detector, element A_{12} will appear on the corresponding LED and b_2 will have moved up to the corresponding AO cell location, resulting in $A_{11}\ b_1 + A_{12}\ b_2$ on this detector. In this manner all terms of c_1 are accumulated by the time it reaches the end of the detector array.

Compared to the inner product architecture above, systolic architectures yield crossbars that are more light efficient, have shorter reconfiguration times, but have lower bandwidth. A number of these 1-D elements in the N-parallel multiplier can be placed in parallel however, permitting each line to be M bits wide, thereby increasing the effective bandwidth by a factor of M ($M \approx 100$ with current devices). Because a new matrix is essentially read in for each new bit or word, the system can be reconfigured rapidly.

An N-parallel systolic matrix multiplier requires $N(4N-3)$ clock cycles to complete one matrix-matrix multiplication, or one arbitrary switching operation on N input lines, each N bits wide. In this time 1 (N-bit) word on each line passes through the network. An N^2-parallel systolic arrangement can do this in $4N-3$ clock cycles. The detector in these cases is a shift-and-add detector, instead of the parallel readout detector needed in the parallel inner-product multiplier above. In most cases, these detectors will limit the system bandwidth. Also the bandwidth of each line decreases as the number of lines is increased. The potential application for this configuration is in switching time-division multiplexed signals. The data is digital in these systems and the operation is strictly synchronous. Also, the light efficiency can be substantially higher than the inner-product processor discussed above. The use of these systems is primarily when the number of lines is not too large, the bandwidth requirements are not extreme, and the reconfiguration is rapid.

2.3 Engagement Architectures

Engagement architectures such as that shown in Fig. 2.3-1 use time-integrating (instead of shift/add) detectors, preferably with parallel outputs.

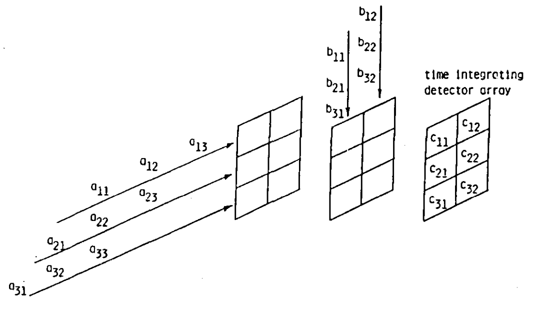


Fig. 2.3-1. Engagement architecture for matrix multiplication.

With sufficiently fast input and output devices, the acousto-optic (AO) cell used as a spatial light modulator (SLM) will limit the bandwidth of each individual line to ν_0/N, where $\nu_0 \approx$ 1 Gb/s. The matrix elements of A enter into a 2-D source array or a multichannel AO cell, skewed and formatted as shown. The data, or elements of B, enter a second, crossed AO cell from the top. A 2-D stationary integrating detector array then accumulates the results and outputs them in parallel.

An N-parallel engagement matrix multiplier requires $N(3N-2)$ clock cycles to complete one matrix-matrix operation or one arbitrary switching operation on N N-bit wide lines, approximately the same as the (N-parallel) systolic case. An N^2-parallel engagement arrangement [BOCK 84] takes $3N-2$ clock cycles. The input data formatting requirements are similar to the systolic case, but interleaved 0's are not required. The N-parallel case uses time multiplexing again. We can think of the width of each functional input line as being equal to the word length so that an M-bit wide line can transfer one word at an instant of time. In the N^2-parallel engagement case, there are N physical input lines, and the switch operates on word slices: it is word serial and bit parallel except for a unit time shift from one physical line to the next. Thus the first line has the first bit of each word, sequentially in time, etc. As in the systolic case, reconfiguration is rapid but updating of the state of the switches is needed even when the state does not change from one multiply to the next.

2.4 N^2-Parallel Outer Product Processor

An N^2-parallel outer product processor such as that shown in Fig. 2.4-1 performs $C=AB$ by taking outer products of column i of A and row i of B, and summing the results over i.

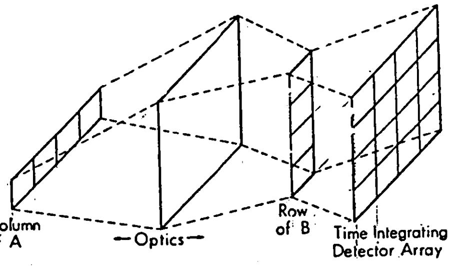


Fig. 2.4-1. N^2-parallel outer product processor.

It can perform a matrix-matrix operation in N clock cycles. However, typical acousto-optic implementations require the clock cycle to be equal to the fill time of the AO cell; this is a much longer clock cycle than the systolic and engagement cases. Thus, an acousto-optic implementation has the same speed order of magnitude as an N-parallel systolic or engagement architecture. Other implementations using electrooptic devices may avoid this difficulty. The data input is a row of B at a time, or bit parallel, word-serial. In the acousto-optic implementation discussed, the input is actually serial, first the first word of the first line (one bit at a time), then the first word of the second line, etc. The detector is a 2-D array of stationary, time-integrating detectors. With this system broadcasting can easily be done, to provide a generalized crossbar. As in the systolic and engagement cases, the data is completely synchronous. Since the A matrix is read in for each matrix multiplication, reconfiguration is rapid, although the state of the switches needs to be updated for each matrix multiply.

2.5 N^2-Parallel Inner Product AO Deflector

The system shown in Fig. 2.5-1 improves overall light efficiency by deflecting light instead of absorbing it.

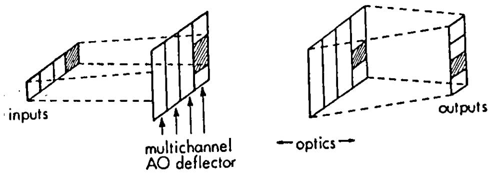


Fig. 2.5-1. N^2-parallel inner product AO deflector.

The input data to the crossbar enters through the 1-D array of sources on the left. Each source is then transferred to the corresponding cell of a multichannel AO device, used as a deflector. There is a separate channel for input line of data, or each column of A. Rather than read in the elements of A into the AO device (N elements into each channel), a single frequency-modulated signal is input into each channel; the frequency determines the amount of y-deflection desired for the corresponding input line. Broadcasting can be achieved, to a limited extent, by superimposing multiple frequencies onto the same channel of the AO cell, or by time multiplexing the different frequencies in the cell. Each channel of the AO cell in this system has just one signal on it; it is not divided into multiple (moving) resolution elements as in the previous AO systems. The light in each channel is then focused down in the x-dimension, yielding a 1-D output array.

In this system only the fill time of the AO cell limits the reconfiguration time of the switch. The bandwidth of the input data lines could be quite high; it is limited by the speed of the sources and detectors, and not by the AO cell. The time bandwidth product requirements of each cell of the AO device are modest. The reconfiguration time of the network is equal to the time window used in each cell, which is less than or equal to the maximum time window of the device; thus a reconfiguration time on the order of 1 μs can be expected.

Another advantage of this system is that, as in the N^2-parallel inner product multiplier, there are no special formatting or buffering requirements on the input or output data. The data enters in a word-serial, bit-parallel fashion; each different functional input line is a different physical input line. Also, there are N control signals instead of N^2, and they are read in in parallel. The number of input and output lines is equal to the number of cells in the AO device, which puts a limit on the number of lines that can be switched with one crossbar. The operation of the switch can be synchronous or asynchronous, and the data can be analog or digital.

2.6 N^2-Parallel Inner Product/Engagement Processor

Another type of matrix-matrix processor architecture usable as an optical crossbar is shown in Fig. 2.6-1.

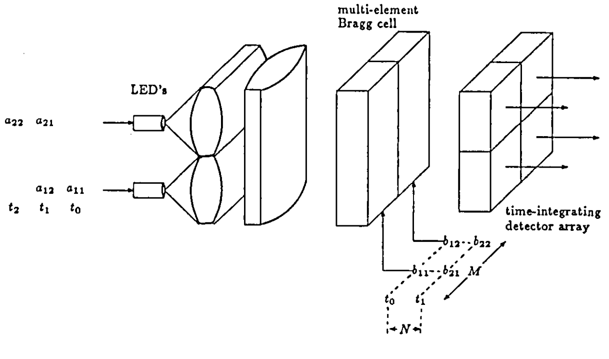


Fig. 2.6-1. N^2-parallel inner product/engagement processor.

The system is a combination of an N^2-parallel inner product processor and an N^2-parallel engagement architecture. The b 's to be multiplied in the system enter a parallel array of AO cells as shown, although the data for each row enters simultaneously instead of being staggered as in the N^2-parallel engagement processor. The input data also arrives in a time-staggered form and controls the illumination of LED's or laser diodes as shown. A set of optics similar to that in the N^2-parallel inner product processor spreads the light across the multichannel AO cell, and the final result accumulates on a time-integrating detector array. The results emerge in a time-offset fashion from a row of the detector array as shown. This system requires $3N-2$ clock cycles to complete a matrix-matrix multiplication and requires a digital synchronous data format. Broadcast operation is possible with this system, and the overall light efficiency can be as high as $1/N$.

2.7 Matrix-Vector System Performance Comparison

The systems described in these previous sections have greatly differing characteristics. For example, some of the systems provide a bandwidth that is essentially independent of the size of the system, while for others, it is an inverse function of the system size. In Table 2.7-1, we compare the six different systems of Sections 2.1-2.6 based on the six parameters of interest listed in section 1.2.

The engagement and outer-product systems with full N^2 parallelism require $N \times N$ time-integrating detectors which are operated synchronously with the data. 2-D $N \times N$ detector arrays have either N output lines or 1 output line. Thus the necessity of multiplexing at least N detector signals onto each electronic output line creates a bottleneck and limits the bandwidth of the overall system. Using N^2 parallel outputs could eliminate this bottleneck; the feasibility of such detectors is worthy of investigation. Also, the number of channels N may be limited by the time-integration of noise. The systolic architectures require a shift/add detector array to perform the summation. $N \times N$ shift/add arrays necessarily have at most N output lines; this and the use of electronics to perform the shift add limits the detector speed, in turn lowering the overall bandwidth of the system.

2.8 Other Matrix-Vector Systems

There are many other systems not described in this paper which show promise for interconnection applications. These systems include: an N^2-parallel encoded binary inner product processor; N^3-parallel matrix-matrix architectures; and an N-parallel strobed 1-D AO deflection system. These systems will be described in future papers.

3.0 Optical Sequential Logic System for Interconnection Networks

Because a computer can be used to implement an interconnection network, we can consider the use of an optical logic system. An optical logic system can be built out of gates and interconnections as shown in Fig. 3.0-1. An optical switching device can be used to implement a 2-D array of optical gates (e.g., NOR gates can be implemented since they can form a complete logic set). These gates can be interconnected with an optical system consisting of lenses and one or more computer-generated holograms [JENK 84a]. Here the interconnections between gates within this computer are fixed, and the system as a whole is used to implement a reconfigurable interconnection network. We have experimentally demonstrated a 16-gate optical circuit consisting of an oscillator and a synchronous

Method of implementation	Number of lines	Bandwidth (BW)	Reconfiguration time	Broadcast	Data format	Type of detector
N^2-parallel inner product	> 256	1 GHz (passive)	$(N)1\ \mu s$	yes	async	real time $N \times 1$ element
N^2-parallel systolic	> 100	10 MHz*	N/BW	yes	sync	$N \times N$ shift/add
N^2-parallel engagement	> 100	10 MHz* - 100 MHz	N/BW	yes	sync	$N \times N$ time-integrating
N^2-parallel outer product	>256	10 MHz* - 1 GHz	max (N/BW, $N \cdot 10$ ns)	yes	sync	$N \times N$ time-integrating
N^2-parallel inner product AO deflector	> 100	1 GHz (passive)	$1\ \mu s$	limited	async	real time $N \times 1$ element
N^2-parallel inner product/ engagement	>100	10 MHz* - 100 MHz	N/BW	yes	sync	$N \times N$ time-integrating

* Limited by output (detector array) electronics. Higher numbers (when given) apply when fibers guide light to discrete detectors or other optical components.

Table 2.7-1. Comparison of optical crossbar capabilities.

master-slave flip/flop with this system [JENK 84b]. We estimate that with current technology large numbers of gates (10^6 – 10^7) can be interconnected. The primary limitation at present is the extremely slow response time of the device (a liquid-crystal light valve) used to implement the gate array (10-100 ms). Fortunately, however, entirely different technologies can be utilized for optical switching, and current research in this area provides hope for realizing respectable speeds: individual optical (NOR) gates have been demonstrated at 82 MHz [JEWE 84a], [JEWE 84b], and research has begun on using this switching mechanism in 2-D arrays. We have investigated the possible implementation of optical interconnection networks with the optical system of Fig. 3.0-1 under the assumption that such 2-D switching arrays will become available.

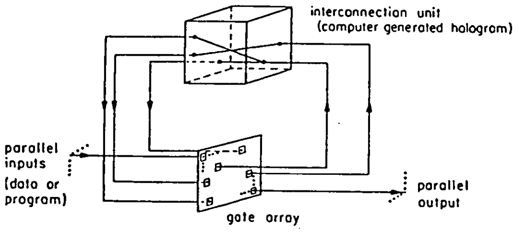


Fig. 3.0-1. Block diagram of the optical sequential logic system.

The number N of input and output nodes of a network implemented with this system is limited by the complexity of the hologram(s) and the number of gates on the device. If we assume an array of $10^3 \times 10^3$ gates on the device and currently available plotting devices for writing the hologram, we can estimate the maximum N that *one* device and *one* hologram interconnection unit can implement. Here we will assume centralized control with the control signals generated externally. A rearrangeable network consisting of a banyan followed by an inverse banyan, with all $2 \log N$ stages implemented in hardware, can be implemented with $N \approx 12{,}000$ (limited by the number of gates). In this case the data can pass through the network in a pipelined fashion. Implementing one shuffle/exchange stage also permits $N \approx 12{,}000$ (limited by hologram complexity), but the data must pass through the same hardware stage repeatedly to obtain rearrangeability. A non-blocking network can be realized by implementing a Clos network [CLOS 53] ($N \approx 650$) or a crossbar ($N \approx 500$). It should be noted that these values of N are not really limits because more than one optical system can ultimately be used in conjunction to implement much larger networks.

The optical system of Fig. 3.0-1 permits a large number of parallel input and output lines, allowing multiple optical systems to be connected to implement large interconnection networks without having to worry about careful partitioning of the network. This system differs from the optical matrix-vector inner product implementations discussed earlier: its data rate may be lower and its reconfiguration rate is higher, and the signal level of the data is regenerated as it passes through the network. The potential advantages over electronics in this case arise from the number and length of gate interconnections within each optical interconnection unit, and the number of parallel lines that can run from one optical system to another.

4.0 Conclusions and Acknowledgements

In this paper, we have reviewed the application of large crossbar interconnection networks for parallel computing systems and described several optical realizations for generalized crossbar and other networks. Several of the techniques for crossbar implementation utilize optical matrix-vector multiplier architectures, and moderately large crossbars (64×64 to 512×512) appear possible with current technology. Future work must consider additional relevant comparison factors such as light efficiency, input/output data formatting, ease of expansion, and methods for cascading small networks to make larger ones. Other difficulties that must be addressed are the relatively slow reconfiguration time of optical networks (primarily due to spatial light modulator limitations), techniques for routing and control, and methods for efficient optical-electronic signal conversion (and vice-versa). The author expresses sincere thanks to B.K. Jenkins, C.S. Raghavendra and A. Varma for their contributions to this work. This research was supported by DARPA/ARO under contract No. DAAG29-84-K-0066.

5.0 References

[ATHA 83] R. A. Athale, "Optical Matrix Algebraic Processors: A Survey," *Proc. 10th Int. Optical Computing Conf.*, Cambridge, MA, April 1983, pp. 24-31.

[BOCK 84] R. P. Bocker, "Optical Digital RUBIC (Rapid Unbiased Bipolar Incoherent Calculator) Cube Processor," *Opt. Eng.* vol. 23, January/February 1984, pp. 26-33.

[BROO 84] G. Broomell, J. R. Heath, "An Integrated-Circuit Crossbar Switching System," *Proceedings of the 4th International Conference on Distributed Computing Systems,"* May 1984, pp. 278-287.

[BURG 83] T. Burggraff, A. Love, R. Malm, A. Rudy, "The IBM Los Gatos Logic Simulation Machine Hardware," *Proceedings of the International Conference on Computer Design,* 1983, pp. 584-587.

[CLOS 53] C. Clos, "A Study of Non-blocking Switching Networks," *Bell Systems Technical Journal,* Vol. 32, No. 2, pp. 406-424, March 1953.

[COMP 82] Special Issue on Highly Parallel Computing, *IEEE Computer,* Vol. 15, No. 1, January 1982.

[COMP 85] Special Issue on Multiprocessing Technology, *IEEE Computer,* Vol. 18, No. 6, June 1985.

[DENN 82] M. M. Denneau, "The Yorktown Simulation Engine," *Proceedings of the 19th Design Automation Conference,* Las Vegas, 1982, pp. 55-59.

[JENK 84a] B. K. Jenkins, *et al.*, "Architectural Implications of a Digital Optical Processor," *Appl. Opt.*, Vol. 23, 1984, p. 3465.

[JENK 84b] B. K. Jenkins, *et al.*, "Sequential Optical Logic Implementation," *Appl. Opt.,* Vol. 23, 1984, p. 3455.

[JEWE 84a] J. L. Jewell, *et al.*, "Use of a Single Nonlinear Fabry-Perot Etalon as Optical Logic Gates," *Appl. Phys. Lett,* Vol. 44, 1984, p. 172.

[JEWE 84b] J. L. Jewell, *et al.*, "Single-Etalon Optical Logic Gates," *Technical Digest, Conference on Lasers and Electrooptics,* Optical Society of America, Washington, D.C., paper THg2, 1984.

[SAWC 85] A. A. Sawchuk, B. K. Jenkins, C. S. Raghavendra, A. Varma, "Optical Interconnection Networks", *Proc. 1985 IEEE International Conference on Parallel Processing,* St. Charles, Illinois, August 20-23 1985, pp 388-392.

[SAWC 86] A. A. Sawchuk, B. K. Jenkins, C. S. Raghavendra, A. Varma, "Optical Matrix-Vector Implementation of Crossbar Interconnection Networks", submitted to 1986 International Conference on Parallel Processing, St. Charles, Illinois, August 1986,

OPTICAL INTERCONNECTION TECHNOLOGY IN THE TELECOMMUNICATIONS NETWORK

Davis H. Hartman

Bell Communications Research
435 South Street
Morristown, New Jersey 07960-1961

ABSTRACT

The use of optics as an alternative method for achieving high speed (10 Gb/s $\geqslant$ B > 500 Mb/s) electronic interconnects is the subject of this paper. Particular emphasis is placed on application of optical interconnects to telecommunications technology. Optical interconnects are defined and delineated according to four separate genre. Issues and answers in the optical interconnect distribution scenario are discussed. The role of integrated optoelectronics for interconnects is stressed. Some specific recent results, theoretical and experimental, are given.

1. Introduction

Over the past fifteen years, photonics has found widespread applications in both commercial and military environments. Having its beginnings in the telecommunications field, photonics offered several clear advantages. Ultra-low loss, minimum dispersion fibers have opened doors to long-haul high speed communications hither-to unheard of. Presently, systems boasting greater than 100 km unrepeatered spans at 500 Mb/s exist. This information capacity has translated to lower cost per channel mile, leading to more profitable systems. Concurrently, other communications applications of fiber optics have emerged. Local Distributed Networks (LDN) and Local Area Networks (LAN) augment telecommunications, providing means for linking and networking remote information systems together. In fact, most recently [1] the use of light emitting diodes and single mode fibers in the local loop is being demonstrated. This application of fiber optics is attributed to new and revolutionary thought regarding the local loop architecture [2]. Recently, a fourth type of application, spinning off of the original telecom thrust, has emerged. This application is called the optical interconnect. Figure 1-1 depicts this. As shown in the figure, there are various types of optical interconnect genre, depending on the length of the interconnect (or distribution) and the switching speed employed. As one progresses to shorter interconnect lengths, higher speeds are implemented. In general, the photonic link becomes part of a signal processing (as opposed to telecommunications) structure.

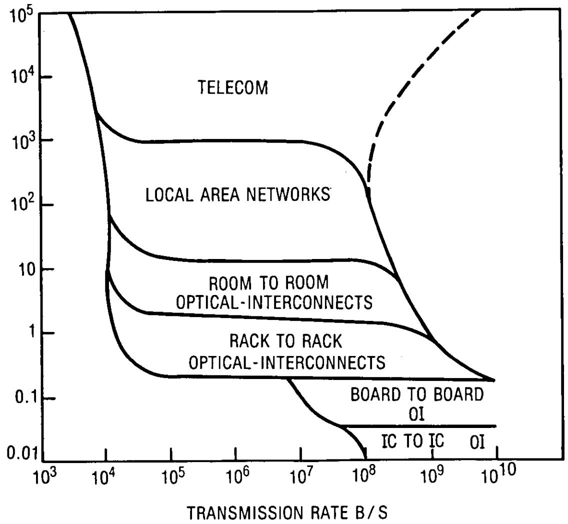


Figure 1-1. Plot of unrepeatered link lengths (in meters) versus transmission rate for photonic links; this plot shows the distinction between long haul telecommunications links, local area networks and the optical interconnect genre.

There is a threefold objective in the presentation of this paper. First, optical interconnects are to be clearly defined and partitioned. Second, it will be shown that optical interconnects are a natural consequence of the merging of telecommunications with information processing systems. Finally, optical interconnects will be presented as a viable tool for achieving high speed massively parallel digital processing.

The paper will be structured as follows; in section 2, optical interconnects will be defined. In section 3 some of the specific problems and limitations of high speed or massively parallel electronic interconnects will be discussed. Section 4 will be devoted to a discussion of various types of optical interconnect applications. Specific attention will be given to the problem of I.C.-to-I.C. interconnect. In section 5, a discussion of the use of integrated optoelectronics to perform high speed clock distribution is given. In this section, results of various specific efforts will be discussed.

CH2345-7/86/0000/0464$01.00 © 1986 IEEE

2. Defining Optical Interconnects

With merging of the telecommunications and computer industries over the recent years, new technical frontiers have been opened. Global information processing and transferral systems are emerging in both commercial and military scenarios. Fiber Optics, having its beginning in long-haul telecommunications, offers significant cost per channel-mile advantage over conventional coaxial systems. As information systems technology expands, other potential solutions are offered by photonics and integrated optics.

Figure 2-1 shows the types of transmission systems in which fiber optics offers solutions. Long haul telecommunications systems are necessary to transmit large blocks of data between remote information centers. As shown, transmission lengths are typically greater than 10 kilometers, and can be thousands of kilometers. One can think of these links as analogous to interstate highways on a roadmap. Data rates are typically very high, and channel capacity is a prime consideration. For these reasons, optical wavelengths at or above 1.3 microns are used exclusively. In this wavelength region, group III-V compound materials, such as InGaAsP-InP are required for sources and detectors.

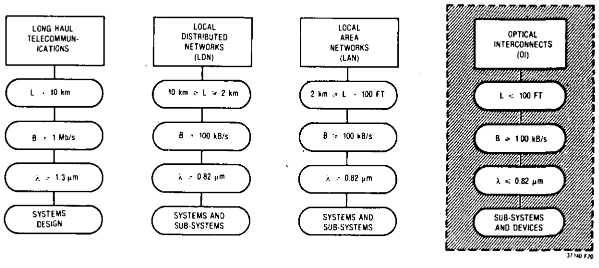


Figure 2-1. Breakdown of the types of fiber optic systems applications relevant to telecommunications.

Local distributed networks (LDN) imply the communications of two or more information centers within a 2km to 10km vicinity. Using the roadmap analogy discussed earlier, these networks serve the same purpose as intercity highways. Again, long wavelength devices are required.

Local Area Networks (LAN's) use even shorter links, and involve the networking of many computers, terminals and information processing equipment within a single information center. Ring systems, star systems, and some point to point transmission systems typify the LAN scene. On the roadmap analogy, LAN's can be thought of as intra-city streets and thoroughways. Both sub-system and system considerations are involved in LAN design.

Finally, a fourth type of application of fiber optics in information systems is called optical interconnects. Here data terminus points are linked over distances less than 100 feet. Point to point and star configurations are visualized. In the roadmap analogy optical interconnects compares with sidewalks, alleys and paths interconnecting individual structures.

The present role of optical interconnects in telecommunications can be summarized by describing the chronology of telecommunications over the recent past. Before the advent of fiber optics, telecommunications design dealt largely with the problems of bandwidth limited channels. The dominant source of transmission system degradation was the transmission line itself. Both coaxial and twisted pair transmission lines suffer from frequency dependent loss in addition to significant resistive loss. The introduction of the optical fiber radically altered this situation. Even when fiber systems were designed at 0.82 micron wavelength, significant improvements in unrepeatered transmission distance and cost per channel mile were accrued [3]. Still, systems design often dealt with dispersion limited channels [4]. The development of low loss fibers (1 to 2 db/km) and the move to 1.3 micron wavelength marked the beginning of distortionless channel transmission system design. For the first time maximum unrepeatered transmission distances were achieved in some systems without the use of equalization.

The next milestone in telecommunications evolution was set by the development of ultra-low loss single mode fibers operated at 1.55 micron wavelength. Systems operating at data rates exceeding 1Gb/s with unrepeatered spans exceeding 100 km were demonstrated in the lab [5]. With such capability the architecture of the entire local loop was rethought. The bandwidth available through the fiber made it feasible in principle to provide wideband services directly to the customer premise [6]. This concept has opened new and potentially potent market opportunities for telephone operating companies. In envisioning this objective, the local loop concept has been altered, as shown in figure 2-2. Distribution of broadband data to subscriber premises is achieved through interface with a remote hub.

At this hub complex signal processing and routing must be achieved at very high data rates (multi-Gb/s). Additionally, optical techniques such as wavelength division multiplexing may be implemented [2]. To support the bandwidth capacity of the fiber, it has become necessary to develop techniques for performing complex digital signal processing functions at multi Gb/s rates. Figure 2-3 shows an example of an interconnection network required for a high speed non-blocking switch array used in a local telephone distribution application. Inevitably, such functions are limited not by the fundamental switching capability of the I.C.'s themselves. Instead, the ability to interconnect the web of data and clock signals within a system poses the fundamental limit.

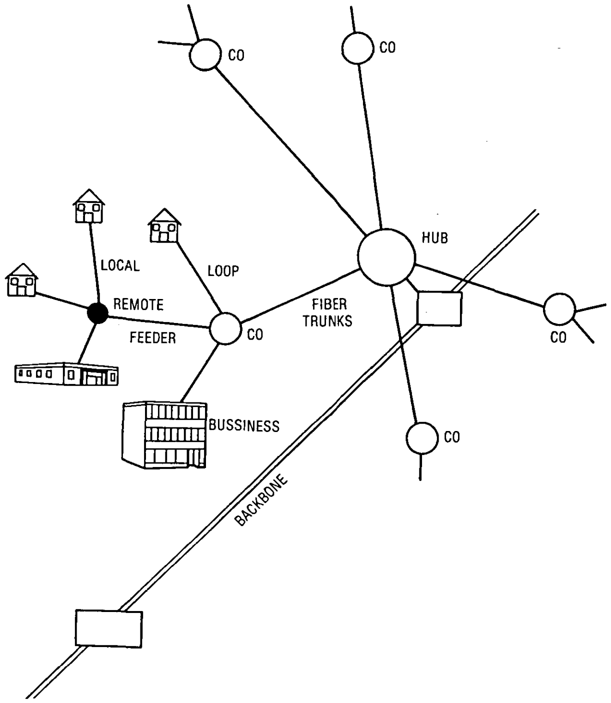


Figure 2-2. The hub exchange network concept; taken from Brackett [8].

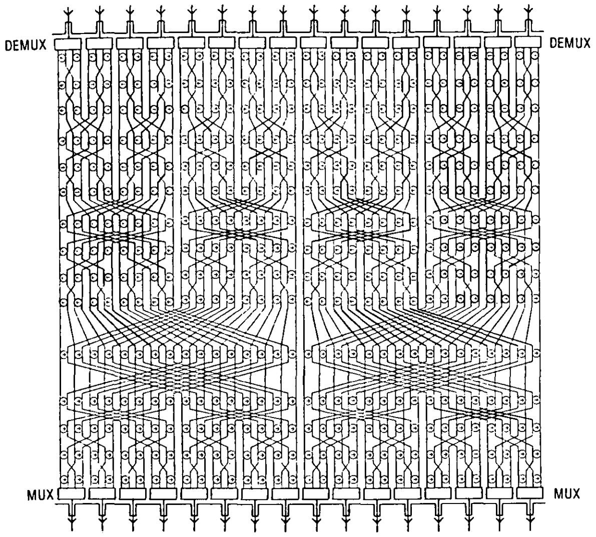


Figure 2-3. Example of a interconnection network required for a high speed non-blocking switch array.

Ironically, it appears that photonics will be an important expedient to the electronic bottleneck created by the fiber medium. Instead of offering a means of routing broadband serial data over kilometers, photonics now offer a way to distribute critical broadband signals over inches or less, on printed circuit boards.

It is possible to divide optical interconnect technology into at least four distinct genre. These genre are based on interconnection length, speed/bandwidth considerations and system application. The five types of optical interconnect scenarios cited are:

- Room-to-room interconnects
- Rack-to-rack interconnects
- Board-to-board interconnects
- I.C.-to-I.C. interconnects

Each of these genre are typified by their own set of applications, problems and solutions. The last application (I.C. to I.C. interconnects) is the primary subject of this paper. It is pointed out at the outset that this application includes various sub-genre, including intra-I.C. interconnects.

2.1 Room—to—Room Interconnects

As shown in figure 2-1, room-to-room optical interconnects are typified by point-to-point links at transmission rates as low as 10kb/s. Conceivably, transmission rates as high as several gigabits per second can exist. Link lengths are typically 100 feet or less. Longer links can occur but would then fall under the general classification of Local Area Network (LAN) links. In fact, some subjective overlap with LAN definition exists within the definition of room-to-room interconnects.

A typical application of room-to-room links is the low data rate computer link. Here issues such as EMI/EMP immunity, signal isolation or even common mode signal isolation may be important. Common mode isolation refers to situations where power and ground are defined differently in the source room and the receiver room. In this case the fiber optic link serves as a sophisticated opto-isolator.

An example of room-to-room optical interconnect situations is the remote tie-in link. Here two rooms may be isolated from each other electronically, for small signal isolation reasons. Or, the source room may be environmentally hazardous, causing EMI/EMP problems. Finally, the isolation situation may be based on security restrictions. In any of these situations, the data rates utilized may be as high as several hundred megabits per second.

2.2 Rack—to—Rack Interconnects

Rack-to-rack optical interconnects are a relatively new concept. However, specific system applications of rack-to-rack interconnects do exist in the field today. For example, AT&T ESS #5 electronic switching systems employs fiber optic rack-to-rack interconnects [7]. This system was implemented because of the reduction in size/weight achieved by replacing copper wire with glass fibers.

The optical rack-to-rack interconnect technology concept offers a fresh look at old problems. The fiber medium boasts unlimited signal bandwidth. This bandwidth is accompanied by a tenfold reduction in size and weight. Through· the proper use of signal multiplexing techniques, the bandwidth capacity of the fiber can translate to overall reduced power consumption. In complicated electronic signal processing scenarios, the rack-to-rack optical interconnect concept allows the designer to treat several networks of on-board computers and data processing equipment on a modular basis. The issues of signal processing and module interconnect/signal transmission can more easily be separated.

For the rack-to-rack interconnect application, the optical fiber is the most useful optical channel. Because fiber losses are so low at optical wavelengths exceeding 1.1 microns (particularly at 1.3 microns and at 1.55 microns), many of the backplane interconnects can be passive. That is, sufficient optical power exists to route signals entering backplane environments from outside without regenerating them. The use of mechano-optical switch matrices or even electro-optic switch matrices may augment the rack-to-rack optical interconnect concept.

2.3 Board–to–Board Interconnects

Figure 2-4 shows three examples of board-to-board optical interconnects. As shown, on much signal processing or generating equipment, there are several P.C. boards physically attached to the controller board, or backplane. Conventionally, all communication between boards is achieved by electrically interconnecting from one P.C. board, through the controller board, to another P.C. board. This practice often leads to limitations on the amount of signal isolation achievable, and on the bandwidth achievable. As shown in the figure, one method used for avoiding the use of the backplane is to communicate from board-to-board via an optical free space link. By attaching an optical transmitter and receiver module onto identical places on the P.C. board, the data link is automatically completed. Using LED's and PIN diodes, data rates up to 100 Mb/s are achievable. By replacing the LED with laser diodes, 1 Gb/s or higher may be achievable.

In situations where high speed and signal isolation requirements are both very severe, a simple fiber link can accomplish the board-to-board interconnect task. This is also shown in figure 2-4. In these situations the issues of compact design for both the transmitter and receiver module become important. Integration of the electronic and optical functions on these modules begins to take on design relevance. The combined requirements of size-weight-power limitations and high reliability/survivability dictate integration.

2.4 I.C. I.C. and Intra–I.C. Interconnects

I.C. interconnects (inter- and intra-) constitute the least mature and most important of the four types discussed. Presently, needs for high speed, parallel processing equipment constitute the largest technological bottleneck in telecommunications and computing technology. Group III-V materials (particularly GaAs and InP) are forming the basis for high speed digital I.C.'s and optoelectronics. The entirety of telephony architecture is changing to support photonics technology [8]. With this change, the much needed commercial market for GaAs digital MMIC's is rapidly emerging. Meanwhile, supercomputer architecture design is calling for advanced digital I.C. technology as well [9], [10]. Several researchers have already realized that once the high speed digital I.C. bottleneck is overcome, the problems of data distribution and interconnections at these speeds is the next bottleneck.

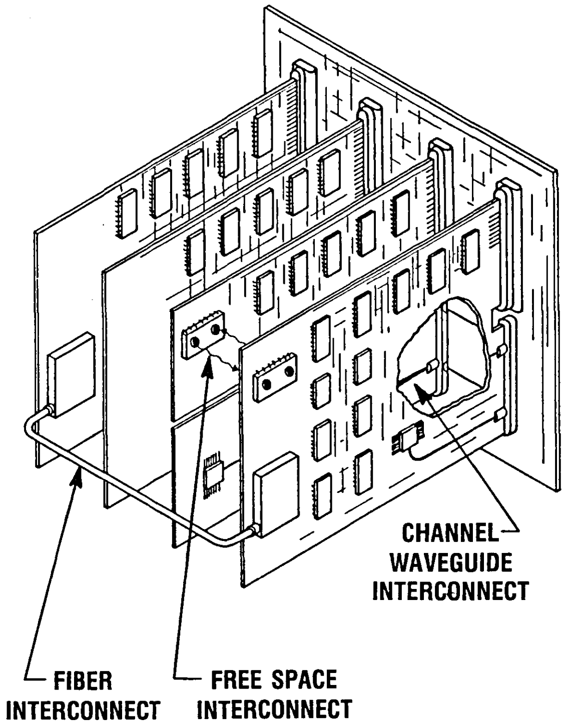


Figure 2-4. The backplane interconnect concept, showing three potentially viable techniques for achieving the interconnect.

One of the most illustrative examples of this situation is the clock distribution problem. Beyond 200 MHz, gate delays, transmission line imperfections and topology constraints pose limitations on clock distribution within a sub-system of I.C.'s. These limitations often constitute the weak link in achieving a switching speed goal. Beyond 1 Gb/s all of these problems are further exacerbated.

Photonics offers new insight into the solutions of these very serious problems. The fiber medium is a low loss, low dispersion channel. Over the short links discussed here (inches or less), the channel is virtually bandwidth unlimited. The transport of baseband information over a fiber can be accurately thought of as the modulation of a carrier oscillating at approximately 2×10^{14} Hz. For this type of modulation the carrier need not be coherent, since direct detection methods are usually employed. With a 2×10^{14} Hz carrier, even extremely wideband digital signals (for example 10 Gb/s) convert to narrowband transmission within the fiber. Even over inches of fiber, millions of carrier wavelengths are traversed. Therefore, transmission line matching becomes analogous to narrowband microwave matching. Both data and clock distribution can be achieved optically by implementing techniques common to narrowband microwave electronics.

The use of fibers constitutes only a small subset of the entire optical interconnect media. For both clock and data distribution, integrated optic transmission lines or ultimately, free space interconnects using dynamically reconfigurable holographic routing [10] may be implemented.

3. The Electrical Interconnect Problem

As a prelude to discussions on optical interconnects, this section discusses methods used in high speed broadband electrical interconnects. Also, some fundamental limits to the electrical interconnect of baseband digital signals are postulated.

3.1 Interconnects from the Device Viewpoint

When designing very high speed digital signal processing equipment, one finds that physical design and electrical design are inseparable. Because of this, high speed physical layouts are unique in appearance. First, metal flat packages are used exclusively in the design. These packages make efficient use of available space, and simplify I.C. die bonding and wire bonding. Furthermore, wire bond length is kept to a minimum; the input leads, when laid down on the printed circuit board closely resemble 50 ohm transmission lines. Second, circuit layout is planar. Circuit layout planarity and the use of ground planes has been found to be essential in high speed design. The backplane concept finds limited use as target system speeds increase beyond 1 Gb/s. Finally, the interconnects between I.C.'s are accomplished using microstrip transmission lines. Delay lines are incorporated onto the p.c. layout. Delays as long as one half a bit length ($\sim 1ns$, or 15 cm at 500 Mb/s) are often required.

Figure 3-1 depicts high speed layout topology, pointing out some of the difficulties one encounters. Impedance mismatches at bends and splits tends to appear as a capacitive load at the junction's bend, giving rise to high frequency reflections. Termination mismatches are the cause of several problems. First, it is difficult to maintain a 50 ohm transmission line impedance from one transmission line to the next. This difficulty is attributable to photolithographic variations and variations in p.c. board thickness and dielectric constant. Resistive mismatches of 10% are common. Second, the presence of parasitic capacitance at the package and the I.C. (bonding pads and wire bonds) can cause reactive impedance mismatches far exceeding 10%.

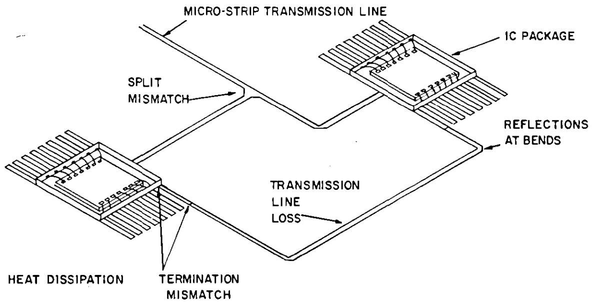


Figure 3-1. Schematic of a typical high speed electronic interconnect and distribution layout, depicting some of the problems associated with such interconnects.

Figure 3-2 shows plots of calculated loss in db/cm for microstrip transmission lines on various dielectric materials. In these calculations, both conductor loss (skin effect) and dielectric loss (for homogenous materials) are included. As shown, materials such as alumina ceramic or quartz are conductor loss limited. This situation is caused by the relatively low resistivity typical of these materials. The higher resistivity dielectric materials more commonly used in p.c. board interconnects are not as lossy as their silicon counterparts. However, the loss mechanism is proportional to $f^{1/2}$ and is most pronounced between 0.1 and 4 GHz. For a broadband high data rate digital pulse stream, the result is pulse distortion in addition to attenuation.

For comparison purposes, the loss characteristics of plastic fibers (1000 db/km or 0.01 db/cm) and glass fibers (10^{-5} db/cm) are shown. Also, reported losses for thin film plastic channel waveguides [11] are shown. It should be evident that consideration of optical waveguides for interconnects involves a trade-off between transmission loss and ultimate manufacturing simplicity. Where one avoids the deleterious effects of transmission line imperfections on microstrip lines, one must consider the extra losses incurred due to bends and optical coupling.

3.2 Interconnect from the Systems Viewpoint

A survey of existing digital switching systems and subsystems suggests a reciprocal relationship between attainable switching speeds and inherent signal processing complexity. Most existing systems switching at Gb/s rates or higher are considerably simpler in design than their lower speed (but perhaps parallel processed, high bit rate) counterparts. This concept holds at the equipment level, the board level, and even at the I.C. level. At the equipment and board level, line lengths (i.e., latency) tend to increase, quenching attempts at attaining higher complexity. Also, the limitations imposed by the interconnect medium pose fundamental problems. But, even at the I.C. level, the reciprocal relationship still holds. For example, at

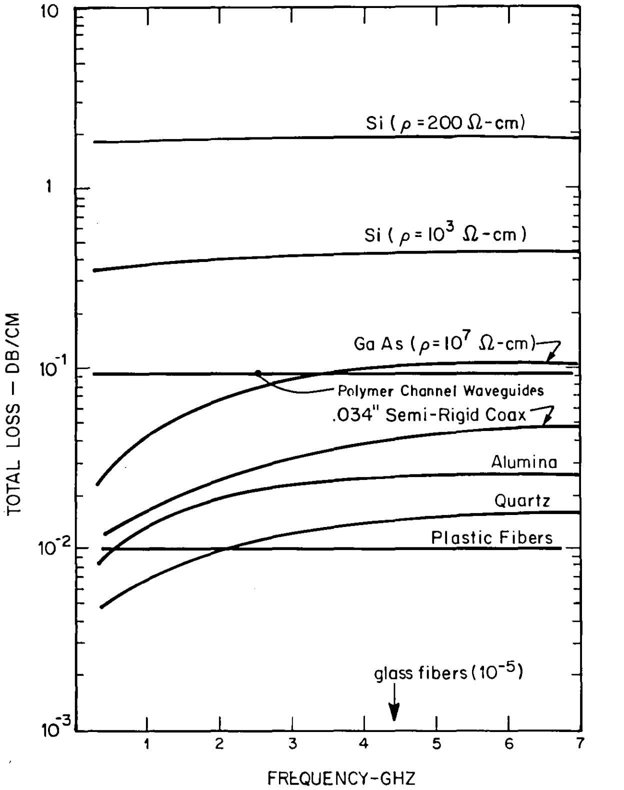


Figure 3-2. Plots of calculated loss (db/cm) for microstrip transmission lines on various dielectric materials, and for optical waveguide material.

speeds beyond 2 Gb/s, GaAs materials technology must be implemented. Because of processing constraints, most GaAs circuits are still relatively simple. Existing GaAs I.C. chips are limited in their ability to dissipate power, and also in their pinout capability. Heat dissipation in turn limits package size reduction, leading to the latency problems discussed earlier.

Using Rent's rule, it is possible to derive a semi-quantitative expression of the bandwidth/complexity relationship. This is done by relating average interconnect length, derived from Rent's rule, to the latency in signal propagation on a printed circuit board. The reader is referred to a previously published paper [12] for details of the calculations.

Figures 3-3 and 3-4 give the results of these calculations. In figure 3-4 is a plot of system switching rate vs. level of parallelism. Depicted is the point at which latency restrictions cause electrical interconnections to become prohibitively complex. It is at that point that optical interconnects can offer significant improvement over purely electrical approaches. An example of the meaning of the parameter M is that of a parallel processed system in which 8 140 Mb/s channels are multiplexed to yield a data rate, D, of 1120 Mb/s; however the actual switching rate is still 140 Mb/s. In this case, B = 140 Mb/s and M = 8. From the figure it is clear that, from a signal latency stand point, system switching rates must exceed 1 Gb/s before optical interconnects offer a viable alternative to electronic technique.

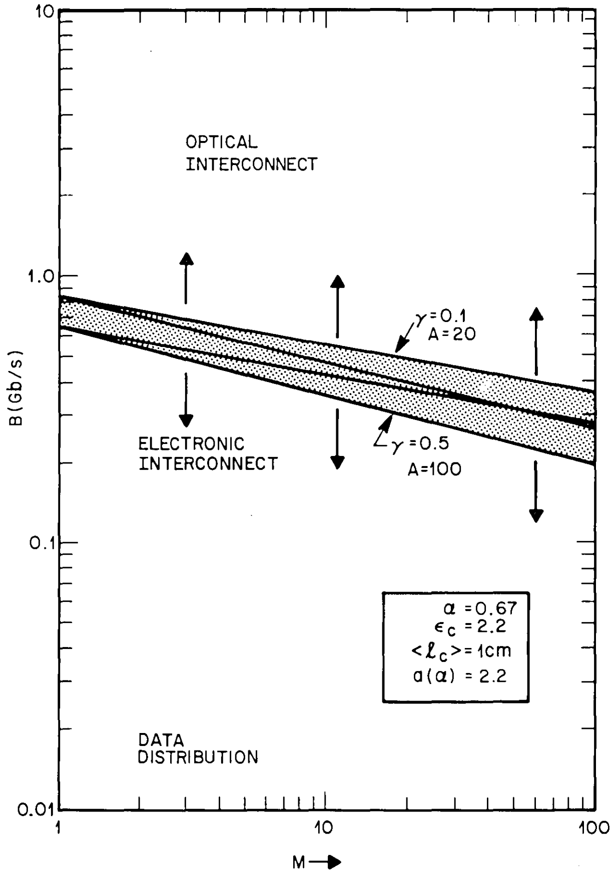


Figure 3-3. Plot of fundamental bit rate at which signal latency becomes an issue in data distribution as a function of M, the circuit level of parallelisms.

Figure 3-4 shows a similar calculation result for the case of clock distribution in synchronous digital systems. Here the system clock rate, B, is plotted vs. L, the printed circuit board size (a measure of system complexity).

As shown, clock distribution systems appear to be more sensitive to latency effects. Systems operating at 500 Mb/s and up may benefit from optical clock distribution. This is why much of the experimental accomplishments to date [13], [14], [15], have been in clock distribution.

4. Optical Interconnect Link Structure

Having discussed some of the important electronic issues in high speed digital interconnects, it is appropriate to look at the structure of an optical interconnect link. Figure 4-1 shows a functional block diagram of a generic optical interconnect link. Reference to the figure shows that both receiver and transmitter are sub-miniaturized.

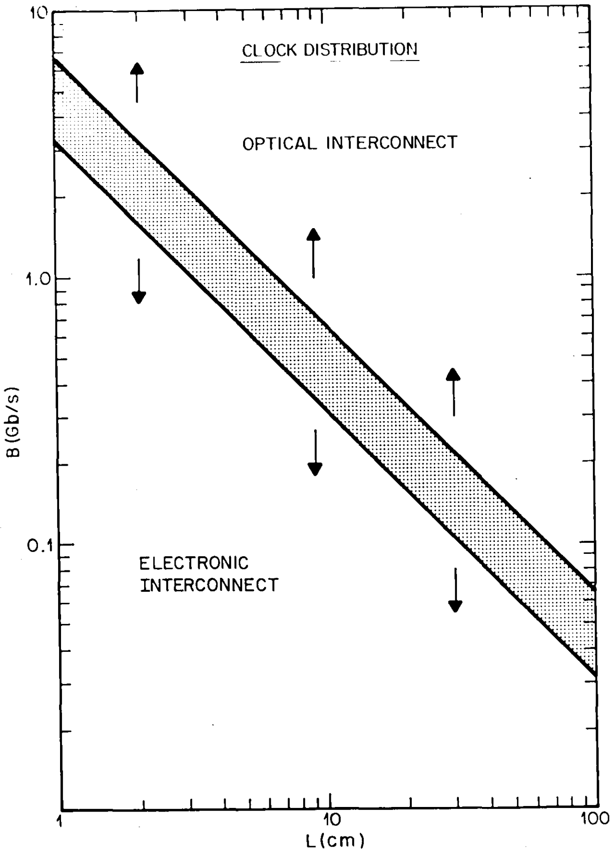


Figure 3-4. Plot of clock rate, B, at which latency becomes an issue in clock distribution, as a function of L, circuits board linear dimension.

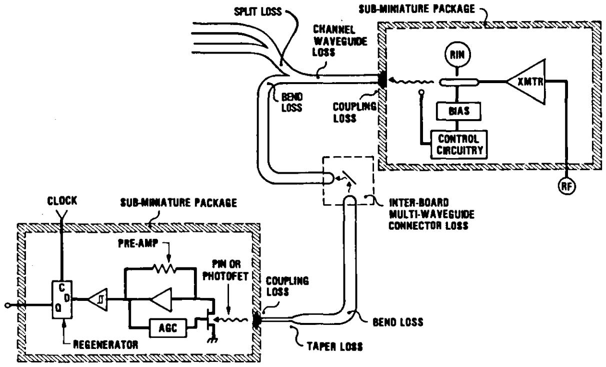


Figure 4-1. Functional block diagram of a generic optical interconnect distribution link.

The sub-miniaturization constraint eliminates standard telecommunications links from the domain of link genre. The type of optical interconnect medium is not specified in the figure (plastic fiber, glass fiber, plastic channel waveguides or even free space are possible media). Inter-board connectors are shown on the diagram, but they need not be present in many of the (intra-board) link structures considered in this paper. Other assumptions embedded in the figure include (1) the assumption of very high speed digital (baseband) links, (2) external system clocking (i.e., no timing recovery) and (3) liberal use of monolithic opto-electronic integration, especially on the receiver end.

In board-to-board and I.C.-to-I.C. interconnect design, high speed and signal isolation are often key motivating factors. To render optical interconnects useful the electro-optic and opto-electronic interfaces must take up very little space. Otherwise circuit pack real estate will be dominated by electro-optics. This situation will severely limit the space available for signal processing.

In addition to the architecturally based requirements for small size, there are also requirements based on the high speed nature of the interconnect signal. At speeds beyond a few hundred Mb/s, package design cannot be separated from electronic circuit design. The parasitic capacitances and inductances of the packages become an intimate part of the electronic circuit. These points are particularly true for broadband digital signals.

Finally, reliability considerations also drive the design of the link. Subminiaturization must be achieved with no compromise in reliability. The use of monolithic opto-electronics in these packages (particularly the receiver) offers the long term potential of enhancing system reliability through reduction of the number of physical interconnections required.

Although the optical fiber has many virtues for use in telecommunications, its use as an optical channel for optical interconnect applications is not as clear. While the optical fiber is an ultra-low loss information channel, with virtually unlimited bandwidth capacity, there are some drawbacks to the use of fibers in optical interconnect applications. Because one of the important issues in optical interconnects is packaging and sub-miniaturization, the placement of the optical channel on an electronic substrate becomes a very desirable goal. Also, the optical and mechanical coupling of the channel to the opto-electronic or electro-optic interface is a major issue. To render a very high speed system utilizing optical interconnects useful in real situations, the system must be easily and reproducibly manufacturable. If a system is largely an electrical one, with perhaps one or two point-to-point optical links, then the use of fibers constitutes a sensible choice. However, if the system requires liberal use of optical interconnects to render it tenable, then the utility of the fiber medium diminishes rapidly.

4.1 Receiver Design Considerations

Unlike a lightwave telecommunications receiver, an optical interconnect receiver is basically a high speed opto-electronic transducer. For optical interconnect applications, the main object is to receive the incoming optical data stream, transform it to electronics, amplify (and filter) it and re-digitize it. This function must be achieved with a minimum expenditure of board area with acceptable reliability. Since a fully retiming-reshaping-regenerating (R^3) receiver can take up as much space as a full printed circuit board or more, and makes use of lumped circuit elements, the R^3 approach is not feasible. Instead, it must be assumed that the system clock will be distributed separately (perhaps optically), and that pulse shaping to achieve raised cosine pulse shapes will not be performed. In spite of (and to some degree, because of) these limitations, very low bit error ratios (BER required will typically be 10^{-17}) and high signal-to-noise ratios will be required. Therefore, there is a conflict between the requirement for relative circuit simplicity and high sensitivity. The usual trade-offs between bandwidth, sensitivity, dynamic range and power consumption still exist. The monolithic opto-electronic integration of front end opto-electronic receivers can help to reduce the severity of these conflicting requirements.

The problem of optimized direct detection optical receiver design for digital applications has been treated quite comprehensively by Smith and Personick [16]. Fundamentally, for direct detection of optical binary signals, the Gaussian approximation for noise distributions constitutes a reasonably accurate assumption. Under these circumstances, the probability of incurring a detection error is given by

$$P(E) = \frac{1}{\sqrt{2\pi}} \int_Q^\infty e^{-x^2}\, dx \,. \tag{4.1}$$

The quantity Q is defined as,

$$Q = \frac{|D - a_i|}{\sigma_i} \,. \tag{4.2}$$

Here, D is the decision threshold, a_i are the mean signal levels for the 0 and 1 state, and the σ_i are the noise terms for the 0 and 1 state. With a binary digital system, then, equation 4.2 constitutes two equations, one for each state. From equations 4.2, any two of the 4 variables D, Q, a_0 and a_1 can be determined once the other two are specified. The quantities a_0 and a_1 are related to each other through the laser diode extinction ratio,

$$a_0 = r\,a_1, \qquad r \ll 1 \,. \tag{4.3}$$

These quantities can also be related to the average converted optical power $\eta \overline{P}_0$, at the photodetector through the relation,

$$\eta \overline{P}_0 = \tfrac{1}{2}(a_0 + a_1) \,. \tag{4.4}$$

Solving for a_0 and a_1 gives,

$$a_0 = 2\eta \overline{P}_0 \left[\frac{r}{1+r}\right] \; ; \; a_1 = 2\eta \overline{P}_0 \left[\frac{1}{1+r}\right] \,. \tag{4.5}$$

Using eq. (4.5) in equations (4.2) expresses the decision equations in terms of the two variables, $\overline{P}_0$, the minimum detectable optical signal for a given BER, and D, the decision threshold. These equations are,

$$D - 2\eta \overline{P}_0 \left[\frac{r}{1+r}\right] = Q\,\sigma_0 \,, \tag{4.6}$$

$$2\eta \overline{P}_0 \left[\frac{1}{1+r}\right] - D = Q\,\sigma_1 \,. \tag{4.7}$$

Equations (4.6) and (4.7) provide the fundamental starting point toward receiver sensitivity calculations. The terms σ_0 and σ_1, the variances in the signal levels in the zero and one state, are specific to the receiver design.

In optical interconnect receiver design, there are four basic types of noise. These are,

1. Signal independent receiver noise current sources.
2. Signal independent receiver noise voltage sources.
3. Signal dependent quantum noise (Poisson Distribution).
4. Signal dependent laser diode intensity fluctuation (non-Gaussian).

Types 1 and 2 contain contributions from various elements in the receiver circuit; for the most part they are thermal noise sources and shot noise sources. The voltage sources are similar, and include flicker noise at the receiver front-end, etc. Type 3 refers to the quantum limit of signal detection and any intersymbol interference terms. Type 4 is commonly disregarded. It refers to basic intensity fluctuation in the optical output of laser diode. Some contributions to these terms are more strongly signal dependent than others. They are all lumped together as relative intensity noise (RIN). The presence of signal dependent RIN significantly alters the form of sensitivity equations, and also the fundamental results in the case of high speed optical interconnects.

RIN is defined in terms of square of the optical fluctuations at the output of the device. Specifically,

$$\mathrm{RIN} = \frac{(\Delta P_0)^2}{P_0^2} \,, \tag{4.8}$$

where P_0 is the laser output power. Sato et al [17] have measured the RIN of various laser diodes as a function of laser bias-to-laser threshold voltage. These measurements, redrawn on a log-log plot, are shown on figure 4-2. The figure shows that the RIN for various structures follows an inverse power function dependence on the quantity $[I_D/I_T - 1]$, with a power index given by the quantity γ.

Typically, γ lies between 2.5 and 4.5, depending on the

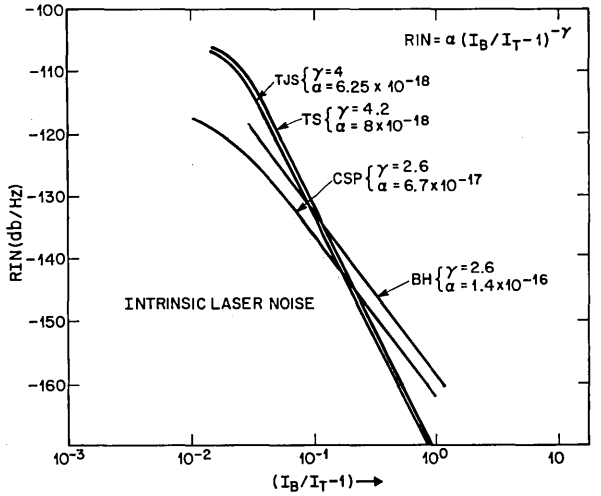


Figure 4-2. Plots of measured RIN for various laser diode structures, as a function of laser bias. The data was obtained from Sato et al [17] and redrawn in log-log fashion.

structure. Through inspection of figure 4-2, two conclusions can immediately be drawn; (1) Intrinisic RIN is strongly signal pattern dependent; S/N is much higher in the "one" state than in the "zero" state. (2) Conventional values of r (i.e. $r \simeq 0.1$ to 0.01) yield quite low S/N ratios at high data rates; this is particularly true for gain-guided devices. Therefore, a potential limit on digital signal fidelity at the source-end exists. This limit clearly depends on the digital data rate and on the system extinction ratio.

Using the equations discussed above, and a simple piecewise linear model for laser diode light output vs. bias current input, it is possible to model the mean square fluctuations in laser output as they appear at the receiver front-end. From this model one can rewrite equations 4.6 and 4.7 explicitly in terms of the signal dependent laser noise term and all of the conventional circuit noise terms normally present at the receiver front-end. Because of the inverse power functional dependence of laser noise on laser optical output, these equations represent coupled transcendental equations, whose solution must be determined numerically. Combining equations 4.6 and 4.7 and eliminating D yields a function of the form,

$$f(\eta \overline{P}_0) = 0 \, . \qquad (4.9)$$

The details of this function are presented in a separate publication [12]. Figure 4-3 shows plots of $\eta \overline{P}_0$ vs. transmission rate for various values of r. Also shown for comparison are plots of $\eta \overline{P}_0$ vs. B for an optimized telecommunications receiver (BER = 10^{-9}) for which the front-ends were hybrid (curve number 5) and monolithic (curve number 4).

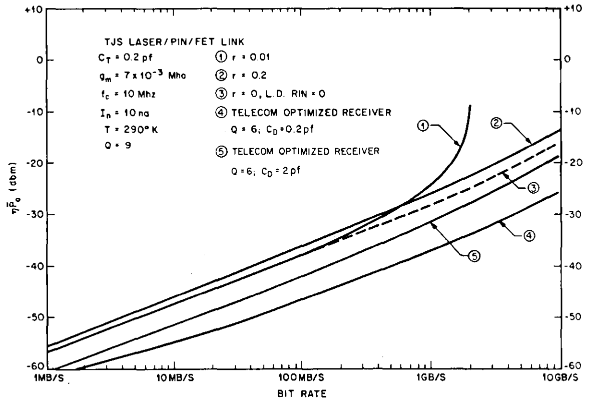


Figure 4-3. Plots of $\eta \overline{P}_0$, the minimum detectable signal for the specified bit error ratio, as a function of digital transmission rate, B, for various transmission scenarios.

Inspection of this calculated data reveals several interesting points. First, the effects of intrinsic laser noise can be dramatic at very high data rates. If r is chosen incorrectly, bit-error-ratio saturation can occur (c.f. curve number 1). In fact, even if r is chosen for optimum conditions, bit-error ratio saturation can still occur, but at a higher data rate. Secondly, there appears to be an optimum value of r. This is shown in Fig. 4-4. Here $\eta \overline{P}_0$ is plotted vs. r at B = 1.5 Gb/S. There is a clear (but fortunately wide) minimum in $\eta \overline{P}_0$ vs. r. This points to the necessity of designing control circuitry that will ensure the stability of the laser threshold current and the laser modulation current relative to each other.

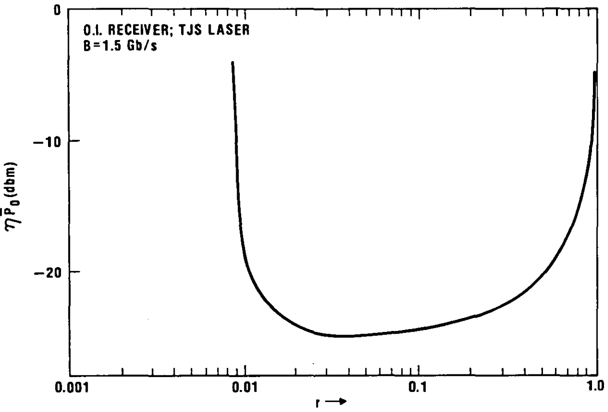


Figure 4-4. Plot of $\eta \overline{P}_0$ vs r for 1.5 Gb/s transmission, showing the existence of an optimum value of r, the laser diode extinction ratio.

Further, the sensitivity that one can expect from a receiver designed to optical interconnect constraints cannot be as high as that of an optimized telecommunications receiver. The differences can be accounted for in the higher BER objective, the wider signal bandwidths required (i.e. larger normalization integral values) and the laser noise term. Finally, and very importantly, curve number 4 in figure 4-3 shows what can be achieved through monolithic front-end integration. Clearly, the monolithic front-end design embodies the potential of making up a considerable portion of receiver sensitivity lost in complying with optical interconnect design criteria. Concurrently, the monolithic approach actually facilitates the achievement of sub-miniaturization; which has influenced the optical interconnect design so strongly. Therefore, monolithic opto-electronic integration adds an element of synergism to optical interconnect design.

4.2 Optical Interconnect Link Analysis

To complete this discussion on optical interconnects, a first-cut look at some typical optical interconnect links is now made. The instrument for looking at optical interconnects is the optical power budget. Table 4-1 shows a summary of the basic types of systems considered. As shown, laser diode and LED based links are both considered. Three different data rates are studied; these are 565 Mb/s, 2.26 Gb/s and 4.52 Gb/s. Short wavelength ($\lambda = 0.82 \mu m$) transmission is considered exclusively. This is done because of the maturity of laser diode technology, amenability of silicon technology at the receiver end and due to the low-loss characteristics of plastic channel waveguides at $\lambda = 0.82 \mu m$. Fanout, or distribution, is considered a major system issue, since point-to-point links can most easily be accomplished electronically; even at very high data rates.

	Link # 1	Link # 2	Link # 3	Link # 4	Link # 5	Link # 6
Transmission Rate (Gb/s)	0.565	2.62	4.52	0.565	2.62	4.52
Data Source	LD	LD	LD	LED	LED	LED
Link Length (cm)	25	25	25	25	25	25
Optical Wavelength (μm)	0.82	0.82	0.82	0.82	0.82	0.82
Photodetector	p–i–n $D = 150\mu m$	p–i–n $D = 100\mu m$	p–i–n $D = 100\mu m$	p–i–n $D = 150\mu m$	p–i–n $D = 100\mu m$	p–i–n $D = 100\mu m$
Front-End Electronics	GaAs FET or Si Bipolar	GaAs FET	GaAs FET	GaAs FET or Si Bipolar	GaAs FET	GaAS FET
Monolithic Front-end?	Yes	Yes	Yes	Yes	Yes	Yes
Channel Waveguides?	Yes	Yes	Yes	Yes	Yes	Yes

Table 4-1. Optical Interconnect Link Types.

Laser based optical interconnects represent the most obvious choice of links. With direct drive, the laser diode can be on full on/off modulated at multi-Gb/s rates. Peak output power of +10 dbm is comon for laser diodes. The potential for even higher peak output exists, although thermal dissipation problems become more severe.

Table 4-2 shows optical power budgets for laser based links at 565 Mb/s, 2.26 Gb/s and 4.52 Gb/s. Source power and link loss elements are tabulated in the upper portion of the table. All optical power terms are given as *average* quantities (with a 50% duty cycle, these values are 3 db below peak quantities). Using the receiver sensitivity calculations from the previous section, minimum detectable signals ($\eta \bar{P}_0$) are then given.

SYSTEM PARAMETER	B = 565 Mb/s	B = 2.26 Gb/s	B = 4.52 Gb/s
Laser Output Power (Avg)	+7 dbm	+7 dbm	+7 dbm
Laser-to-waveguide coupling loss (db)	-2	-2	-2
Distribution loss (db)	TBD	TBD	TBD
Waveguide loss (db, @ 0.2 db/cm)	-5	-5	-5
Waveguide bend loss (db)	-2	-2	-2
Waveguide-to-detector coupling loss (db)	-2	-3	-3
Total power incident at detector (not including distribution loss).	-4 dbm	-5 dbm	-5 dbm
O.I. Receiver Minimum detectable signal ($\eta \bar{P}_0$, in dbm)	-28 dbm	-22 dbm	-18 dbm
Net Optical Gain Available (db)	24 db	17 db	13 db
Margins and Degradation Factors:			
Detector d.c. Quantum Efficiency (db)	-1	-1	-1
Detector a.c. Q.E. [monolithic receiver assumed] (db)	-1	-1	-1
Power Rail Degradation (db)	-2	-3	-3
Laser diode EOL Margin (db)	-2	-2	-2
Useful Optical Gain Available (db)	18	10	3
Maximum distribution possible	64	10	4

Table 4-2. Optical Interconnect Link Budget - Laser based system.

The difference is the net optical gain available (in dbs). Next, system margins and degradation factors are subtracted from the net optical gain. The result is the useful optical gain. As presented in the table, no distribution (or fanout) is assumed. Instead, after the useful optical gain is determined, the maximum possible fanout is determined from that number. Clearly, the distribution loss must include power splitter insertion loss as well as the simple power division term. Therefore, the number derived in the calculation represents an upper bound on distribution.

Link loss terms included are source-to-waveguide coupling loss, waveguide intrinsic loss, waveguide bend loss and waveguide-to-detector coupling loss. The reader is referred to [15] for a more detailed explanation of the values assumed. From these numbers and receiver sensitivity values (c.f. figure 4-3) the net optical gain available at the receiver input is 24 db, 17 db and 13 db for 565 Mb/s, 2.26 Gb/s and 4.52 Gb/s operation, respectively.

Next, an attempt is made to include other systems margins and degradation factors. The first term allocates 1 db to photodetector d.c. quantum efficiency. An additional 1 db is allocated to account for the difficulty in producing OEIC's with sufficient photodetector photon collection depth to prevent substrate photo current production. The power rail degradation term is discussed in [12]. Finally, to account for the slow degradation of laser diodes over their lifetime, a 2 db end-of-life margin is included.

After these margin and degradation terms are included, useful optical gain at the three data rates is 18 db (565 Mb/s), 10 db (2.26 Gb/s) and 6 db (4.52 Gb/s). If all this available margin is allocated to distribution of signals to various nodes, then the maximum distribution possible is 64 (565 Mb/s), 10 (2.26 Gb/s) and 4 (4.52 Gb/s).

As already discussed, laser diodes will most likely represent the element of choice in future optical interconnect links. However because of the fact that laser diode reliability is not yet firmly established at the 10^7 hr mark or higher, (and variances are still high), alternative sources such as LED's may be attractive in some instances. Therefore we present link budgets for LED based systems as well as laser diode based links. Table 4-3 summarizes the budget for LED based links. The first, and most significant difference between LED and laser diode based links is in the output power available from an LED. Currently, edge-emitting LED's have been fabricated capable of emitting -4 dbm of optical power into free space at 565 Mb/s.

SYSTEM PARAMETER	B - 565 Mb/s	B - 2.26 Gb/s	B - 4.52 Gb/s
Laser Output Power (Avg)	-4 dbm	-11 dbm	-14 dbm
Laser-to-waveguide coupling loss (db)	-3	-3	-3
Distribution loss (db)	TBD	TBD	TBD
Waveguide loss (db, @ 0.2 db/cm)	-5	-5	-5
Waveguide bend loss (db)	-2	-2	-2
Waveguide-to-detector coupling loss (db)	-2	-3	-3
Total power incident at detector (not including distribution loss).	-16 dbm	-24 dbm	-27 dbm
O.I. Receiver Minimum detectable signal ($\eta\overline{P}_0$, in dbm)	-31 dbm	-24 dbm	-21 dbm
Net Optical Gain Available (db)	15	0	0
Margins and Degradation Factors			
Detector d.c. Quantum Efficiency (db)	-1	-1	-1
Detector a.c. Q.E. [monolithic receiver assumed] (db)	-1	-1	-1
Power Rail Degradation (db)	-2	-3	-3
Laser diode EOL Margin (db)	-1	-1	-1
Useful Optical Gain Available (db)	10	0	0
Maximum distribution possible	10	0	0

Table 4-3. Optical Interconnect Link Budget - LED based system.

Suzuki et al [18] have reported the fabrication of LED's operating at 2 Gb/s with an average output power of -11 dbm. To date no other devices have been reported with modulation capabilities beyond this. Because of the wider angular power distribution of LED's, coupling to waveguides is not as efficient as with laser diodes. However, when using large feature size channel waveguides (typically $150\mu m \times 75\mu m$), coupling losses of 3 db should be achievable. All other loss terms are expected to be similar to those for laser based links. Therefore, the total power incident at the photodetector will be approximately -16 dbm (565 Mb/s), -24 dbm (2.26 Gb/s) and -27 dbm (4.52 Gb/s). Next, it is noted that receiver sensitivity values are slightly better for the LED based system. This reflects the absence of source noise effects (c.f. curve no. 3 in figure 4-3). Combining these sensitivity values with the incident power reveals that the only data rate at which power margin exists is 565 Mb/s. Then, subtracting the same degradation factors and a 1 db end-of-life margin for the LED yields 10 db useful optical gain available at 565 Mb/s. This translates to a maximum distribution of 10.

Inspection of tables 4-2 and 4-3 reveals several interesting points. Most obvious is that as higher bit rates are approached, the available optical gain for distributing signals vanishes rapidly. This occurs because the minimum detectable signal increases monotonically with bit rate, and because the available power from optical sources is limited. Systems degradations and necessary margins cut deeply into this diminishing gain. One of the key motivations for device design in optical interconnects, then, is to find ways to remedy this situation. There are several ways to do this. The first and most obvious one is to provide more optical power at the source. In spite of the fact that high power lasers are being designed [19], they must be accompanied with low threshold currents, low noise properties and agile modulation characteristics. Even when this is accomplished, the obstacles to providing acceptable broadband performance at multi gigabit per second rates, while dissipating heat acceptably in an sub-miniaturized package, are formidable.

At the receiver end, monolithic design is clearly a necessary ingredient for optical interconnects. Monolithic OEIC receivers are the only practical means for reducing input parasitics, thereby achieving acceptable sensitivity and speed in a sub-miniaturized package design. Figure 4-5 summarizes the results of this section. Here receiver sensitivity ($\eta\overline{P}_0$) and transmitter output power are both plotted vs. data rate, B. Curves for both LED's and laser diodes are shown. Clearly the optical margin stays open longer for laser diode-based systems.

The dotted lines for laser diode transmitters beyond 5 Gb/s reflect the level of understanding regarding full on/off modulation at these data rates and at high optical powers. Clearly, at very high data rates, external modulation of laser diodes may become a necessity, in spite of the serious packaging issues. Furthermore, to achieve high power output, a shift to the longer optical wavelengths, where optically induced damage in $LiNbO_3$ is not as serious a problem, may be in order. Methods for reducing some of the systems degradations terms are also desirable.

If any of these measures are successful, the optical interconnect concept can carry viability with it beyond several Gb/s, where it will be most needed. As data rates move toward double digit gigabit rates, circuit subminiaturization will continue, and the free space

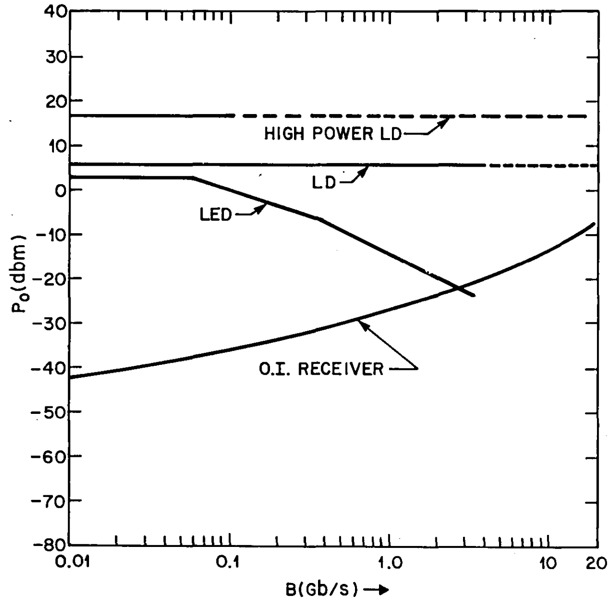


Figure 4-5. Receiver sensitivity for O.I. applications, transmitter power, both plotted as a function of B, the transmission rate. The figure depicts the closing of the optical power margin with increasing transmission rate.

implementation of optical interconnects will become more important. However, it is asserted here that many of the same constraints (e.g. receiver sensitivity, fanout limitations, thermal dissipation) will still set fundamental engineering limits to what can and cannot be done in high speed signal processing at these rates.

5. Integrated Opto–Electronics for Optical Interconnects

In board-to-board and I.C.-to-I.C. interconnect situations, high speed and signal isolation are often key motivating factors. The issues of compact design for both transmitter and receiver modules become very important. Integration of optics with electronics offers not only size/weight reduction advantage, but performance enhancement as well. Monolithic integration of optics and electronics (known as integrated opto-electronics or integrated electro-optics) represents the ultimate in device integration. Monolithic receivers offer the potential of enhanced bandwidth and sensitivity (due to reduced input parasitics). Monolithic transmitters offer fanout and signal multiplexing advantages. Used together, the two offer a means by which wire bonds and bonding pads can be eliminated at critical high speed modes. Through the reduction of interconnection nodes, packaging constraints are alleviated and reliability of the overall system benefits.

This section discusses progress made in one of these areas. In particular, monolithic integration of optical detectors with associated high speed receiver electronics is addressed. The reader is referred to previous publications ([13], [14]) for more details of the discussion to follow.

The I.C. described here was developed to demonstrate the viability of optical interconnects in the clock distribution problem. The system concept is shown in figure 5-1. An optical clock signal, generated from a laser diode, is split into many parts using a fiber optic power splitter. A number of synchronously operated I.C.'s are then clocked optically from the same source. The technique used to achieve planar optic-electronic coupling was called the lateral fiber-I.C. coupler [14].

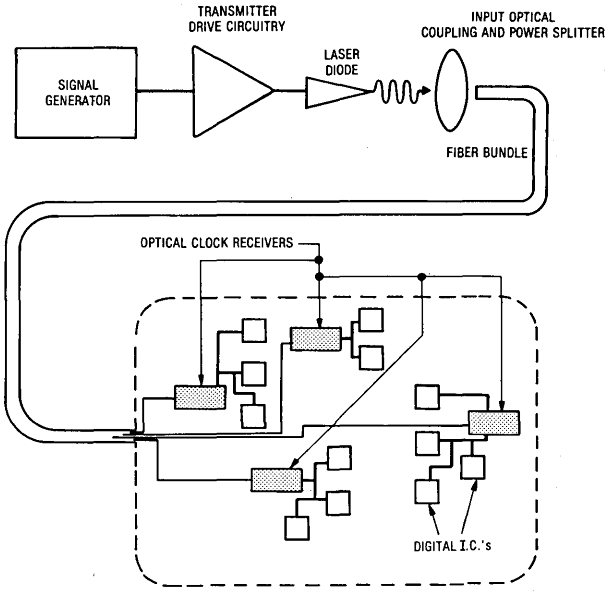


Figure 5-1. Block diagram of an optical clock distribution system that operated at 500 Mb/s and used fibers as the interconnect medium [12], [14].

The concept is shown in figure 5-2. The end surface of a fiber is lapped and polished at approximately 45°. Then a thin layer of aluminum is deposited on the surface. The result is a 45° mirror integrated onto the fiber. The fiber is then coupled laterally to the detector/amplifier I.C. The I.C. is then packaged in a miniature metal flat pack for incorporation onto the p.c. board.

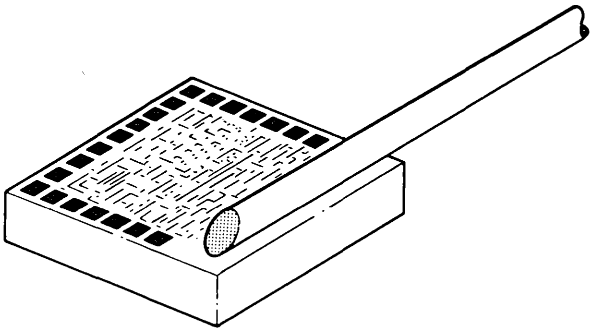


Figure 5-2. Schematic representation of the lateral fiber coupling technique.

Design of the I.C. was predicated on the requirement that an existing ECL process would be used in fabrication. An absolute minimum of modification of this process was to take place in the photodetector fabrication. The mainline process used is a very high density monolithic integrated circuit process designed specifically for high speed ECL circuits operating at subnanosecond speed. In this design, the active region of the bipolar transistors in high frequency amplifier circuits is about $1.1 \mu m$.

Since the deepest diffusion performed in the integrated circuit process is the buried layer, at approximately $3 \mu m$, and since the light penetration depth of silicon at $\lambda = 0.82 \mu m$ is approximately $10 \mu m$, it is clear that some modification or addition to the I.C. process was required to achieve acceptable photodetector performance. The solution involved a compromise between the two competing objectives of optimizing optical response and leaving the I.C. process unaffected. At the beginning of the I.C. process flow, the photodetector area was defined by a mask and a deep phosphorus implant was made into the p-type substrate. The phosphorus was then in-diffused to widen the compensated intrinsic region as much as possible. Implant was made to a depth of approximately $5 \mu m$. The resultant diffusion profile for the integrated photodetector indicated an intrinsic region width (defined here at FWHM) at zero bias of approximately $4.5 \mu m$.

Figure 5-3 shows a block diagram of the detector/amplifier circuit design. Functionally, it is composed of five sections. These are 1) the photodetector, 2) the front end transimpedance amplifier, 3) an intermediate stage amplifier, 4) an ECL limiting amplifier and 5) a D.C. bias control feedback network. Transimpedence gain was 3000 ohms. In the design, the primary goal was to obtain the highest gain-bandwidth product (at 500 Mhz) possible, with receiver sensitivity a secondary issue.

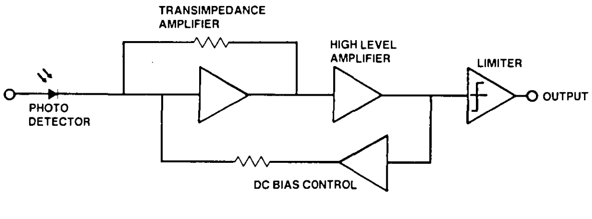


Figure 5-3. Block diagram of the 500 Mhz detector/amplifier circuit design.

5.1 Detector/Amplifier Measured Performance

The detector/amplifier I.C.s were characterized during a test and evaluation phase of the development effort. Parameters measured in these tests included optical sensitivity, frequency response, waveform reproduction, and DC stabilization. The highlights of this test effort are presented here. Data presented here is typical, not best case. Furthermore, data from several different devices is shown in the scope traces and plots. Reference [13] gives a more complete description of the device performance.

For the purposes of evaluation of the photodiode and the amplifier circuits separately, a family of I.C.s were fabricated in which the photodiode contact was separated from the amplifier input. On these I.C.s, bonding pads were placed at the photodiode output and amplifier input. The photodiode responsivity to unmodulated optical signals was measured with a test configuration involving a $0.82 \mu m$ laser diode source coupled to a communications grade $50 \mu m$ core graded index fiber. Cladding modes were eliminated with the addition of coupling gel over a 3 cm length of stripped fiber near the laser-fiber interface. The average dc responsivity for the $80 \mu m$ detector was .5A/W. Excellent linearity prevailed over the usable input power region.

Transient response of the detectors was measured by illuminating them with pulses from a laser diode excited by a HP33004 comb generator. These pulses occurred at a 500 MHz repetition rate and exhibited sharp rise and fall times, approximating a delta-function pulse. When measured using a high speed photodetector, they showed nondeconvolved pulse widths (FWHM) of 40 ps. Figure 5-4 shows a sampling scope trace of the response of our $60 \mu m$ diameter integrated silicon photodetectors to these pulses. Rise times of 60 ps and FWHM of 150 ps are measured, indicating a high frequency response (into 50 ohms) of approximately 4 to 5 GHz. The tail in the response reflects the trade-offs involved in the integrated opto-electronic design.

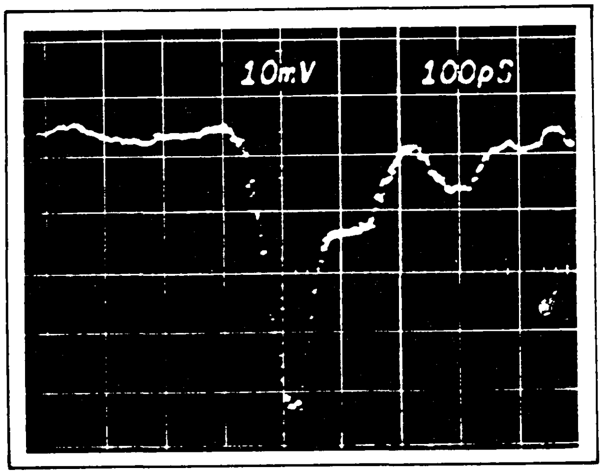


Figure 5-4. Measured impulse response of the integrated n-i-p photodiode (load impedence 50 ohms).

Many devices were tested with the optical input provided by the lateral coupling technique described earlier. Figure 5-5 shows a micro-photograph of the packaged detector/amplifier. Tests revealed that additional optical losses occurring from the use of the lateral coupler compared to direct coupling are typically 1.5 dB.

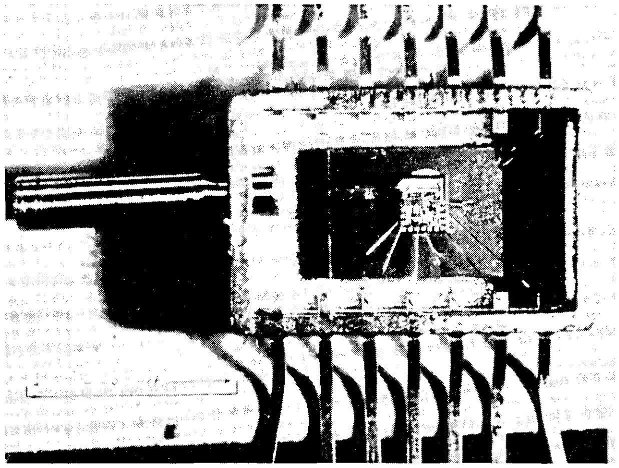


Figure 5-5. Micro-photograph of the packaged detector-amplifier I.C.

Swept frequency response of the packaged devices was measured for a detector/amplifier having a modulated optical signal input through a lateral coupler. The results are presented in figure 5-6. This figure shows the combined frequency response of the photodiode driving the amplification stages.

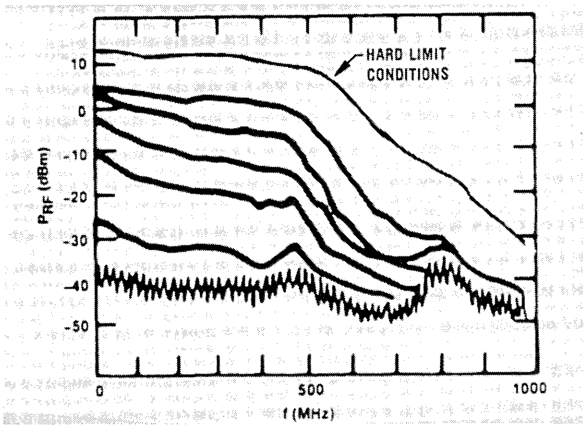


Figure 5-6. Swept frequency response of the packaged detector-amplifier I.C.'s.

The photodiode characteristic response is evident at the low frequency end of the spectrum with peaking evident at the high end (small signal region). The top trace shows the detector/amplifier output response when the input was driven by nearly 1 mW optical power. This is the maximum amount of optical power the I.C. can handle before DC bias problems occur. Also, it is more than 10 times the power required to drive the output sine wave across its full transfer curve. This is referred to in the figure as a "hard limit condition." In the lower traces the average input optical power begins at approximately 100 uW and decreases in 5 dB steps. The packaged devices were also tested for response to binary data sequences. This was done by modulating the laser diode transmitter circuit with a pseudo-random binary sequence generator output. The results are presented in figure 5-7. It is evident that the waveforms are accurately reproduced and that the risetimes are sufficient for transfer of wideband binary information. Measured output pulse jitter was less than 50 ps.

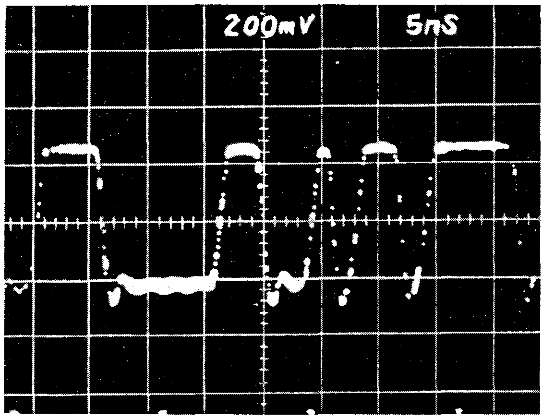


Figure 5-7. Response of the packaged detector/amplifier I.C.'s to a 500 Mb/s NRZ data input.

6. Summary

This paper has addressed the concept of high speed digital interconnects, with an eye on the use of photonics as an alternative to conventional methods. An attempt has been made to point out some of the fundamental engineering limits (as opposed to fundamental physical limits) to high speed interconnects.

The primary theme of the paper is the assertion that as target system data rates are pushed into the multi-Gb/s domain, packaging and interconnections become more and more important. This is because the interconnect media becomes transmission line-like. Under these circumstances wideband transmission forces the designer to consider the device package as part of the electronic circuit. Mechanical and electronic design are merged into one. No longer is one more important, or more simple than the other.

To illustrate this theme, the problem of receiver design for optical interconnects has been addressed. Link budgets for laser diode based and LED based optical interconnect links have been presented. Finally, a specific example of the utility of integrated optoelectronics in optical interconnects has been given.

REFERENCES

[1] Gimlett, J. L., et al, "Transmission experiments at 560 Mb/S and 140 Mb/s using single-mode fiber and 1300 nm LED's", Electr. Lett., *21*, pp. 1198-1200 (1985).

[2] Brackett, C. A., "A view of the emerging photonic network", ISSC '86, to be published.

[3] Jacobs, I., "Atlanta Fiber System Experiment: Overview", B.S.T.J., *57*, pp. 1717-1721 (July-August 1978).

[4] Muska, W. M., T. Li, T. P. Lee, A. G. Dentai, "Material-Dispersion-Limited Operation of High-Bit-Rate Optical-Fiber Data Links Using LED's", Electr. Lett., *13*, pp. 605-607 (1977).

[5] Blank, L. C., L. Bickers, S. D. Walker, "120-Gbit · km lightwave system experiments using 1.478-μm and 1.52-μm distributed feedback lasers", #WB4; OFC '85 Technical Digest.

[6] Personick, S. D., "Fiber optics; technology and applications", Plenum Press, 1985, Ch. 11.

[7] Hackett, W. H., R. H. Saul, P. W. Shumate, "Optical links brighten office opportunities", Bell Laboratories Record, pp. 5-9 (March 1983).

[8] Personick, S. D., "An engineering perspective on the applications of photonic switching technology", Conference record, IEEE Global Telecommunications Conference, Nov. 1984.

[9] Caulfield, H. J., J. A. Neff, W. T. Rhodes, "Optical Computing: the coming revolution in optical processing", Laser Focus, *19* (11), pp. 100-110 (Nov. 1983).

[10] Goodman, J. W., F. J. Leonberger, S. Y. Kung, R. A. Athale, "Optical Interconnects for VLSI Systems", Proc. IEEE, *72* (7), pp. 850-866 (July 1984).

[11] Kurokawa, T., N. Takato, Y. Katayama, "Polymer Optical Circuits for Multimode Optical Fiber Systems", Applied Optics, *19* (18), pp. 3124-3129 (Sept. 1980).

[12] Hartman, D. H., "Digital High Speed Interconnects - a study of the optical alternative", to be published in Optical Engineering, *25* (10), (October 1986).

[13] Hartman, D. H., M. K. Grace, C. R. Ryan, "A Monolithic Silicon Photodetector/Amplifier I.C. for Fiber and Integrated Optics Applications", IEEE Journal of Lightwave Technology, *LT-3* (4), pp. 729-738 (August 1985).

[14] Hartman, D. H., M. K. Grace, F. V. Richard, "An Efficient Lateral Fiber-Optic Electronic Coupling and Packaging Technique Suitable for VHSIC Applications", IEEE Journal of Lightwave Technology, *LT-4* (1), pp. 73-82 (Jan. 1986).

[15] Clymer, B. D., J. W. Goodman, "Optical clock distribution to silicon chips", to be published in Optical Engineering, *25* (10), (October 1986).

[16] Smith, R. G., S. D. Personick, "Receiver Design for optical fiber communication systems", Topics in Applied Physics, *39*, Chapter 4, Springer-Verlag, 1982.

[17] Sata, K., "Intensity Noise of Semiconductor Laser Diodes in Fiber Optic Analog Video Transmission", IEEE Journal of Quantum Electronics, *QE-19* (9), pp. 1380-1391 (Sept. 1983).

[18] Suzuki, A., T. Uji, Y. Inomoto, J. Hayashi, Y. Isoda, H. Nomura, "InGaAsP/InP 1.3 μm wavelength surface-emitting LED's for high-speed short-haul Optical Communications", IEEE J. Lightwave Tech., *LT-3* (6), pp. 1217-1222 (Dec. 1985).

[19] Goldstein, B., M. Ettenberg, N. A. Dinkel, J. K. Butler, "A high-power channelled-substrate-planar AlGaAs Laser", Appl. Phys. Lett., *47* (7), pp. 665-657 (Oct. 1985).

STANDARDS AND ARCHITECTURE FOR TOKEN-RING LOCAL AREA NETWORKS

Jacalyn Winkler
Jane Munn

IBM Corporation
Research Triangle Park, North Carolina

ABSTRACT

Local area network standards have been developed by the Institute of Electrical and Electronics Engineers (IEEE) and are becoming international standards through the International Organization for Standardization (ISO). The token-ring is an approved IEEE standard and an ISO draft international standard (DIS). This paper presents the architecture for token-ring local area networks as described in the IEEE 802.5 Standard and ISO DIS 8802/5. Also included is a discussion of the logical link control sublayer as described in the IEEE 802.2 Standard and ISO DIS 8802/2.

INTRODUCTION

The IEEE Project 802 Committee has developed a family of standards for local area networks (LANs). These standards encompass the data link and physical layers of the Open Systems Interconnection (OSI) Reference Model [1] developed by the International Organization for Standardization (OSI). The data link layer for LANs is divided into two sublayers: medium access control and logical link control. The IEEE 802.5 Standard [2] defines the specifications, formats, and protocols of the physical layer and the medium access control sublayer for the token-ring local area network. The IEEE 802.2 Standard [3] defines the services and procedures of the logical link control sublayer.

The physical layer of a token-ring LAN provides data synchronization, data symbol encoding and decoding, and physical attachment to the ring. The medium access control (MAC) sublayer controls access to the ring, so that individual stations have the opportunity to transmit without interference from other stations. The logical link control (LLC) sublayer provides two distinct types of service that provide for the transfer of information between network layer entities on a LAN. (Network layer entities are entities above the data link layer that interface directly to the LLC sublayer.) One type of service, unacknowledged connectionless, provides a datagram approach to information exchange where information is simply sent and received without correlation to previous or subsequent information and with no form of acknowledgment to ensure delivery. Another type of service, connection-oriented, provides error-free, in-sequence delivery of information without duplication.

A token-ring LAN is comprised of stations connected sequentially by a series of point-to-point links. Each station acts as an active repeater and regenerator of signals on the ring. Due to the physical configuration and protocol operation of the token-ring, problem detection, isolation, and bypass are possible. The token-ring architecture also provides eight levels of priority for access to the shared transmission medium, allowing fair access to the medium within each priority level.

On a token-ring LAN, a station obtains the right to transmit data by capturing a token (a special bit pattern circulating the ring between data transmissions). The transmitting station changes the token to a frame and appends addressing information and data. Each station on the ring examines the addressing information and repeats the frame. A station recognizing the destination address as its own also copies the frame. When the frame returns to the sender, the sender removes the frame from the ring and generates a new token.

One station on the ring - called the active monitor - provides token-monitoring and other protocol monitoring functions to ensure that the ring is operating normally. The other stations - called standby monitors - monitor the health of the active monitor, and are ready to take over if the current active monitor fails. Although only one station on the ring is designated as the active monitor at any one time, the active monitor function is part of the architecture of every station on the ring.

Stations on the token-ring detect and report error conditions allowing for the isolation of errors. The errors detected by a station are divided into two categories: soft and hard errors. Soft errors are those that temporarily degrade the ring performance. Examples of soft errors detected by a station are line hits, possible duplicate addresses, and invalid symbols between starting and ending delimiters. Hard errors prevent the token-ring protocols from functioning properly. Examples of hard errors are a streaming station, and a broken wire.

CH2345-7/86/0000/0479$01.00 © 1986 IEEE

The specifications, formats, and protocols of the physical layer and the medium access control and logical link control sublayers for the token-ring LAN are discussed in more detail in the following sections.

PHYSICAL LAYER

The physical layer of the token-ring LAN synchronizes its clock with the signal received from the ring so that it can interpret received data. The data signalling rates as defined by the IEEE 802.5 Standard [2] are one and four million bits per second. Station attachment is via shielded twisted pair cable. Unshielded twisted pair attachment is under study by the IEEE 802.5 committee.

Differential Manchester Encoding

The token-ring LAN uses the differential Manchester code to convert binary data (bits) into signal elements. These signal elements:

- Contain no direct-current (DC) components, so they avoid plating or other deterioration at attachment contacts
- Allow the receiver to derive clocking pulses from the encoded signal.

For each bit of data, the code consists of two signal elements of opposite polarity. This maintains the DC balance in each bit and guarantees a transition for clocking. The value of each bit (0 or 1) is determined by comparing the first signal element of the bit with the last signal element of the previous bit. If the signal elements have the opposite polarity, the bit has a value of 0; otherwise it has a value of 1. The polarity (positive or negative) of the signal element does not affect the decoding of the 0's and 1's; only the transition, or lack of such transition, affects the decoding.

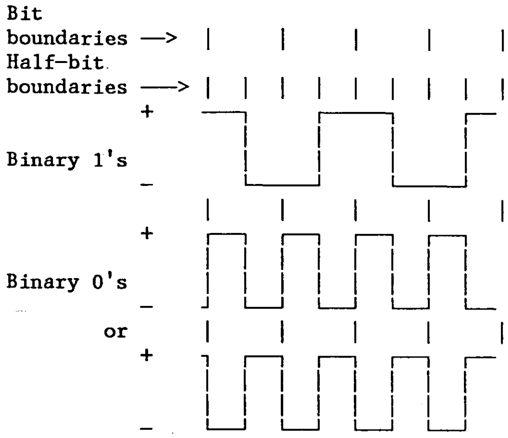


Figure 1. Differential Manchester Code

A code violation is said to occur if there is no transition (from plus to minus or minus to plus) at the bit midpoint. Frames and tokens are delimited by starting and ending delimiters that use four code violations (two positive and two negative to maintain DC balance) to guarantee uniqueness from the data stream.

MEDIUM ACCESS CONTROL SUBLAYER

The medium access control sublayer supports medium-dependent functions and uses the services of the physical layer to provide services to the logical link control sublayer. It allows a station to gain access to the ring for transmission, to detect and report error conditions, and to report configuration changes in the ring. The token-ring frame and token formats used by the MAC sublayer are provided in an appendix. The following sections describe the various functions of the MAC sublayer and the use of MAC management frames to control the operation of the token-ring LAN.

Access Priority

The access priority indicates the priority of a frame or token. The priority is contained in the first three bits of the access control field (see Appendix), allowing for eight levels of priority. Reservations for a priority token are placed in the last three bits (the reservation bits) of the same field. A station can transmit a frame using a token with a priority less than or equal to the priority of the frame. If an appropriate token is not received, the station may request a token at the required priority by placing a reservation in the access control field of a repeated frame or token.

When a station removes one of its transmitted frames from the ring and finds a non-zero value in the reservation bits, it must originate a non-zero priority token. The station determines the priority of the token based on the priority used by the station for the recently transmitted frame, the reservation bits received in the returning frame, and any stored priority (due to a previous transmission of a priority token). The priority token circles the entire ring giving every station the chance to transmit frames of a priority greater than or equal to the priority of the token. This allows for fair access at each priority level.

When the priority token returns to the originator, the station removes the token from the ring and issues a token at a priority determined by the reservation bits and the previous priority.

Active Monitor

The active monitor detects lost tokens, and frames and priority tokens that circle the ring more than once. The following sections describe the mechanisms used by the active monitor to detect and correct these error conditions.

Neighbor-Notification Process: The neighbor-notification process is initiated by the active monitor. It is used to inform stations that an active monitor is present on the ring, and of the address of their nearest active upstream neighbor (NAUN). The NAUN address is used for error isolation.

The neighbor notification process begins when the active monitor transmits an Active Monitor Present MAC frame to all stations on the ring. The first station to receive the Active Monitor Present MAC frame sees the address recognized and frame copied bits in the frame status field (see Appendix) set to zero indicating the frame was sent by its NAUN. It then sets these bits to one, saves the source address of the received frame as its NAUN address, and transmits a Standby Monitor Present MAC frame to all stations on the ring.

The next station immediately downstream copies the Active Monitor Present MAC frame, sees the address recognized and frame copied bits set to one, and disregards the frame. The station then copies its NAUN address from the Standby Monitor Present MAC frame that was transmitted by its upstream neighbor, sets the address recognized and frame copied bits in that frame to one, and transmits its own Standby Monitor Present MAC frame to all stations on the ring.

In this way neighbor-notification proceeds around the ring, with other stations transmitting their Standby Monitor Present MAC frames, until the active monitor copies the last Standby Monitor Present MAC frame, in which the address recognized and frame copied bits are set to zero. Neighbor notification thus enables a station to learn its NAUN address, and to provide its address to its downstream neighbor.

Frame and Token Control: The active monitor uses the monitor bit in the access control field to detect circulating frames or tokens with priority greater than zero. The monitor bit is set to zero in tokens and transmitted frames. When the active monitor repeats a frame or priority token with the monitor bit set to zero, it sets the monitor bit to one. If the bit has already been set to one, the active monitor assumes that the token or frame has already circulated the ring once (the station that originated the token or frame did not remove it). The active monitor purges the ring (see "Purge Process") and originates a new token.

The active monitor ensures that a token is always available on the ring by using a timer with a relatively short time-out that exceeds the time required for the longest possible frame to circle the ring. The active monitor restarts this timer each time it repeats a token or frame. If this timer expires, the active monitor assumes that the token was lost on the ring. The active monitor purges the ring (see "Purge Process") and originates a new token.

Purge Process: The active monitor purges the ring by transmitting a Ring Purge MAC frame to all stations on the ring. Receipt of the returned frame indicates to the active monitor the ring is viable. Other stations receiving a Purge MAC frame reset to repeat mode and cancel or restart appropriate timers.

Standby Monitor

The function of a standby monitor is to detect failures in the active monitor and other disruptions of the ring protocol. Each standby monitor maintains two timers to perform this function.

One timer, which has a longer duration than the active monitor's token monitoring timer, ensures that the active monitor's token monitoring function is operative. This timer is restarted when a token is repeated. If this timer expires, the standby monitor initiates token-claiming (see "Token-Claiming Process") to elect a new active monitor.

A second timer used to detect the absence or failure of the active monitor is restarted each time the standby monitor receives an Active Monitor Present MAC frame. If this timer expires, the standby monitor assumes that an active monitor is not present on the ring, or that the active monitor has malfunctioned. The standby monitor then initiates token-claiming.

Token-Claiming Process

Token-claiming is the process used to elect an active monitor. A standby monitor detecting an error, or the absence of the active monitor, initiates the token-claiming process by broadcasting Claim Token MAC frames to all stations on the ring. A station copying a Claim Token MAC frame compares its individual address to the source address of the received frame. If the source address is less than the station's individual address, the station transmits its own Claim Token MAC frames; otherwise, the station repeats received frames. If a station transmitting Claim Token MAC frames receives a frame with a source address higher than its individual address, the station stops transmitting its own Claim Token MAC frames and begins repeating received frames.

Receipt of its own Claim Token MAC frames indicates that the station has been elected as the active monitor. This station then purges the ring and originates a new token.

Beaconing Process

A station detecting a hard error broadcasts Beacon MAC frames to all stations on the ring to inform them of the detected error. The Beacon MAC frame includes the address of the nearest active upstream neighbor for use in error isolation and recovery. Upon receipt of Beacon MAC frames, the NAUN of the Beaconer removes from the ring and tests the continuity of the wire connecting it to the ring. If an error is detected, the station remains off the ring and the ring recovers. If the NAUN does not find an error when testing its lobe, the station reattaches to the ring.

After a period of time, if the ring has not recovered, the beaconing station removes and tests the continuity of the wire connecting it to the ring. Again, if an error is detected, the station remains bypassed allowing the ring to recover. If no error is detected, the station reattaches to the ring. If the error persists, manual intervention is necessary to recover the ring.

Attaching to the Ring

In order to communicate on the ring, a station must successfully go through a sequence of events during the ring insertion process. Before physically inserting in the ring, a station performs self tests to ensure the continuity of the wiring connecting it to the ring and that the station itself is functioning properly. After successfully completing these tests, the station physically attaches to the ring. The station then checks to be sure there is an active monitor on the ring by detecting a frame sent by the active monitor. If there is not an active monitor on the ring, the station initiates token-claiming.

After either detecting or establishing an active monitor, the station checks for the presence of another station on the ring with the same address by issuing a Duplicate Address Test MAC frame to its individual address. If the frame returns without being copied by another station, the station's address is assumed to be unique. The station then waits to participate in neighbor notification to learn its NAUN's address, and to provide its address to its downstream neighbor.

Lastly, the station requests parameter values from a management server on the ring. This request contains registration information for the station attaching to the network. If a management server is present on the ring, the station completes the insertion process after receiving a response from the server. If a server is not active on the ring, the station completes the insertion process and uses the default values for the ring parameters.

Soft Error Detecting and Reporting

Soft errors are intermittent faults that temporarily disrupt normal operation of the token-ring; they are normally tolerated by error recovery procedures. Soft errors are indicated by architectural inconsistencies (such as cyclic redundancy checks or timeouts) in received or repeated frames, and by a ring station's inability to process received frames. If soft errors result in degraded ring performance, the ring can be reconfigured to bypass the faulty node.

Each ring station maintains a set of counters to measure the frequency of occurrence of the most critical soft errors. At specified intervals, these errors are reported to a management server using a Report Soft Error MAC frame. The Report Soft Error MAC frame reports the number of errors detected since the last report was made. This report identifies the transmitting station's NAUN for use in error isolation and recovery.

Configuration Control

A management server may collect configuration information from stations on the ring. Configuration reports are generated when the insertion or removal of a station is detected during the neighbor notification process. A configuration report is also generated when a new active monitor is elected.

A management server may change the configuration by forcing a station off the ring. A management server may also query a station for various status information unique to that station.

LOGICAL LINK CONTROL

The IEEE 802.2 Standard [3] for logical link control defines peer-to-peer protocol procedures for the transfer of information between any pair of network layer entities on a local area network. Logical link control is not unique to the token-ring; it is intended to be used in conjunction with all the medium access control and physical layer standards defined in IEEE 802.

The formats and protocols of logical link control were derived from HDLC standards [4]. The format of the LLC protocol data unit (LPDU), the unit of information delivered from or to the MAC sublayer, differs from HDLC in that it does not include a frame check sequence or frame delimiters, and it contains two address fields. The functions of frame delimiting and frame check sequence generation and verification are provided by the medium access control sublayer; therefore, they are not necessary in the logical link control sublayer.

The two address fields contain the destination and source link service access point addresses. Link service access points (LSAPs) are the logical points at which network layer entities acquire the services of the data link layer (see Figure 2). To support the coexistence of multiple network layer entities, the LLC sublayer supports up to 127 service access points. The destination link service access point address in the LPDU is used by the receiving LLC to identify the network layer entity for which information is received. Within a network layer entity, the source link service access point address in conjunction with the MAC source address is used to determine the logical "connection" for which information is received. A network layer entity may have multiple logical "connections" active simultaneously.

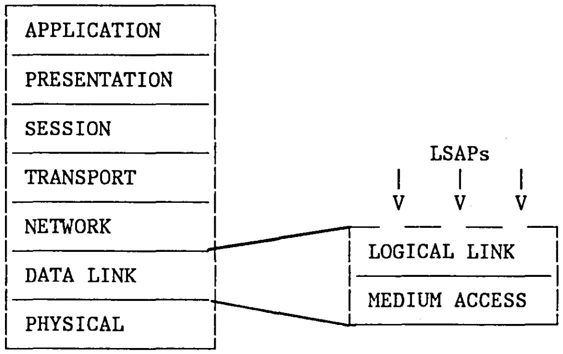


Figure 2. Link Service Access Points in the OSI Reference Model

The logical link control protocols closely resemble those defined in HDLC [4]. While basically a connection-oriented protocol, HDLC does define an optional facility for unacknowledged information transfer. LLC differs from HDLC in that it requires the support of unacknowledged information transfer and defines as optional a connection-oriented protocol.

Logical Link Control Services

Logical link control provides two types of service to the network layer: unacknowledged connectionless (type 1) and connection-oriented (type 2). The requirements of the higher layers determine which service type is applicable in a given situation. To provide a common set of minimal services, all stations are required to provide unacknowledged connectionless service.

Unacknowledged connectionless service is a datagram approach to information exchange where information is simply sent and received without correlation to previous or subsequent information and with no form of acknowledgment to ensure delivery. The LLC sublayer requires the network layer to explicitly specify the source and destination addresses in all requests for unacknowledged connectionless data transfer service. When information is received, the LLC sublayer routes the information as well as the addresses to the appropriate network layer entity as specified by the destination link service access point address. No state information is maintained by the LLC sublayer for unacknowledged connectionless data transfers. Unacknowledged connectionless service is useful when higher layers provide adequate error recovery and sequencing, or when it is not essential to guarantee the delivery of information.

Connection-oriented service provides a reliable communications path between any two network layer entities (identified by LSAP addresses) on a local area network. This service provides flow control, sequencing, and error recovery procedures. A "data link connection" is established, at the request of a network layer entity, between two LSAPs prior to any exchange of information. The connection may be reset to its initial state or terminated at any time by either network layer entity involved in the connection. Connection-oriented service is applicable when higher layer protocols require error free service.

The performance tradeoffs between using connection-oriented service and performing error recovery in the LLC sublayer, and using unacknowledged connectionless service and performing error recovery at a higher layer, is a subject of continual controversy. For an analysis, see "Performance Analysis of Error Control Alternatives in Local Area Networks" by Syed and Field [5].

Logical Link Control Protocols

The protocols defined for both unacknowledged connectionless and connection-oriented data transfer were derived from HDLC protocols. As in HDLC, the concept of "command" and "response" protocol data units is used. In the following sections, descriptions of the protocols for each type of service are provided.

Unacknowledged Connectionless Service: Unacknowledged connectionless service provides data transfer between two network layer or management entities with minimum protocol complexity. Figure 3 lists the LLC commands and responses that provide unacknowledged connectionless service.

LPDU Name	Abbr.	Command	Response
Unnumbered Information	UI	X	
Exchange Identification	XID	X	X
Test	TEST	X	X

Figure 3. LLC Commands and Responses for Connectionless Service

The UI command LPDU is used to transport data from one network layer entity to another. UI command LPDUs may be sent or received at any time. Received UI LPDUs are neither acknowledged nor verified for correct sequence. The receiver of an UI command LPDU simply passes the contents of the information field to the network layer entity designated by the destination link service access point address. UI command LPDUs can be lost if an exception (for example, a transmission error or a receiver-busy condition) occurs during transmission. Higher layer protocols are responsible for detection of lost or duplicate LPDUs, and for any required retransmissions.

The XID command LPDU is used to convey identification and characteristics of the sending station, and to cause the receiving station to respond with the XID response LPDU. The IEEE 802.2 Standard [3] defines the format for the information field of an XID LPDU. The field contains information related to the types of service provided by the station or by a specific LSAP, and for connection-oriented service, it specifies the receive window size. All stations must support the information field defined by IEEE 802.2 and must be capable of returning an XID response when an XID command is received. The logical link control standard does not exclude the exchange of user-defined XID information formats, such as, those defined by Systems Network Architecture (SNA).

The TEST LPDU is used to perform a basic test of the transmission path between two stations. The information field is supplied by the user or a management facility. All stations must have the ability to return a TEST response when a TEST command is received.

Although XID and TEST are considered part of the unacknowledged connectionless service, they provide facilities that are more likely to be invoked by management entities rather than by network layer entities. An implementation may choose to provide error recovery procedures for XID and TEST within the logical link control sublayer as long as the flow sequences remain independent from that of the connection-oriented service.

Connection-Oriented Service: The protocol procedures for connection-oriented service are similar to the HDLC asynchronous balanced mode of operation (see [4] for details). Figure 4 lists the LLC commands and responses that are used in connection-oriented service. Unlike HDLC, which allows the choice of 8 or 128 as the modulus for sequence numbering, logical link control defines the modulus to be 128.

Before information can be exchanged between two network layer entities, a data link connection must be established between two LSAPs. Connection establishment is requested by a network layer entity. Upon receiving such a request, the LLC sublayer sends a SABME command LPDU to the destination specified in the request. The receiver of the command accepts the connection by returning an UA response LPDU, or rejects the connection by returning a DM response LPDU. If the connection request is rejected, the initiating network layer entity is notified and must retry its connection request at a later time if it still desires the connection. If the connection request is accepted, the data link connection is successfully established and the two network layer entities may exchange information. The LLC sublayer provides in-sequence information transfer without loss or duplication of data.

LPDU Name	Abbr.	Command	Response
Information	I	X	X
Receive Ready	RR	X	X
Receive Not Ready	RNR	X	X
Reject	REJ	X	X
Set Asynchronous Balanced Mode Extended	SABME	X	
Disconnect	DISC	X	
Unnumbered Acknowledgment	UA		X
Disconnected Mode	DM		X
Frame Reject	FRMR		X

Figure 4. LLC Commands and Responses for Connection-Oriented Service

Information is transferred from one network layer entity to another using I command or response LPDUs. (The concept of a command or a response in a peer-to-peer environment is not meaningful in many cases; a station may use either in many situations.) Each I LPDU contains a send sequence number that allows the receiver to verify proper ordering. The sender of an I LPDU runs an acknowledgment timer and maintains a copy of the entire LPDU until an acknowledgment is received. To maximize throughput, a number of I LPDUs may be outstanding, i.e. sent but not yet acknowledged, in each direction of a data link connection before the transmitter must stop and wait for an acknowledgment. This transmit window has a maximum value of 127.

If a received I LPDU contains the next sequence number expected, the information is delivered to the network layer entity and an acknowledgment is returned to the sender. Acknowledgments are indicated by the value of the receive sequence number in I, RR, RNR, and REJ LPDUs. All I LPDUs numbered up through the receive sequence number minus one are acknowledged; hence, a single LPDU may acknowledge multiple I LPDUs. The IEEE 802.2 standard does not require an acknowledgment to an I LPDU be returned at the earliest opportunity unless the I LPDU was a command with the P bit set to "1" (see [3] for details). If an I LPDU is not available for sending, a station may delay the sending of an acknowledgment for some period of time bounded by the probability of the expiration of the remote acknowledgment timer, for either an I LPDU to become available for transmission, or to accumulate additional I LPDUs to be acknowledged in a single RR LPDU, subject to window size constraints. One method that may be used to control acknowledgment sending involves the use of a timer and a counter such that an explicit acknowledgment will be sent whenever the timer expires or the count limit is reached (see [6] for details).

If an I LPDU is received out of sequence, the information field is discarded and a REJ LPDU containing a receive sequence number equal to the expected sequence number is returned. The receiver of the REJ LPDU will retransmit the I LPDU containing the expected sequence number and any other unacknowledged I LPDUs that had been previously sent following the one that was rejected.

Lost frames that cannot be discovered by sequence number validation are detected when the acknowledgment timer expires. Upon expiration, a "checkpointing" operation is initiated. "Checkpointing" consists of sending a RR command LPDU with the P bit set to "1." The receiver of the command is obligated to respond with its expected sequence number. By interrogating this number, the initiator of the "checkpointing" operation can determine if retransmission of any I LPDUs is necessary.

If a station becomes temporarily unable to receive information frames due to resource constraints such as buffering limitations, it can notify the remote LLC by sending a RNR LPDU. The receiver of a RNR LPDU should stop sending I LPDUs until it receives further notification. A RR LPDU may be sent to notify the remote LLC that it may resume sending I LPDUs.

Either network layer entity involved in the connection may request that the connection be terminated. A DISC LPDU is sent to notify the remote LLC of the disconnection. The receiver of the DISC LPDU responds with an UA LPDU. The disconnection is not negotiable; the receiver of a DISC LPDU may not reject the disconnection request.

If a protocol error is detected, the station detecting the error sends a FRMR LPDU containing a summary of the detected error to the originating station. When a station receives a FRMR LPDU, it may attempt to reset the connection or treat the connection as being disconnected.

CONCLUSIONS

The IEEE 802.5 and 802.2 Standards have progressed to Draft International Standards within ISO. The architecture for token-ring LANs as defined in the standards provides superior fault isolation, problem determination, and recovery capabilities. The protocols allow excellent operational characteristics at high speeds and under heavy loads.

The architecture for token-ring LANs can easily be extended to include the interconnection of rings via bridges. This extension would provide improved connectivity, performance, and availability. Source routing provides a method of routing frames through a bridged network, where the source of a frame explicitly identifies the route that a frame is to follow. The appendix contains a proposed addition to the token-ring frame format to include the routing information field used in source routing.

APPENDIX: TOKEN-RING FRAME AND TOKEN FORMATS

The basic transmission unit on the token-ring LAN is the frame. Frames are composed of a number of fields of one or more bytes, as shown below:

```
┌──┬───────┬───────────┬─────────┬──────//────┬───//─────┬─────────┬──┬──┐
|S |Control|Destination|Source   |Routing     |Informa-  |Frame    |E |F |
|D |Fields |Address    |Address  |Information|  tion    |Check    |D |S |
|  |       |           |         |            |          |Sequence |  |  |
└──┴───────┴───────────┴─────────┴──────//────┴───//─────┴─────────┴──┴──┘
|<────────── Physical Header ──────────────>|            |<── Physical ->|
                                                               Trailer
```

SD = starting delimiter
ED = ending delimiter
FS = frame status field

Figure 5. Frame Format

Starting Delimiter: The starting delimiter is a single byte with the following format:

```
┌─────────────────┐
| J K 0 J K 0 0 0 |
└─────────────────┘
```

J = Code Violation
K = Code Violation

Figure 6. Starting Delimiter

All valid frames and tokens start with this byte.

The J and K indicate code violations, which identify the byte as a delimiter. Use of code violations avoids using special bit patterns of normal codes for delimiters; therefore, bit insertion, as used in HDLC, is not required.

Control Fields: The control fields are each one byte in length and are used to determine access to the ring. The control fields have the following format:

```
┌──────────────────────┬──────────────────┐
│ P P P  T  M  R R R   │ F F  Z Z Z Z Z Z │
└──────────────────────┴──────────────────┘
 Access Control        │ Frame Control

 P = Priority Bit         F = Frame Type Bit
 T = Token Bit            Z = Control Bit
 M = Monitor Bit
 R = Reservation Bit
```

Figure 7. Control Fields

The priority bits indicate the priority of a token or frame. There are eight priority levels, from zero to seven.

The token bit is used to distinguish between frames and tokens. A token is the signal used to give a station permission to transmit. The figure below shows the format of the token:

Starting Delimiter	Access Control	Ending Delimiter

Figure 8. Token Format

The monitor bit is used by the active monitor to prevent a token whose priority is greater than zero, or any frame, from continuously circling the ring.

The reservation bits allow stations with high access priorities to request, in repeated frames or tokens, the priority of the next token. There are eight reservation levels, from zero to seven.

The frame type bits indicate the type of information contained in the frame as either MAC management, or user data.

The control bits are used in the MAC management frames to determine the method for copying a frame. If the control bits equal zero, the station copies the frame using normal receive buffers and does not copy the frame if no buffers are available. If the control bits are greater than or equal to one and the normal receive buffers are full, the station copies the frame into buffers reserved for MAC protocol management to insure appropriate action is taken by the station.

Destination Address: The destination address identifies the ring stations that are to copy the frame. Destination addresses are always six bytes in length with the following format:

```
              bytes
| 1 | 2 | 3 | 4 | 5 | 6 |
┌─┬─────┬─┬──────────────┐
│ │     │ │              │
└─┴─────┴─┴──────────────┘
 ││         └─ Functional Address bit
 │└─ Universal / Local Administration bit
 └── Individual / Group Address bit
```

Figure 9. Destination Address

The individual/group address bit indicates whether the frame is addressed to a specific station or a group of stations on the ring. The universal/local administration bit indicates whether the address is administered by IEEE thereby, guaranteed to be unique across all types of LANs, or administered such that the responsibility for unique addresses is placed on the local administration. All addresses must be unique in a given network. The functional address bit indicates whether a locally administered group address is a bit mapped address (functional address) identifying a management function, or a flat six byte group address.

These indicators are an integral part of each station's address, and considered during address-recognition.

Source Address: The source address identifies the ring station that originated the frame. Source address is six bytes in length with the following format:

```
              bytes
| 1 | 2 | 3 | 4 | 5 | 6 |
┌─┬──────────────────────┐
│ │                      │
└─┴──────────────────────┘
 │└─ Universal / Local Administration bit
 └── Routing Information Indicator
```

Figure 10. Source Address

A source address is always an individual address, so the individual/group address bit distinction defined in the destination address field is not needed. Instead, this bit may be used as a routing information indicator specifying whether or not a routing information field is present in the frame.

As in the destination addresses, bit 1 of byte 0 indicates whether the address is administered by IEEE or a local authority.

Routing Information Field: Following the source address is a routing information field present only in frames leaving the source ring. The field, when present, consists of a two-byte routing control field and up to fourteen two-byte segment numbers.

bytes

2	4	6		m
Routing Control	Segment Number	Segment Number	//	Segment Number

Figure 11. Routing Information Field

Routing Control Field: The routing control field is two octets in length with the following format:

	B	r	LTH	D	r
bits	1	2	5	1	7

B = Broadcast
r = reserved
LTH = Length
D = Direction

Figure 12. Routing Control Field

The Broadcast bit indicates if the frame is destined for all segments in a network. This bit does not imply that the frame is destined for all stations on all rings. The Length bits indicate the length of the routing information field including the routing control field. The Direction bit indicates if the frame is traveling from the originating station to the target station or visa versa.

Information Field: The variable-length information field contains an integral number of bytes of MAC management or user information.

While there is no explicit restriction on the size of the data field, the values of various MAC timers limit the practical maximum frame size. In an environment where there are several rings, connected by bridges, the bridges may also be the limiting factor in determining the practical maximum frame size.

Frame Check Sequence: The frame check sequence is a four-byte cyclic redundancy check (CRC) covering the frame control field, the addresses, the routing information field, the information field, and the frame check sequence itself.

Ending Delimiter: The ending delimiter is a single byte with the following format:

J K 1 J K 1	0	E

J = Code Violation
K = Code Violation
E = Error Detected Bit

Figure 13. Ending Delimiter

As in the case with the starting delimiter, code violations (J, K) are used to denote the ending delimiter. The ending delimiter is different from the starting delimiter in that bits 2 and 5 are set to 1. Using different patterns for the starting and ending delimiters simplifies the detection of each.

The error-detected bit is used to indicate that an error was detected in the frame by a station on the ring. It also indicates to a station detecting an error in a frame or token, whether the error has been logged by another station on the ring.

Frame Status: The frame status field is a single byte with the following format:

A	C	r	r	A	C	r	r

A = Address Recognized Bits
C = Frame Copied Bits
r = reserved

Figure 14. Frame Status Field

The address recognized and frame copied bits indicate the following conditions to the sending station: no station recognized the destination address and the frame was not copied, a station recognized the destination address and copied the frame, or a station recognized the destination address but did not copy the frame.

REFERENCES

1. International Organization for Standardization, "Information Processing Systems, Open Systems Interconnection, Basic Reference Model," ISO 7498-1984, 1984.

2. IEEE Computer Society, Token Ring Access Method and Physical Layer Specifications, ANSI/IEEE Standard 802.5 - 1985 (ISO/DP 8802/5), IEEE, 1985.

3. IEEE Computer Society, Logical Link Control, ANSI/IEEE Standard 802.2 - 1985 (ISO/DIS 8802/2), IEEE, 1984.

4. International Organization for Standardization, "Information processing systems, Data communications, High-level data link control procedures, Consolidation of classes of procedures," ISO 7809-1984, 1984.

5. A. Syed, and J. A. Field, "Performance Analysis of Error Control Alternatives in Local Area Networks," Proceedings of Infocom 1986, Miami, April, 1986, pp.503-509.

6. IBM Corporation, IBM Token-Ring Network Architecture Reference, Publication Number 6165877, 1986.

THE IBM TOKEN-RING NETWORK - A FUNCTIONAL PERSPECTIVE

MICHAEL WILLETT

IBM CORPORATION
NETWORK SYSTEMS DESIGN
RESEARCH TRIANGLE PARK, NORTH CAROLINA 27709

ABSTRACT

The acceptance of the personal computer as an intelligent workstation, coupled with the increasing demand for shared access to common files and for electronic office communication, has created a need for a reliable, high-speed, local communication network. This paper is an overview of an implementation of the IEEE 802.5 token-ring network recently released by IBM. The paper covers the general architecture of the network and focuses on how this implementation achieves high reliability and the flexibility of an open system, which are essential to meet present and growing data communication needs.

1.1 INTRODUCTION

A LOCAL AREA NETWORK (or LAN) is a geographically confined communication system, generally inhabiting a shared transmission medium. The main ingredients of a LAN are:

- the transmission medium
- the access protocols, which govern the way that access to the shared medium is achieved
- the adapter, for controlling the connection of a device to the medium
- the communication protocols, once access is achieved.

A gathering of people in a room has all the ingredients of a local area network. The transmission medium is the air filling the room (a bus topology!). The access protocols may vary from meeting to meeting. In a social gathering and within the confines of social amenities, we contend with others for the right to speak. At the next lull in the discussion, we begin to speak. If someone else begins at the same time, we both politely back off for a brief moment and then re-initiate our conversation. Eventually, someone wins the contention and gains control of the transmission medium. In a lively gathering, a high percentage of the time could be spent on the access protocols. In a more structured business meeting, access may proceed in an orderly fashion around the table. In this case and even with everyone having something to contribute, very little time is spent on the access protocols. If each person's time to speak is limited, then each will have access to the medium again within a guaranteed amount of time. The adapter is our set of vocal cords and the communication protocols are the ordinary rules of language, specialized to the occasion.

In the late sixties, the ALOHA Network at the University of Hawaii was developed to apply packet radio techniques for communication between a central computer and terminals scattered about the Hawaiian Islands. This network used a contention-based access method for gaining control of the central processor. Contention methodology was enhanced in the Ethernet communication system, developed by Xerox in the early seventies. Ethernet utilizes Carrier Sense Multiple Access with Collision Detect (CSMA/CD) to control access of multiple processors to a shared bus. A station initiates transmission only if the bus is quiet. Because this access method requires backoff and retransmission whenever multiple transmissions collide on the bus, the throughput is adversely affected when high utilization of the communication bandwidth is attempted. Also, since the transmission must travel the length of the bus and return before potential collisions are detected by the transmitting station, the access method does not provide a proportional increase in throughput as the transmission rate is increased, unless the bus is shortened.

The token-ring access scheme (10) offers an alternative method for sharing a common transmission medium. Transmission is point-to-point serially around a ring, with the transmitter of one station attached to the receiver of another station. A special bit sequence called a TOKEN is passed from each station to the next downstream station. If a station wishes to transmit, then that station will capture the token (ie, not forward it) and instead transmit a frame, addressed to another individual station or a group of stations in the network. All stations monitor the destination address of frames being forwarded. The intended recipient copies the frame while forwarding it. The transmitting station has the responsibility of removing a frame when it has completed the 'round' trip. As soon as this 'stripping' process begins, the station can release the token to the next downstream station.

The ring topology would seem to be vulnerable to a break in the transmission path, but the sequential ordering of stations is the basis for effective on-ring error isolation and recovery (see below). In addition, the baseband, digital technology involved is mature and can be implemented inexpensively. The point-to-point nature of the transmission lends itself to the use of optical fiber cable. The controlled nature of the access method remains stable at very high speeds, so that token-ring access control was selected as the basis for the 100 megabit/sec Fiber Distributed Data Interface (FDDI), a local area network standard being drafted by the ANSI X3T9.5 Committee (see (13)).

Early work on IBM's token-ring architecture was carried out at the IBM/Zurich Research Laboratory. Members of the LAN Architecture and System Design Departments of IBM at the Research Triangle Park in North Carolina evolved the details of the architecture and made significant contributions to the LAN standards activities.

With the formal introduction of the IBM Token-Ring Network in 1985, software and hardware developers applied considerable energy to producing software written to the published specifications for interfaces and hardware that could co-exist on the transmission subsystem. Token-ring technology can support the local data communication needs of an establishment of almost any size. The widespread adoption of the personal computer as an intelligent workstation in the business environment, coupled with the increasing need for peer-to-peer and host-interactive connectivity, emphasize the requirement for an establishment-wide local area network.

In the following sections, IBM's implementation of a token-ring LAN is highlighted, with attention focused on the openness, reliability/availability, and standardization of the network, beginning with the transmission subsystem. Successively higher-level layers of the implementation are also discussed.

1.2 IBM CABLING SYSTEM AND COMPONENTS

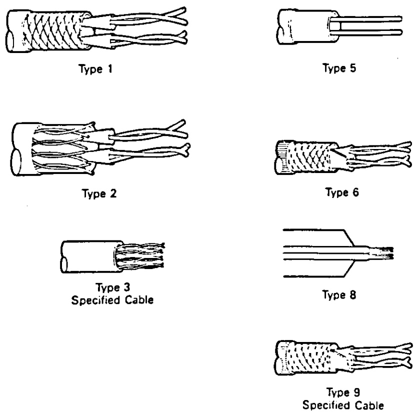
A major part of any communication system is the transmission subsystem. Standardization of communication protocols will do little to promote the acceptance of LAN technology without an equal effort to standardize the medium. Look in the ceiling or under the floors of any building that has hosted a computer and/or telephone communication system for a few years, and you will find a maze of wire, much of which has been abandoned. In 1984, IBM introduced components and specifications for a cabling system which are designed to accommodate the local communication needs of an establishment for the foreseeable future.

The IBM Cabling System has the following attributes:

- 'STANDARD' - The component and cabling specifications have been made available. Independent vendors/suppliers provide the components and the installation expertise, in addition to IBM.

- OPEN - Universal connectors and faceplates allow for easy access to the medium of a variety of communication products. IBM has announced its intention to make future devices compatible with the cabling system. Many communication products compatible with the cabling system are already available from multiple sources. A variety of communication products are accommodated by using the appropriate connector cable (or balun) in the office, without rewiring to the wiring closets.

- STRUCTURED - Instead of being a maze of point-to-point wires, the cabling is installed with a structured topology. Cable drops in offices are routed through raceways to wiring



- Type 1 — Data communication
- Type 2 — Data and voice communication (Both Type 1 & 2 can be plenum/non-plenum)
- Type 3 — data and voice communication (telephone twisted-pair)
- Type 5 — Fiber optic (Between wiring closets/network backbone)
- Type 6 — Data communication (Patch cables)
- Type 8 — Undercarpet cable (Data only)
- Type 9 — data communication

FIGURE 1: CABLE TYPES

closets, which are strategically located throughout a building. The section of cable from the office to the wiring closet is called a LOBE.

The cabling terminates in wiring racks or telephone punch-down blocks. The cabling can then be connected together at a patch panel to provide the endpoint connectivity desired. Rearrangement of the physical connectivity is easily accomplished by changing the patch-panel connections. This star-shaped wiring topology allows for a 'virtual rewiring' of the building without any of the traditional expense. Wiring closets are themselves connected together to provide structured, systematic connectivity throughout the establishment.

A new outlet in an office or a new type of cable can be difficult and expensive to provide. Ideally, a building is prewired with the cabling system during construction, with sufficient office drops and overall accessibility provided to handle future expansion.

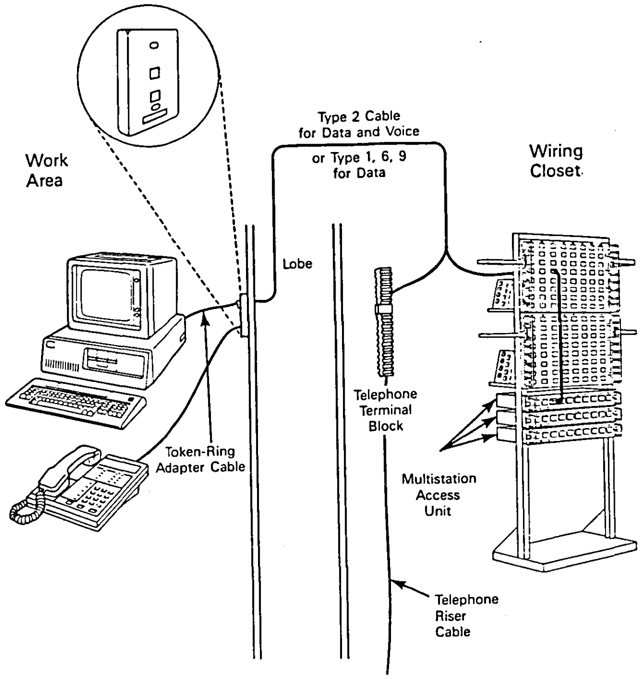


FIGURE 2 : OFFICE TO WIRING CLOSET LOBE

1.2.1 NETWORK CABLING COMPONENTS

The Token-Ring Network can be superimposed on the cabling system by adding the appropriate components in the wiring closets and in the offices.

A Multistation Access Unit allows up to 8 lobes to be combined serially in the wiring closet or (if convenient) locally within an office. The access unit is not powered. Instead, a 'phantom' current from the attaching device activates a switch in the access unit to insert the lobe into the active ring of the network or to bypass an inactive or inoperative device.

Copper repeaters can be used to extend the length of the main ring path between wiring closets beyond the specified limits. The repeater reshapes and retransmits the signal, just as does the adapter.

Optical fiber repeaters can be used in pairs to extend the length of the main ring path up to 2 kilometers by using optical fiber cable. Fiber is being increasingly used in point-to-point communication because of its high transmission rate characteristics and its immunity to electromagnetic interference.

1.3 MAC/LLC IMPLEMENTATION

The Data Link Control (DLC) layer of the communication architecture is specified in the standards that have been developed by the Institute for Electrical and Electronic Engineers (IEEE) Project 802 Committee.

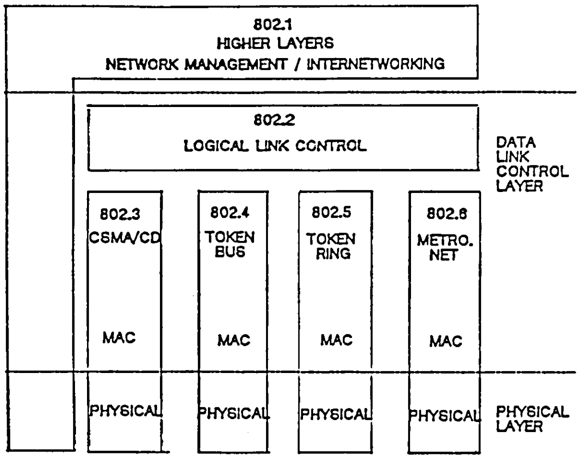


FIGURE 3: IEEE PROJECT 802 STRUCTURE

The 802.5: Medium Access Control (MAC) sublayer standard specifies the token-ring access method. The 802.2: Logical Link Control (LLC) sublayer standard specifies the remainder of the Data Link Control, independent of the particular access method. All 802 LANs have the same LLC.

The MAC sublayer handles the token protocols and the ring recovery procedures. Each ring station monitors the traffic on the network to determine if errors or interruption in the MAC layer protocols have occurred. Under normal operation, one station on the ring has special functions activated, with this active monitor capability on standby in all other stations.

The active monitor has an incremental 24-bit latency buffer to insure that a (24-bit) token will fit on the ring. The active monitor can also sense lost tokens, circulating frames, or circulating priority tokens. Additional active monitor duties include token generation in case the network has no token and activation of a 'neighbor notification' process whereby each station learns the address of its nearest active upstream neighbor (NAUN). The active monitor also provides a master timing clock for the rest of the ring, which is imbedded in the data stream through the use of Manchester encoding. The other stations are in standby mode, checking the health of the active monitor. Any station can assume the role of the active monitor, if needed. The process for 'electing' an active monitor is called 'claim token'.

The ring topology with its unidirectional flow provides a robust environment for error isolation and recovery. Since each station knows its upstream neighbor, the first station (in order around the ring) to note an error will then know the fault domain (two stations and the connecting cable) for that error. This information is reported to network management and used in automatic error recovery techniques. Error types include soft errors such as bit errors in a frame or hard errors such as loss of signal.

In the case of hard errors, a 'beaconing' protocol is used to notify all stations that error recovery is taking place and to share the fault domain information. For example, the station immediately downstream from a wire fault will repeatedly transmit a 'beacon' frame containing its own address and that of its NAUN. The other stations on the ring repeat the beacon frame. When (and if) the NAUN station sees its own address in a beacon frame, it removes itself from the ring and performs a self-test. If this test indicates normal operation, the station will re-insert into the ring. If normal ring operation has not returned in a brief period of time, the 'downstream' member of the fault domain (the beaconing station) will also go through removal and self-test. Automatic recovery and the other MAC-layer functions are handled by the processor on the adapter.

Any station that wishes to join the network goes through a five-step process to insert itself into the ring:

1. Lobe check, whereby test frames are transmitted to the bypass circuitry in the multistation access unit and wrapped back to the station for validation,
2. Active monitor present check,
3. duplicate address check,
4. participation in 'neighbor notification' protocol,
5. report to network management.

If the insertion process fails, notification is given to the attaching device.

The services of the Logical Link Control sublayer are accessed through logical, addressable points within the sublayer called Service Access Points (SAPs). Multiple logical link endpoints can exist in a single station, using SAP addressing. The services of this sublayer are of two types. TYPE 1: CONNECTIONLESS service provides for transmission (and receipt) of information without implementing link error recovery or frame acknowledgment. This type of service is also called datagram service. TYPE 2: CONNECTION-ORIENTED service requires that SAPs be extended by connection components, which serve as connection endpoints for DLC links. A DLC link consists of the two MAC/SAP address pairs, one pair in each communicating station. The connection component is responsible for processing the events which affect the particular DLC link. Frames transmitted on such links are numbered in sequence and acknowledged upon receipt. Any errors detected cause link recovery protocols to be executed. For further details, see (14).

1.4 ADAPTER DESIGN

Devices requiring attachment to the token-ring LAN must interface to an adapter that implements MAC-level protocols, regardless of the higher-layer protocols. For example, these functions are incorporated into the Front End, Protocol Handler, and Message Processor of the adapter developed to attach IBM personal computers to the ring.

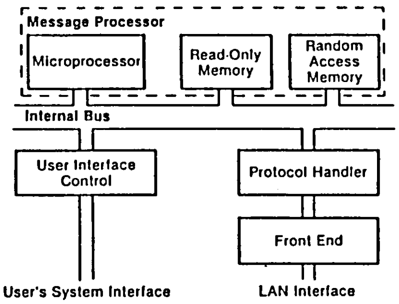


Figure4. Functional Partitioning of the Adapter

1.4.1 FRONT END

The receive/transmit functions of the adapter are implemented in the Front End. Analog circuits reshape and redrive the received signal. The 'clock' is also extracted from the received signal by means of a phase-locked loop. The Front End also supports lobe and adapter testing in addition to error monitoring of the physical attachment.

1.4.2 PROTOCOL HANDLER

Two data paths are incorporated in the Protocol Handler, which is under the control of microcode stored in the memory of the Message Handler. The serial path is used to forward received frames on the ring. The parallel path converts the serial data stream into two-byte segments, checking address fields to determine if a frame is destined for that adapter. Copied frames are stored in the random-access memory of the adapter.

1.4.3 MESSAGE PROCESSOR

Included in this segment of the adapter are a microprocessor, read-only memory for storing the operating program of the adapter, and random-access memory used to buffer messages. The Message Processor determines if a received frame requires action at the MAC layer or must be passed to the LLC layer.

1.4.4 USER INTERFACE CONTROL

The Interface Controller handles the transfer of data between the random-access memories of the adapter and the attaching device. This shared memory approach, using a memory map common to both the adapter and personal computer memories, enhances the performance of the adapter by eliminating the overhead of parameter passing, since memory address and data segment length definitions are identical between the two processors. A bidirectional interrupt capability and programmable timing facility are also provided. A permanent, adapter address (if provided) is contained in programmable, read-only memory. The standard adapter contains 8K of random-access memory. A memory-enhanced adapter with 16K of memory is also available for high-performance devices such as bridges and servers in which more data buffers are needed.

The user interface can be customized to the internal structure of the attaching device. For example, the adapter was modified to mate to the IBM 3725 Communication Controller, which then attaches high-end host machines to the network.

A joint development effort between IBM and Texas Instruments led to the TMS380 chip set (12), which implements the MAC layer protocols necessary for attachment to a token-ring LAN. This chip set is functionally equivalent to the MAC layer portion of the IBM adapters and is available for attachment of equipment from other manufacturers. Of course, LLC and higher layer protocols and a customized, device interface need to be provided as well.

1.5 BRIDGES AND SOURCE ROUTING

A bridge is a routing mechanism between two or more segments (eg, rings) of a local area network. The bridging function takes place within the Data Link Control layer, either at the MAC or LLC sublayers. The bridge is transparent to higher layer protocols. Such a relay function, existing at low levels in the communication protocols, will usually provide high throughput between the connecting segments. Contrast this with a gateway, which connects networks at levels higher than the data-link control level and therefore is involved to some extent with protocol conversion and address mapping.

The current bridge between token-ring segments uses two adapters installed in an IBM Personal Computer or Industrial Computer, and performs a relay function at the MAC sublayer. The processor of the personal computer executes a bridge program to achieve the bridging function. It is important to realize that each ring maintains its own token and MAC-level protocols. Tokens do not cross bridges. The automatic recovery techniques (if needed) are isolated to the single affected ring. Therefore the aggregate data capacity of the bridge-connected network is increased. The bridge selectively copies frames on one ring that are marked for routing and determines if the frame should be transmitted on the connecting ring.

A route through a network consists of a sequence of ring/bridge numbers, starting at the source ring and ending at the destination ring. No specific routing protocols have been recommended by the IEEE 802 Committees. The routing scheme utilized by the IBM Token-Ring Network is called SOURCE ROUTING, which means that the source station must provide the route and insert it in the frame as extended addressing. Each bridge first notes the existence of a routing information field in a frame and then scans that field to determine if its own ring/bridge/ring combination appears. If so, then the frame is forwarded. Routes can either be predefined, which is static, difficult to manage, and subject to aging, or determined dynamically when needed. In dynamic source routing, a source station wishing to determine a route to a particular adapter address, which may reside on another ring in the network, first broadcasts a special route-selection frame, which is forwarded by all bridges (subject to certain restrictions). Each bridge appends its own bridge/ring numbers to the route, thereby 'growing' the route as the frame traverses the network. The receiving adapter returns each such frame received, using the stored route in the frame in reverse order. The first such response received by the initiating station contains the 'shortest' route effectively. This route is used in further dialogue with the destination station.

Source routing does not require either the bridges or network management to have a current, global view of the network configuration. This property enhances the dynamics and robustness of network connectivity. Also, these protocols support multiple routes between source and destination rings.

1.6 PROGRAMMING INTERFACES

In order to promote the development of network software and the attachment of IBM and non-IBM equipment, documented interfaces and protocols exist at all communication levels. The Texas Instruments chip set and chip sets available from other vendors provide a MAC-level interface that can be incorporated into a custom adapter for a variety of communicating devices.

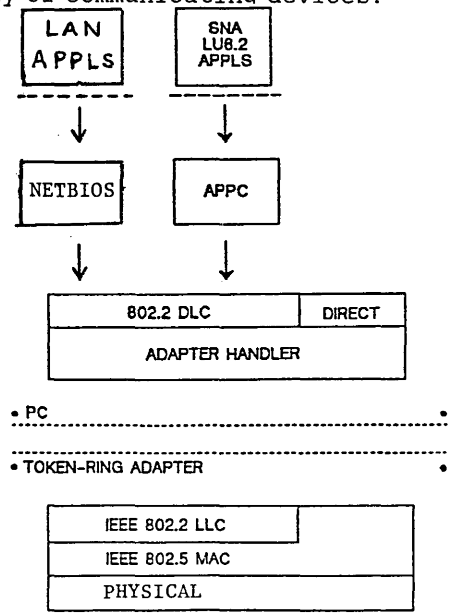


FIGURE 5: ADAPTER/SOFTWARE INTERFACES

The IBM personal computer adapters (standard and Adapter II) also provide a MAC-level interface for those users who want to write software to that interface. The DLC interface provides access to the LLC functions on the adapter. The Adapter Handler code supplied with the adapters provides a more user-friendly interface for the adapter interfaces just described.

The Network Basic Input/Output System (NETBIOS) interface is rapidly becoming a defacto-standard, programming interface for local area networks. The NETBIOS interface is provided by a program that runs in an IBM PC containing an adapter. Other networks provide the same interface, regardless of the underlying access and transmission protocols. This wide use of NETBIOS promotes the development of extensive network software that is portable across a variety of networks. Both the NETBIOS interface and the underlying protocols used to implement NETBIOS on the Token-Ring Network are documented in IBM publications.

The Advanced Program-to-Program Communication for the IBM Personal Computer (APPC/PC) program product provides a Systems Network Architecture (SNA) LU6.2 interface. This interface has also been developed for a number of other communication products, which means that software written to that interface should be directly compatible with the Token-Ring Network. Both the interface and protocols of APPC have been documented.

Other programs written to the NETBIOS interface, such as the PC LAN Program and the Asynchronous Communications Server, provide server interfaces to the network and access to the public switched network.

In contrast to the early development of communication systems in which interfaces and protocols were proprietary, the communication systems of today, integrating hardware and software from many different vendors, require an unprecedented level of openness and standardization. The IBM Token-Ring Network exemplifies these requirements.

1.7 NETWORK MANAGEMENT

The MAC-level protocols contain a number of procedures for automatic error monitoring and recovery. In the event of a network hard error, the fault domain is generated by the detecting station and transmitted on the network. A report is also generated by the first station detecting a soft error. The Ring Diagnostic provided with the adapters, when executed in an IBM Personal Computer, will retrieve these reports from the network and display them for the user. This information is a useful ingredient in the problem determination procedures provided. An extension of the Ring Diagnostic functions is also available, called the Network Manager program, which executes further ring monitoring and station control functions.

A network of bridged rings has a three-tiered management hierarchy.

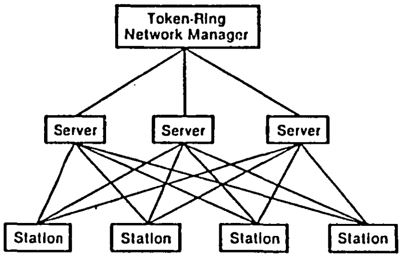


Figure 6 Management Hierarchy for Token-Ring Local Area Networks

Each separate ring has its own token protocols and ring monitoring functions, distributed across the adapters on the ring. MAC-level bridges add a second level and provide a convenient station on each attaching ring in which to house other network management functions. For example, the monitoring and reporting functions of the Ring Diagnostic are integrated into the bridge code. These server functions, residing in a bridge station or some other specially designated station on each ring, could be designed to report selective information to a network manager function residing elsewhere in the network or to perform specialized management functions.

Examples of potential management servers include:

- RING ERROR MONITOR - collects and analyzes hard- and soft-error reports from the local stations. Selected results can be reported to the network manager. These server functions are incorporated into the Ring Diagnostic and the Network Manager program product.
- CONFIGURATION MONITOR - collects configuration change reports from stations.
- RING PARAMETER SERVER - initializes and maintains a consistent set of values for operational parameters in ring stations.
- RING LAYER MANAGER - requests the status of stations, changes the values of operational parameters, and forces configuration changes.

The network manager entity on the Token-Ring Network could in turn report selected information to a higher management function whose domain is the whole establishment communication complex.

1.8 CONCLUSIONS

The IBM Token-Ring Network is an open network, utilizing the latest technology, and adhering to internationally-accepted standards for local area networks. The reliability of the components and the automatic error isolation and recovery designed into the low-level protocols provide for high availability of the network. The star-wired, ring topology promotes a systematic layout of the transmission medium in a building. The token-access protocols offer orderly, predictable access to the network, even under high utilization. Bridges and repeaters provide for network expansion, both in physical size and number of stations.

Together with the IBM Cabling System, the network offers a strategic solution to an establishment's data communication and connectivity needs for the future.

1.9 REFERENCES

1. W. Bux, F. Closs, K. Kummerle, H. Keller, H. R. Mueller, "Architecture and Design of a Reliable Token-Ring Network", IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, Vol SAC-1, No 5 (November 1983).
2. W. Bux, F. Closs, P. A. Janson, K. Kummerle, H. R. Mueller, E. H. Rothauser, "A Local-Area Communication Network Based Upon a Reliable Token-Ring System", PROCEEDINGS of the IFIP TC-6 International In-Depth Symposium on Local Computer Networks, Florence, Italy (April 1982), pp.69-82.
3. W. Bux, "Local-Area Subnetworks: A Performance Comparison", IEEE TRANSACTIONS ON COMMUNICATIONS, Vol 29 (1981), pp.1465-1473.
4. R. C. Dixon, "Ring Network Topology for Local Data Communications, PROCEEDINGS of COMPCON Fall 1982, Washington, DC, September, 1982, pp.591-605.
5. R. C. Dixon, N. C. Strole, J. D. Markov, "A Token-Ring Network for Local Data Communications", IBM SYSTEMS JOURNAL, Vol 22, No 1/2 (June, 1983), pp. 47-62.
6. R. M. Metcalfe, D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Communication Networks", COMMUNICATIONS OF THE ACM, Vol 19, No 7 (1976), pp. 395-404.
7. J. H. Saltzer, K. T. Pogran, "A Star-Shaped Ring Network with High Maintainability", PROCEEDINGS of the Local Area Communications Network Symposium, Boston, May 1979, pp.179-190.
8. J. H. Saltzer, D. D. Clark, K. T. Pogran, "Why A Ring?", PROCEEDINGS of the Seventh Data Communications Symposium, 1981, pp.211-217.
9. N. C. Strole, "A Local Communication Network Based on Interconnected Token-Access Rings: A Tutorial", IBM JOURNAL OF RESEARCH AND DEVELOPMENT, Vol 27, 1983.
10. IEEE Computer Society, "Token Ring Access Method and Physical Layer Specifications", ANSI/IEEE Standard 802.5 - 1985 (ISO/DP 8802/5), IEEE, 1985.

11. IEEE Computer Society, "Logical Link Control", ANSI/IEEE Standard 802.2 - 1985 (ISO/DIS 8802/2), IEEE, 1984.

12. J. Carlo, C. Hamner, "Implementing the IEEE 802.5 Token-Ring Standard", in these PROCEEDINGS.

13. S. Joshi, "The Fiber Distributed Data Interface: A Bright Future Ahead", in these PROCEEDINGS.

14. J. Munn, J. Winkler, "Standards and Architecture for Token-Ring Local Area Networks", in these PROCEEDINGS.

IMPLEMENTING THE IEEE 802.5 TOKEN-RING STANDARD

M.C. Hamner and J.T. Carlo

Texas Instruments Incorporated
Houston, Texas 77001

ABSTRACT

This paper will describe the methodology for implementing the 802.5 Token Ring standard with the TMS380 VLSI chipset. Critical decisions were made for the Physical and Medium Access Control (MAC) specifications to determine which functions were designed into hardware, allowed to vary using external components or controlled by software. The separation of hardware and software functions is guided by the placement of token-related processes in hardware and MAC-layer frame processing in software. Particular attention is paid to thorough testing and verification of the hardware and software to guarantee ring integrity, isolate possible failures and assure complete 802.5 compatibility.

SECTION 1 INTRODUCTION

The TMS380 token ring chipset(1) is illustrated in Figure 1.

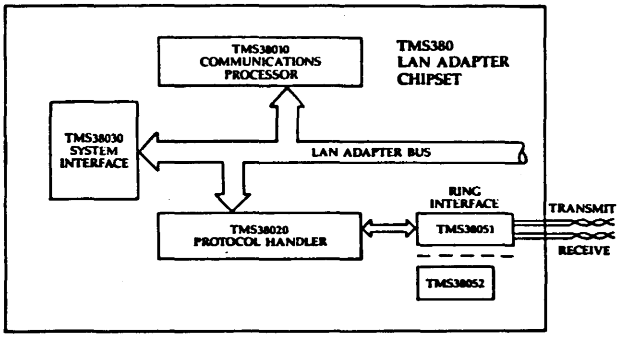


Figure 1
TMS380 Implementation of IEEE 802.5

The chipset consists of three VLSI integrated circuits and two bipolar ring interface devices. Sixteen Kbytes of ROM code, containing both MAC frame processing, diagnostics, and a high level user interface is provided in the Protocol Handler(PH). The Communication Processor(CP) contains 2.75K Bytes of RAM for data buffering and software tables. Both of these memory areas are expandable using external ROM, EPROM or RAM. The two ring interface circuits provide the data conditioning and recovered-clock function for attaching to the twisted pair medium of the token ring. The PH performs the real-time token ring protocols, such as token capture and priority control. The CP executes code which contains the Medium Access Control(2) and management protocol processes of the IEEE 802.5 and test software which ensures integrity of the station and cable attaching the station to the ring. The code also provides a command interface into the adapter for higher layer software. This interface frees the user from all MAC-level processing and isolates MAC processes from the user, protecting the integrity of the ring. The System Interface Chip (SIF) is a Direct Memory Access (DMA) controller for moving data between the host system memory to and from adapter memory.

The remaining sections of this paper describe the partitioning of the IEEE 802.5 functions in the TMS380 chipset. Section 2 describes the ring interface implementation which interfaces to the Physical Layer cabling system. Section 3 describes the Medium Access Control processes implemented in hardware and software. Finally, Section 4 describes those options which are provided to allow adapter station self-test procedures, bridges, and other products to be implemented.

CH2345-7/86/0000/0498$01.00 © 1986 IEEE

SECTION 2 PHYSICAL LAYER

2.1)HARDWARE PARAMETERS

The physical layer controls the electrical connection to the medium, signal conditioning and clock regeneration. The physical layer functions are in hardware. However, different cabling considerations and network optimization are accomplished by allowing some function parameters to be controlled by variable "glue" components. Figure 2 illustrates the block diagram of the TMS380 Ring Interface.

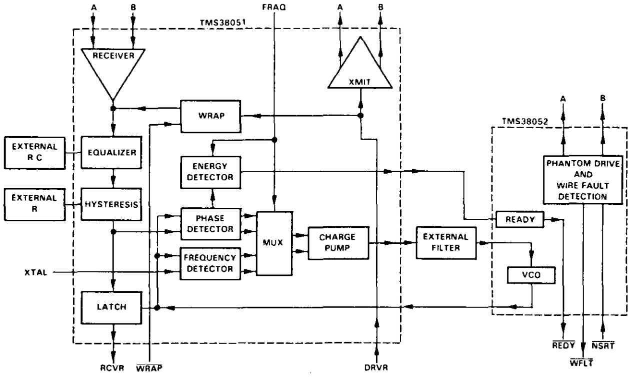


Figure 2
Ring Interface Functional Diagram

To maintain maximum flexibility during design, tradeoffs between hardware circuits on-chip and parameter setting by external glue components were considered. These parameters were varied during program development to allow optimization of system error budgets and verification of analytical models.

Referring to Figure 2, the ring interface functions are split into two separate chips to minimize electrical coupling between the Voltage Controlled Oscillator(VCO) and other clocking signals. The transmitter, receiver, and other blocks which contain data transitions are separated from the VCO to minimize clock jitter caused by crosstalk between the VCO and data.

An externally controlled adaptive EQUALIZER compensates for cable distortion and attenuation. Equalization is effective at low amplitude signals and transparent at high amplitude signals, thus matching the effects of increased frequency distortion with increased cable attenuation. This equalization can be externally modified to match the attenuation characteristics of the transmission medium by changing the external R/C circuit. This is particularly useful when changing from wire to fiber media.

A hysteresis resistor is externally connected to the HYSTERESIS block shown in Figure 2 to allow the ring interface circuit to set the noise threshold from 10mv to 30mv. IEEE 802.5 defines the minimum received signal must be no less than 50mv. Proper setting of the hysteresis resistor prevents noise from being interpreted as data. This external setting will depend on particular crosstalk characteristics of the cable and can be optimized for wire or fiber operation. Typically, a value of 10mv noise rejection threshold is recommended.

In the token ring, a single station, the monitor, provides the master clock which all other stations on the ring regenerate using a PLL. The PLL filter circuit and gain are also controlled by external components. The recommended bandwidth and damping factor of 100kHz and 25.7 respectively were determined for the IBM Token-Ring Network(3). These values determine the jitter build-up, and were set experimentally and proved analytically in order to take into account the entire system error rate budget.

2.1)RELIABILITY INDICATORS

A major feature of the TMS380 is the use of specific hardware fault indicators to monitor network availability at the physical layer. These indicators serve to isolate problems between the cabling wiring coupler and other stations on the ring. The major indicators are: wire fault, which verifies cable continuity to the trunk coupling unit; energy detection, which senses the loss of incoming signal; and frequency error which detects a difference between the regenerated Phase-Locked Loop (PLL) clock and the station's Crystal Clock.

Because cabling problems are one of the major sources of network problems, the TMS380 chipset incorporates a wire fault detection circuit to measure DC continuity of the cable to the Trunk Coupling Unit(TCU). The phantom drive current, as illustrated in Figure 3, provides for the relay switching and physical insertion into the ring.

In addition, each wire in the twisted pair, as shown by the paths (A) and (B) in Figure 3, carries a separate DC current. If the resistivity seen at the outputs PHOUTA or PHOUTB is less than 100 ohms(shorted cable) or if the resistivity is greater than 9900 ohms(open circuit), then a wire fault bit is set and indicated to the user.

An energy detection circuit is provided to monitor transitions on the received data

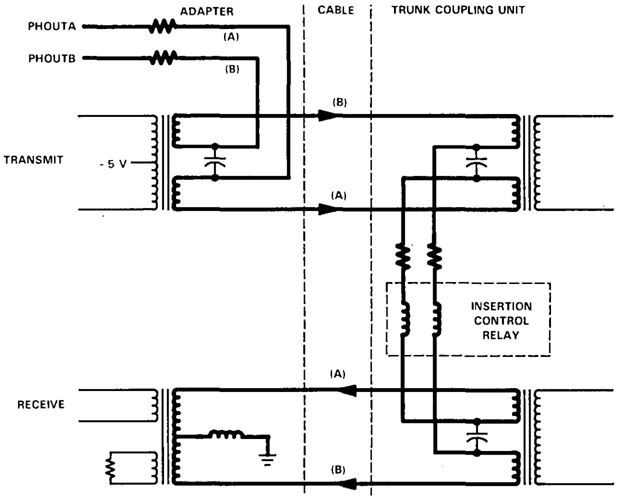


Figure 3
Phantom Drive Signal Path

signal. Under normal circumstances, the REDY- signal shown in Figure 2 is asserted and the PH deasserts FRAQ so that the VCO frequency is set by the received data transitions. If transitions are absent, then the PLL will drift off frequency, causing the PH to assert FRAQ, and the VCO frequency is synchronized to the station's own crystal oscillator clock(XTAL). Thus, in the event of loss of input signal transitions, the PLL is restored to the correct frequency range by rapid hardware intervention. In addition, a bit is set in the PH to inform the station software that a "signal loss" has occurred.

These hardware fault detectors at the Physical Layer provide indication of wiring integrity. A final key feature of the ring interface circuit is the wrap mode, which allows the capability to run diagnostics completely within a single station. In this mode the adapter is a single ring station with the full protocol set. This allows self-test diagnostics to be executed prior to insertion into the ring and can be combined with the external wrap mode (lobe test) to provide an excellent method for localizing problems at the station level.

SECTION 3 MAC LAYER

The implementation of the Medium Access Control (MAC) layer is separated between those processes which must be implemented in hardware, due to ease or speed requirements, and those which could be implemented in software. This separation is illustrated in Figure 4.

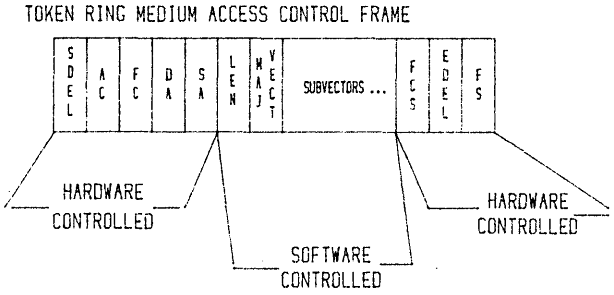


Figure 4
MAC Process Separation

While the hardware controls access to the medium, the software controls the contents of frames. Hardware is used to implement bit-level functions in order to achieve a minimum two bit delay for the repeat path through the PH. Token processing, priority control, address recognition and Frame Check Sequence(FCS) generation and checking are implemented in hardware. MAC frame processing and the MAC processes which require extensive decision tree evaluation are implemented in software. Throughout the design process, attention was paid to implementing with flexibility. Thus, hardware features and hardware operating modes can be selected by software.

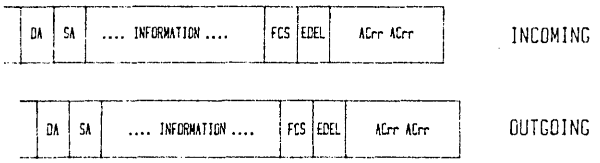
3.1) HARDWARE PROCESSES

Those processes requiring real-time bit manipulation are implemented in hardware in the PH chip. These include processes which capture and modify tokens, modify frame contents, and monitor data on the ring.

The first of these is address recognition. The address recognition feature was designed to be fast enough to recognize an address on a frame entering the station, and correctly set the Address Recognized Indicators (A) in that same frame. The PH must meet the shortest possible timing window. In the case of the A-bit setting, the shortest possible time is calculated by summing the bytes between the end of the Destination Address and the first A-bit in the Frame Status field as shown in Figure 5.

This time is equivalent to a minimum of 98 bits for a single byte information field, which is equivalent to 24.5 microseconds for a ring speed of 4 Mbps.

In addition, the PH is required to be capable of copying back-to-back frames addressed to it. The PH is capable of setting the A/C bits, recognizing a Starting Delimiter(SDEL) immediately following the FS field, and entering the address match process. At 4Mbps this is accomplished with an inter-frame spacing of eight bits. For the flexibility, the inter-frame gap size is determined by a software controlled register in the PH.



RECOGNIZE ADDRESS TIME EQUALS:

```
48 BITS - SA
 8 BITS - INFORMATION
32 BITS - FCS
 8 BITS - EDEL
 2 BITS - PH DELAY
-----------------------
98 BITS  ==>> 24.5 MICROSECONDS
```

Figure 5
Address Recognition Bit-Setting Time

For transmission, the hardware of the PH is responsible for token recognition and capture, and the digital transmission to the ring interface. The hardware synchronizes to the SDEL of a token, and then monitors the Access Control(AC) field to determine if the incoming data is a frame or a token. If it is a token, then the PH sets the token bit in the AC, inserts the user's data following the AC, and attaches the Ending Delimiter(EDEL) and Frame Status(FS) fields at the end of transmission. The PH must convert parallel data from the Adapter Bus into a serial bit stream for transmission. This process is performed entirely in hardware for speed reasons. It is, however, the responsibility of adapter software to initiate the transmission.

The transmit process of the IEEE 802.5 standard is also responsible for proper handling of priority on the ring(4). In the PH, the priority handling functions are implemented in the "fairness" process. If the priority level is increased by a station, that station must ensure that it eventually returns to the level from which it increased priority. To ensure that the "stacking" station receives a fair chance of transmitting at a given priority level, the hardware which performs the fairness process is separated from the transmit process hardware. The PH's fairness process logic is logically downstream from the transmit process hardware in the same node. If active, the fairness hardware modifies the token that was released by the transmit process to create a token of proper priority.

The IEEE 802.5 standard states that the active monitor station on the ring must periodically check for the presence of valid tokens or frames on the ring. The active monitor station uses this information to determine when a Ring Purge is necessary. To do this, a combination of hardware and software detection mechanisms are used. The PH recognizes a valid SDEL-AC sequence as "Any Token", and sets a bit readable by software. At the expiration of a 10ms timer, software checks that the bit has been set, and clears the bit. Software in the monitor station uses this feature to detect the absence of tokens, and initiate Ring Purge.

Stand-by monitors must monitor the ring for "good" tokens, every 2.6 seconds. A "good" token is defined as a priority zero token or a token of priority greater than zero followed by a frame of priority greater than zero. This process is performed in a manner similar to the "any" token check. The hardware sets a bit whenever it detects a good token, and the software checks for the bit and resets it at the expiration of a 2.6 second timer.

To implement protocol processing timers, two timers were placed in hardware. All protocol timers are based on these timers. The Return to Repeat timer indicates a lost frame to the transmit process in the PH. This timer guarantees ring integrity by automatically returning the PH to repeat mode after a frame has exceeded a certain length, independent of other hardware. The General Purpose Timer is the hardware basis for all other protocol timers. This timer is located in the Communications Processor and its duration is controlled by software through a register. A period of 10 milliseconds was chosen. This increment is small enough to handle all the timers specified by the IEEE 802.5 standard.

3.2) SOFTWARE PROCESSES

The remainder of the Medium Access Control (MAC) processes of the token ring are implemented in the TMS380 adapter software. These include neighbor notification, monitor contention, monitor functions, and MAC management support. These processes were implemented in software to keep them flexible and provide the ability to change parameters for different speed rings.

When the PH completes reception of a frame, adapter software is notified. If the received frame is a MAC frame (FC=0x), then control is passed to an adapter software task known as the "Ring Task".

The Ring Task parses the frame to first

determine if it is a ring station class MAC frame or if it is destined for the user system. If it is a ring station class frame, further parsing determines which MAC major vector has been received, and what parameters accompany the command. Depending on the value of the major vector and whether this station is the active monitor, appropriate software routines are initiated.

When the software needs to transmit a MAC frame, it queues that frame to the PH, just like any other data frame. MAC frame transmission may be caused by a timer expiration or may be in response to a received frame. To ensure that vital MAC frames are transmitted in a timely manner, the TMS380 software reserves one internal buffer for the transmission of certain MAC frames. Those MAC frames which are transmitted from this buffer are:

Report Error
Report Monitor Error
Report Neighbor Notification Incomplete
Report New Monitor
Active Monitor Present
Duplicate Address Test
Report SUA Change
Standby Monitor Present
Request Initialization
MAC frames which use a token are transmitted at priority 3, and are always queued ahead of user frames that are pending transmission.

SECTION 4 <u>FEATURES FOR FLEXIBILITY</u>

Though many of the Medium Access Control processes were built into hardware, the ability to override these was deemed necessary for both flexibility and for testing. In this light, various options were built into the TMS38020 Protocol Handler to allow control of hardware functions by software.

One option is that the FCS may be generated by PH hardware or transmitted directly from buffer RAM. This allows adapter software to retransmit an FCS which it received from the ring, instead of regenerating it. This feature could be used in a bridge adapter. The Transmit FCS mode is also very useful for testing the adapter. The PH's FCS detection mechanism can be tested by sending an invalid FCS with frame data. The Transmit FCS mode is selected by writing a bit in a control register so that the mode of operation can change dynamically from frame to frame.

Another option of the PH transmit process is termed "Baud Data Mode". In this mode, the PH transmitter interprets data in the adapter buffer RAM in the format shown in Figure 6.

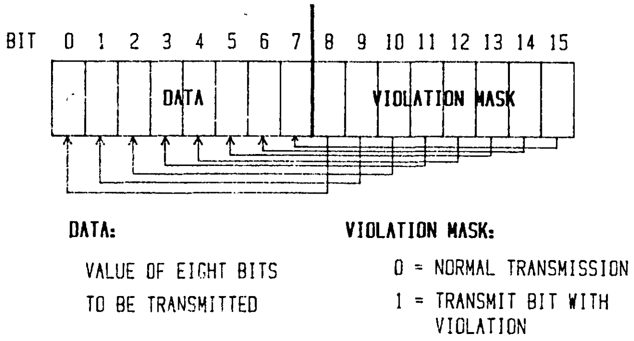

BIT 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
DATA
VIOLATION MASK
DATA:
VALUE OF EIGHT BITS TO BE TRANSMITTED
VIOLATION MASK:
0 = NORMAL TRANSMISSION
1 = TRANSMIT BIT WITH VIOLATION

Figure 6
Baud Data Mode Format

This format allows the user to indicate, by bit, if the bit is to be sent as a valid Manchester encoded value or a violation pattern. In this way, software can simulate the starting and ending delimiters of a frame. Also, for testing the PH error recovery mechanisms, code can induce code violations in a transmitted frame.

To support transmit operations that do not wait for a free token, the transmit process of the PH will operate in the Transmit Immediate mode. This mode, set by software, causes the PH to transmit as soon as it is told the location of frame data in adapter memory. Monitor Contention, Beacon Process, and Ring Purge use this mode.

For debug, or for software priority control, the PH contains a Fire Token register. When this register is written to, the PH will release a token onto the ring, using the contents of this register as the AC field. This allows adapter software to release tokens at any given time onto the ring. This register is used in the diagnostic code to test adapter response to various types of tokens. It is also used in the monitor station to release a token following the Ring Purge process.

The Transmit Idle mode of the transmit process causes the PH to transmit a continuous stream of zero bits without regard to what is being received. This mode can be used for various testing modes. Transmit idles state is entered upon the detection of a BURST5 error in order to prevent downstream adapters from also detecting a BURST5 error, thus isolating the fault.

In order to maintain proper operation on the ring in case of data buffer congestion, the PH receive process provides a mode of operation known as "copy express buffer only". In this mode, the PH will only copy frames addressed to itself, if the value of the Physical Control Field (PCF) bits in the Frame Control field is greater than zero. MAC frames which are essential to ring operation are defined to have PCF bits greater than zero. The adapter reserves an internal buffer, known as the "express buffer", for use whenever it has no other buffers remaining. The IEEE 802.5 standard states that if the adapter is unable to copy a MAC frame destined for the station, it should take action based on the value of the PCF bits. In order to support this requirement , whenever the PH recognizes the address on a MAC frame with PCF bits greater than one, it issues an "attention" interrupt. With this interrupt, the value of the PCF bits is stored in a register, readable by adapter software.

The PH receive process also offers the option of copying all traffic on the ring, regardless of the destination address. The Address Recognized and Frame Copied bits on frames copied in this mode are not affected, unless the frame is addressed to the station. This feature offers the user the ability to observe all traffic on the ring. This can be useful for protocol analysis and debug, or for performance analyzers on the ring.

The PH provides the ability, with its "No-Strip" mode, to transmit frames onto the ring and immediately return to repeat mode. This capability can be used any time the adapter wants to place frames on the ring that it does not remove. For example, the TMS380 Bring-up diagnostics use this mode to test the operation of the PH fairness process.

SECTION 5 CONCLUSION

This paper has described the implementation decisions that were made in the design of the TMS380 token ring adapter. We have described the division between Physical and Medium Access Control functions. The TMS380 Ring Interface devices control most of the physical layer protocols, including insertion into the ring, transmission of valid signal, and reception of signal from the ring. The PH chip performs the remainder of the physical layer protocols, including jitter compensation and monitor clocking. The MAC layer is implemented in both the PH and in software executed by the CP. The PH handles token manipulation while the software controls the interpretation of received MAC frames and the transmission of new MAC frames.

Throughout the design process, careful attention was paid to providing flexibility in those features which were designed into hardware. In this manner, the ability to use different media types, or to fine tune MAC protocols, was maintained. All of these features for flexibility are used by today's chipset in self-test, and can be used in the laboratory environment for system test.

REFERENCES

1)"TMS380 User's Guide", Texas Instruments, September 1985. This book describes the TMS380 chipset and its application to an IBM-compatible Token-Ring Network.

2)"IEEE 802.5 Token Ring Access Method", March 1985, IEEE Standards Board. This is the IEEE specification for the token ring. Signal characteristics are described on page 80.

3)"IBM Token-Ring Network Architecture Reference", February 1986. This document describes the detailed Medium Access Control functions that are used to maintain and control the attaching ring stations.

4)Szczepanek, A., Shore, B., "A VLSI Implementation of a Protocol Controller for Use with IEEE 802.5 Compatible Networks", ISSCC 1984. This paper provides additional details of the design of the TMS380 Protocol Handler.

THE FIBER DISTRIBUTED DATA INTERFACE: A BRIGHT FUTURE AHEAD

Sunil P. Joshi
Advanced Micro Devices
901 Thompson Place
Sunnyvale, CA 94088

ABSTRACT

This paper provides an introductory overview of an emerging local area network (LAN) standard called the Fiber Distributed Data Interface (FDDI). The FDDI standard is being defined by the American National Standards Institute in its technical committee ANSC X3T9.5. FDDI is a 100 Megabits/sec optical fiber based token ring and uses a dual-ring topology for improved fault-tolerance. Some environments where FDDI may be applicable are described along with the FDDI features that make it suitable for these environments.

INTRODUCTION

The Fiber Distributed Data Interface (FDDI) is an emerging LAN standard which provides a high performance LAN solution for high-end applications such as interconnecting mainframes and their peripherals, and CAD/CAM applications.

The FDDI standard is being defined by the American National Standards Committee, ANSC X3T9.5, which is responsible for high-speed local area networks. The ANSC technical committee is a part of the American National Standards Institute (ANSI) which is the standards organization that represents the U.S. in the International Standards Organization (ISO). FDDI is a fiber-optic token-ring network which operates at a 100 megabits/sec data rate. The FDDI protocol has used the IEEE 802.5 token-ring protocol as a starting point and has made several enhancements to enable a cost-effective high-speed implementation.

There is a close working relationship between IEEE and ANSC to prevent overlaps in their activities. IEEE is defining standards at data rates below 50Mbps while ANSC is looking at speeds above 50 Mbps.

The initial FDDI specifications are better suited for data traffic as opposed to voice, and are referred to as the FDDI-1 standard. Aspects of FDDI-1 which affect hardware implementation are already frozen as a draft standard. As an enhancement to FDDI-1, the committee has begun work on a new definition called FDDI-2, which provides a better interface for both data and voice traffic.

The FDDI-1 specification consists of four documents specifying the data link layer and physical layer of the seven-layer Open Systems Interconnect model (see fig. 1). The Physical Media Dependent (PMD) document deals with the fiber optic cable, connectors and jitter specifications at the electro-optic interface. The Physical Layer (PHY) document specifies the clock recovery and encoding mechanisms. The Media Access Control (MAC) document specifies the token-passing protocol and the Station Management (SMT) document deals with network management issues.

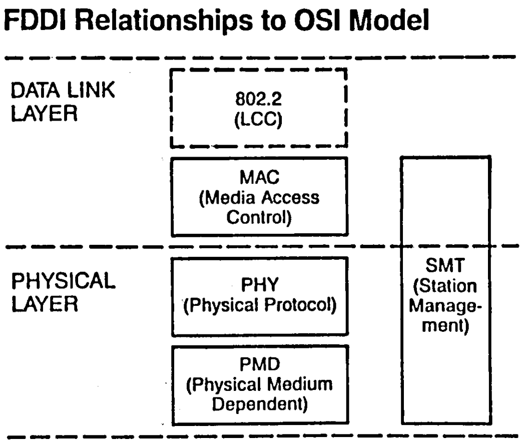

FDDI Relationships to OSI Model
DATA LINK LAYER
802.2 (LCC)
MAC (Media Access Control)
PHYSICAL LAYER
PHY (Physical Protocol)
PMD (Physical Medium Dependent)
SMT (Station Management)

Figure 1: Scope of FDDI Specifications

Why Optical Fiber

FDDI decided to use optical fiber as a transmission medium for a variety of reasons, including:

a. Bandwidth: Fiber cable provides a very high capacity for carrying information. The data rate can be in the range of several hundred megabits per second.

b. Attenuation: Fiber provides low attenuation, resulting in efficient communication over several kilometers without repeaters.

c. Noise Immunity: Fiber cables transmit information as light and neither generate nor are affected by electromagnetic interference.

d. Security: Since it is not easy to tap an optical fiber cable without interrupting communication, fiber is more secure then copper cable from malicious interception.

e. Cost: The cost of optical fiber cable has fallen considerably over the last two years and fiber is projected to become cheaper than co-axial cable in a year or so. Since fiber is lightweight and thinner, it is often easier to pull through overcrowded ducts, resulting in lower installation costs. In fact, the additional cost of putting a second fiber in the same cable is negligible; hence, FDDI has opted for a duplex fiber cable and provided a high-level of fault-tolerance through this added redundancy.

In the past, the cost of the laser diodes or avalanche photo diodes to interface with the fiber has been high. However, technological advances have provided low-cost LEDs and pin-diodes that can operate at the 100Mbps rate specified by FDDI.

The physical ring topology is ideally suited for use with fiber because it provides a point-to-point connection between nodes and therefore does not require taps which are harder to build with fiber.

MAJOR ENVIRONMENTS

FDDI is targeted towards four major environments: office floor, computer room, factory floor, and campus interconnections. Fig. 2 illustrates how a variety of applications could co-exist in a typical company environment.

Office Floor

A typical office environment consists of word processors, desk-top personal computers, facsimile machines, terminals, and printers. Most of these are low-cost, medium performance devices which feature direct user interfaces and require relatively slow networks for interconnection.

In addition to these applications, offices in the engineering environment will have Engineering Workstations, Computer Aided Design (CAD) equipment, graphics or imaging machines, and multi-user micro or mini-computers. The need to handle computation-intensive operations or moving large blocks of data, sets this equipment apart from PC-type devices. The performance of the network is crucial when hundreds of millions of bytes of graphics data are being moved, and as such these devices have acceptable, but higher network interface costs than terminals or PCs. Some applications will benefit from the higher data rate offered by FDDI.

Computer Room

The computer room can be visualized as a collection of very high performance mainframes or mini-computers connected over a backend network to peripheral controllers, communications controllers, file servers, and data-base machines. The peripheral controllers in turn can attach to individual disk drives or tape drives. In addition, there may be high speed laser printers on the backend network.

Usually the backend networks in a computer room do not cover a large span, but they must operate at a very high data rate and be extremely reliable and fault tolerant. In the past, proprietary networks have been used for computer room connections. Also, parallel interconnect schemes such as the Intelligent Peripheral Interface (IPI) have been used since the distance limitations are not a concern. Now FDDI is emerging as an alternative backend network offering increased performance.

Factory Floor

Networking is an essential element of the automated factory. A factory floor will typically include many kinds of controllers (for numerical, robotic and process control) as well as data acquisition and display devices and imaging and computing equipment.

Deterministic or time-bounded access to the network is a key requirement of a factory network. Process control applications, such as nuclear reactors, need periodic updating of control information within a guaranteed access time.

The first-generation factory network will be the IEEE 802.4 token-bus network, which forms a subset of the large Manufacturing Automation Protocol (MAP) being defined by several companies in the U.S. MAP networks will use a combination of baseband and broadband transmission on the cable.

FDDI would be useful in the factory environment when either higher-speed or greater noise immunity is needed.

Campus LANs

The backbone LAN can connect the different networks in various buildings on a campus spanning several miles.

The connections to the backbone network can be gateways which provide the necessary protocol conversion and buffering between two LAN's. A backbone network, however, does not have to merely connect smaller networks. It could very well have a direct interface to the mainframe computers in each building. Also, it is desirable to interconnect distributed PBXs carrying both voice and data using the same backbone. If the transmission of real-time voice is needed then the network has to be able to provide

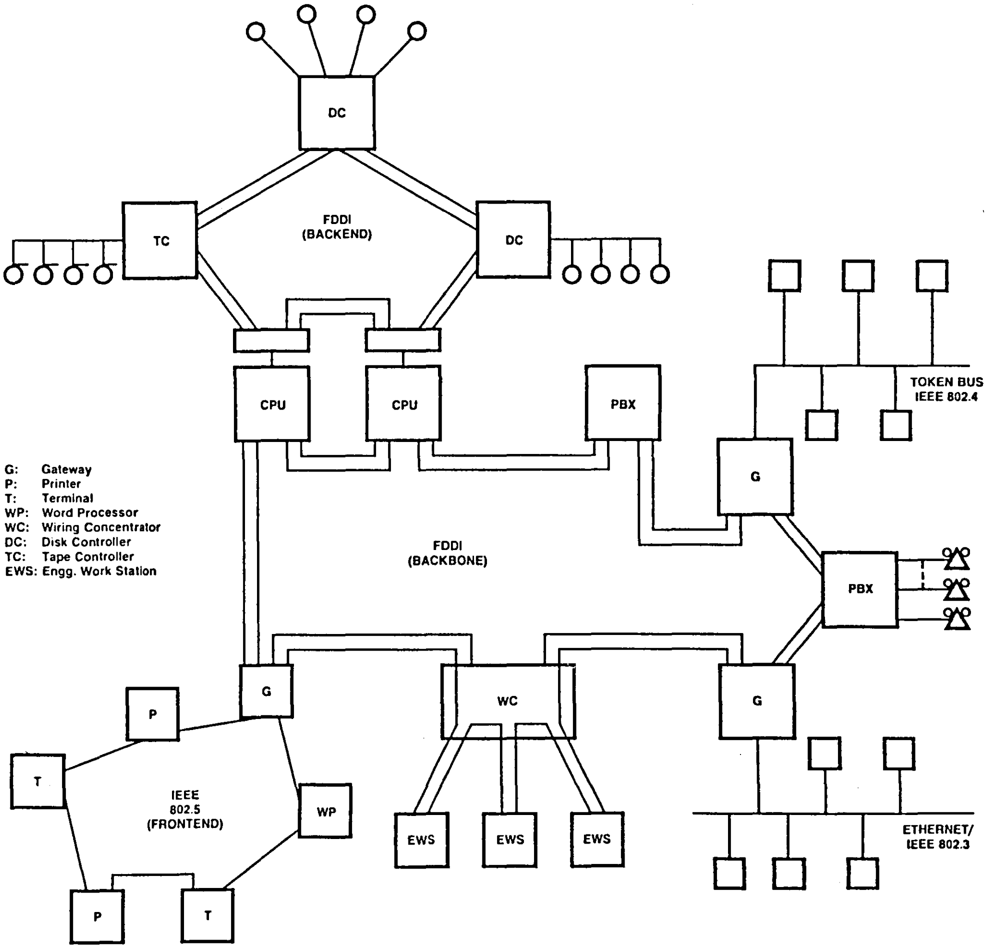


Fig. 2: A View of a Company-Wide Network

some circuit-switched capability in addition to packet-switching. Once this is possible, transmission of real-time video also is easy to provide.

Since a campus network is a backbone network, it should have a high data rate and operate over extended distances.

FDDI TOPOLOGY AND PROTOCOL

FDDI provides a considerable amount of flexibility in configuring its topology and also provides a protocol suitable for a variety of applications.

Configuration Limits

The FDDI specification places no lower limits on the number of stations and the distance between the stations. There are also no absolute upper limits. For example, for the sake of calculation of default timer values, and to keep the ring latency down to a few milliseconds, a system is assumed to include up to 1000 stations on the ring, with up to two kilometers between adjacent stations and up to 200 kilometers of total fiber cable. However, these parameters can be mutually adjusted to produce the optimum configuration for the chosen application, (for instance more stations can be allowed over shorter distances).

A Dual-Ring Approach

The FDDI ring is a combination of two independent counter-rotating rings, each running at a 100 Mbps data rate. If both rings operate simultaneously, the effective transmission rate is 200 Mbps. It is also possible to have configurations in which one ring connects all the stations, with the second counter-rotating ring connecting only a few select stations.

Figure 3 shows a possible FDDI configuration with optical fiber cables interconnecting to form the rings; the paths through which the data circulates around the ring are also depicted. The ring which reaches all the stations is termed "primary". The secondary ring carries data in the opposite direction, which is useful during ring reconfiguration. The advantage of two rings is that if one ring fails, the network can reconfigure using the other ring and still keep operating.

Class A and Class B Stations

The stations which connect to the FDDI rings are divided into two categories: Class A and Class B. A Class A station is one which connects to both rings simultaneously, and a Class B station is

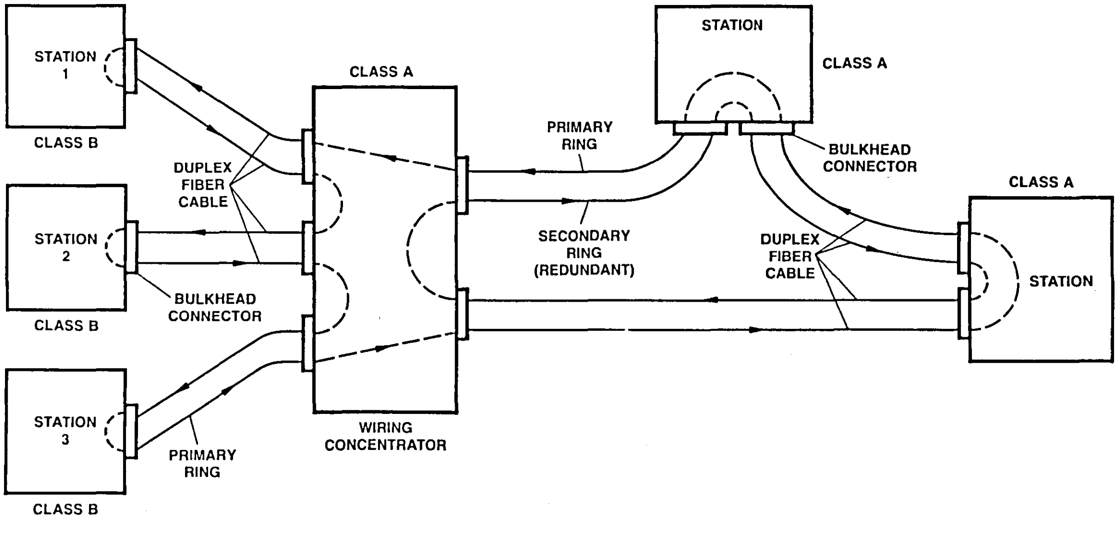


Fig. 3: FDDI Dual Ring

one which connects to only one of the rings.

The two classes help tailor the complexity of systems to meet cost objectives. Since Class B stations need to connect to only one ring, they can be implemented at lower cost. The disadvantage is that the Class B station is isolated if its connection to the wiring concentrator fails. Class A stations, on the other hand, require additional hardware to connect to dual rings, but are protected against failure. If there is a link failure they can keep operating in a reconfigured ring.

Typically those stations that need more fault tolerance will be configured as Class A. Less critical stations can be Class B. Another kind of Class A interconnect is a wiring concentrator (WC).

Wiring Concentrator

A wiring concentrator, as the name implies, is a hub node through which several other stations can be connected. A WC allows a physical ring to be easily maintained like star networks. The WC can be a service point, and several WCs can be established at various distributed locations from which shorter connections can be taken out to individual stations. A WC is always a Class A station and other Class A and B stations can connect to it.

The way the fiber cable is packaged makes the concept of WCs very attractive. The fiber cable in FDDI contains two physical fibers packaged in one jacket with a bulkhead connector on either end. The same cable is used for all class A and class B connections. For example, for the dual ring connecting the Class A stations each jacketed cable would contain one fiber for the primary and one for the secondary; for the the Class B stations the identical cable would carry the incoming and outgoing signals on the same ring for a station. Hence, when a Class B station has to be connected to a WC, only one physical cable has to be routed between the station and the WC. Because it carries two fibers, a ring path is physically established.

Fault Tolerance In FDDI

Several levels of fault tolerance are possible in FDDI. Some types of cable faults may cause the ring to reconfigure. Loss of power on certain stations may switch an optical relay in place.

Ring Configuration

Some of the faults in a network are caused by either failed components or broken cables. Even if a connection is not fully broken, there may be a substantial degradation which shows up as an increase in the bit-error rate. Figure 4 shows how

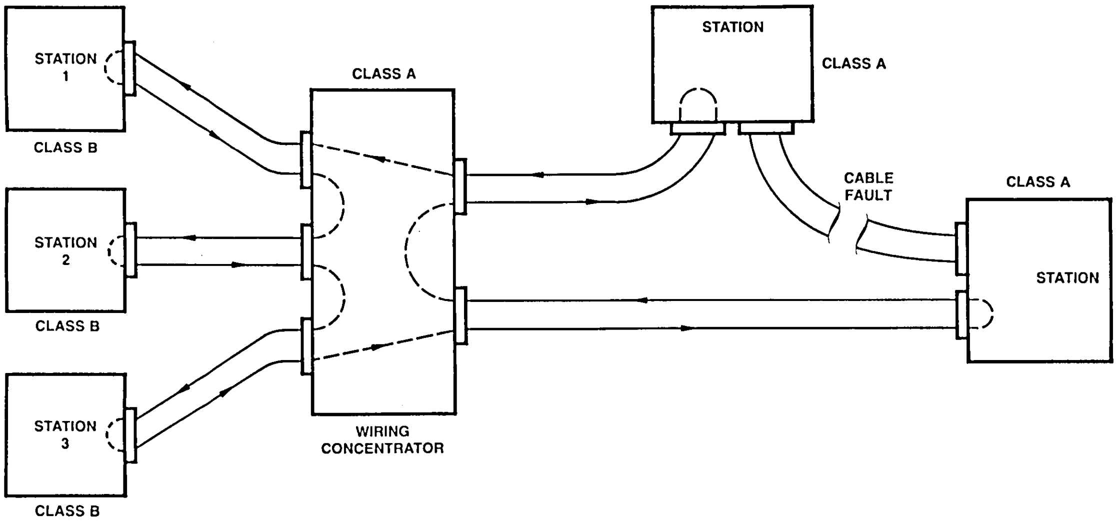


Fig. 4: Ring Reconfiguration

a ring would reconfigure its data paths if a link (or a pair of links in a cable) in the dual ring becomes inoperative. The stations would sense the breakage and use the appropriate paths on the secondary ring to keep the network running. In FDDI this reconfiguration happens automatically, within a few milliseconds. A station management interface is being defined in FDDI to facilitate this. When a broken ring is restored, the station management handshake will allow the ring to go back to its original state.

The effect of a second failure in the dual ring is interesting too. Figure 5 shows how a network can split into two smaller independent networks with both remaining operative internally.

If there is a cable fault in the cable going to a Class B station, this station is cut off from the network, and the WC can provide a bypass for the node.

Optical Bypass

Situations in which stations connected to the ring lose electrical power or are turned off can be taken care of by providing optical bypasses. A bypass provides a path for the light to bypass a station, using an electrically operated relay, such that when power is lost in the station a mirror directs the light through an alternative path.

A bypass can be provided in either Class A or Class B stations. Most Class A stations that include WCs will probably implement bypasses to get this additional level of fault tolerance. A Class A station without an optical bypass will appear as if all of its connections to adjacent stations are broken when powered off. Optical bypasses (or even electrical bypasses) may be used in WCs at their interface to Class B stations to take care of situations where the Class B stations may fail.

Encoding Scheme

Several of the low-speed standards, including the IEEE standards, use Manchester encoding for baseband transmissions. Unfortunately, Manchester encoding is only 50% efficient, and its use in FDDI's 100 Mbps data rate would have required 200 megabauds on the medium, with the LEDs and PIN receivers operating at 200 MHz.

FDDI uses a more efficient encoding scheme, called 4B/5B, to keep the baud rate down. In 4B/5B, the encoding is done on four bits at a time to create a five cell "symbol" on the medium. The

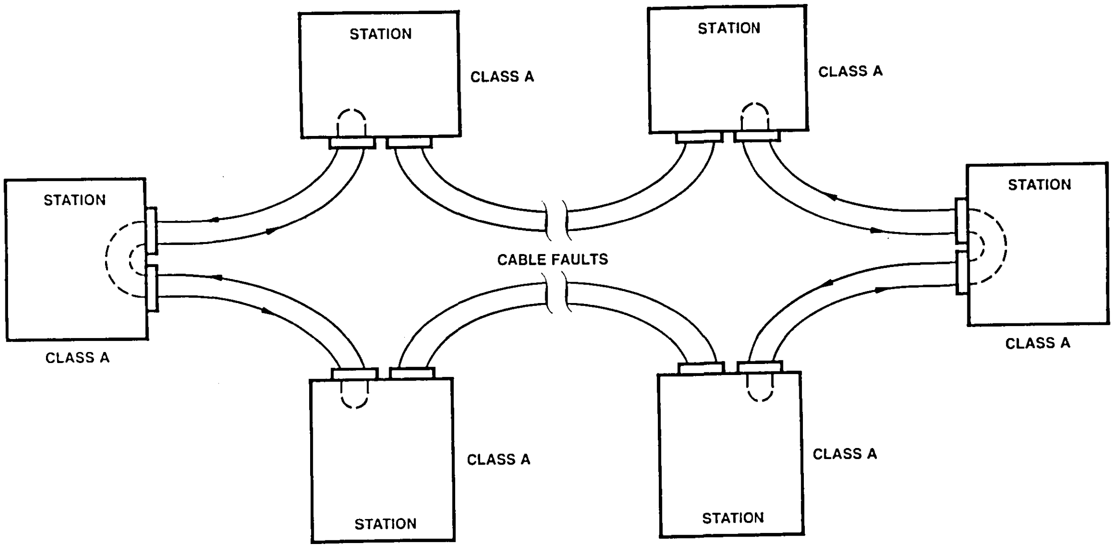


Fig. 5: A Ring with Two Cable Faults

efficiency is 80%, and at 100 Mbps the baud rate becomes 125 megabauds. The advantage is that inexpensive LEDs and PIN diode receivers, which only have to operate at 125 MHz, can be used.

TOKEN-PASSING RING

FDDI uses a token-passing ring consisting of stations serially connected by a transmission medium to form a closed loop. The packets are transmitted sequentially from one station to the next, where they are retimed and regenerated before they are passed on to the following (or "downstream") station. The idle stations can either be bypassed or function as active repeaters. The addressed station copies the packet as it passes by. Finally, the station that transmitted the packet strips the packet off the ring.

A station gains the right to transmit when it has the token, which is a special packet that circulates on the ring behind the last transmitted packet. A station wanting to transmit captures the token, puts its packet(s) on the ring, and then issues a new token, which the next station can capture for its transmission.

Access Scheme

Using an access scheme called timed-token access, FDDI allows each station a fair share of access to the network, and at the same time maintains an upper bound on the token rotation time. The token rotation time determines how often the stations get an opportunity to transmit.

In order to satisfy the requirements of various applications a station needs to know two things. One is the maximum latency before the token returns again to the given station. The other is the amount of traffic that the station can send once the token returns. The FDDI protocol allows the station to determine both these parameters through negotiation. The FDDI MAC document contains the necessary services for implementing these features. The SMT document, presently in definition, intends to formalize the precise protocol.

The total bandwidth available on the ring can be dynamically partitioned off as either synchronous or asynchronous bandwidth. Here synchronous refers to bandwidth that is guaranteed to a station for its use, every time it gets the token. The leftover bandwidth is the asynchronous bandwidth and is shared by all the stations.

Two kinds of packets can be transmitted on FDDI. Synchronous packets can be sent by the stations that have negotiated the synchronous bandwidth, every time the token is received. However, asynchronous packets can be sent only if the token is received early (i.e., within the upper bound agreed to). The asynchronous packets can be of one of eight priorities. Which priority to send is determined by a threshold time value.

FDDI also supports a special token class called restricted tokens. This is provided mainly for backend networks which have devices that need to control the network for data streaming.

BENEFITS TO VARIOUS APPLICATIONS

As shown in Figure 1, FDDI can be used as a backend, backbone, and frontend network. Different features of FDDI make it attractive for each environment.

Backend Network: For a backend network in a computer room, reliability and performance are particularly important considerations. In this environment, it is desirable for two stations to maintain unimpared operation, even if up to six intervening stations are powered-down (causing their optical bypass relays to be in the active connection path between the communication stations). FDDI is projected to meet this constraint when the total fiber length between communication stations is less than 300 meters.

The dual-ring, by reconfiguring, provides additional fault-tolerance. In a backend network most stations will be Class A nodes talking to both rings. If the 100 Mbps data rate is not sufficient in this application, an additional 100 Mbps is also available on the secondary ring as long as the ring is not reconfigured.

FDDI also supports a class of service called restricted token. This sets up a master/slave connection between the host computer and a peripheral controller, allowing the two to hog the network and stream data between them. Although the restricted token violates the deterministic access of a token ring, it is acceptable in a backend environment.

Backbone Network: A backbone network stretching over a campus may span several kilometers. The backbone can be viewed as a collection of trunk lines connecting several computer rooms or individual networks in the office environment or distributed PBXs. A high bandwidth on the backbone ensures that it is not a bottleneck during internetworking. FDDI is designed to allow links at least two kilometers in length between adjacent stations with no optical bypasses on either end. The total allowable fiber path length to satisfy the default timer values and ring latency constraints is 200 kilometers. Since duplex cable is used, the actual length of cable in the entire ring is 100 kilometers. The 100 Mbps data rate provides ample bandwidth for this application.

Another concern during internetworking is station addressing. Since FDDI permits both 16-bit and 48-bit addresses with physical and logical addressing support for both, as well as broadcast capability, the addressing translation across gateways and bridges is simplified.

For supporting distributed PBXs, a circuit-switched or time-slotted access method is desirable. The ANSC X3T9.5 committee is starting preliminary work on the FDDI-2 Specification which will implement a combination of circuit-and packet-switched traffic. This will make it easy to provide a gateway to the telephone network or ISDN.

Frontend Network: In the office floor situation, FDDI is suitable as a frontend network. The cost of connection can be minimized by having a preponderance of Class B stations. These Class B stations can be attached to wiring concentrators, which in turn will provide Class A connections to the backbone FDDI network.

Wiring concentrators in FDDI are required to be powered and as a result can be made intelligent enough so that optical bypassing will not be needed for Class B stations. Class B connections lower the cost of the connection since the duplicate physical layer logic is not needed. At the same time, they still form a part of the main ring and can transfer data at the 100 Mbps data rate.

FDDI provides for a separation of up to 500 meters between a Class B station and a wiring concentrator. When any of the Class B stations is powered down or its link is broken, the wiring concentrator can electrically bypass it very easily. The wiring concentrator also provides a convenient maintenance point.

FDDI VS IEEE 802.5 - A COMPARISON

The FDDI specification is an outgrowth of the 802.5 token-ring standard and includes the changes necessary to permit a cost-effective implementation at higher data rates. In addition it also includes new features not present in 802.5, that are necessary at the higher speeds.

Figure 6 shows the differences between the two standards. The point-to-point clocking in FDDI avoids the compounding of jitter from node to node and allows larger ring implementations. Also the 4B/5B encoding is more efficient and allows the use of 125 MHz LEDs at the 100 Mbs data rate. The restricted token feature in FDDI allows backend network nodes to "hog" the network for doing streaming operations.

The FDDI attempts to simplify hardware implementation in two ways. By allowing a half-duplex implementation only one set of FIFO buffers and CRC logic is needed. Also the protocol requires manipulation only at the byte boundaries instead of bit-boundaries, which makes it possible to put the logic in CMOS technology instead of ECL.

SUMMARY

The FDDI committee has made reasonable trade-offs to ensure that the technology needed for implementing FDDI is presently available, and that the standard will not be obsolete in the next several years. The standard has kept in mind the upgradability to higher speeds and additional services that will be necessary in the future.

The emergence of FDDI as a popular standard is spurring integrated circuits development from semiconductor vendors. The use of dedicated VLSI to support FDDI will dramatically lower the cost of an FDDI interface.

Advanced Micro Devices has a chip-set called Supernet in development which will provide the hardware required for implementing FDDI. Supernet consists of four chips: the Ram Buffer Controller (RBC), the Data Path Controller (DPC), the Fiber Optic Ring Media Access Controller (FORMAC) and the Encoder/Decoder (ENDEC). The RBC and DPC chips perform the buffer management functions, the FORMAC implements the token passing protocol and the ENDEC does the encoding, decoding and clock recovery.

#	FEATURE	IEEE 802.5	FDDI
1.	Data Rate	4 Mbs or 16 Mbs	100 Mbs
2.	Medium	Twisted Pair, Fiber	Fiber
3.	Configuration	Single Ring	Dual Ring
4.	Management	Centralized: Requires ring monitor	Distributed: No ring monitor needed
5.	Clocking	Synchronous: Transmit clock is derived from recovered clock.	Point-to-Point: each node transmits using its own crystal.
6.	Encoding	Differential Manchester: 50% efficient.	4B/5B Group Code: 80% efficient.
7.	Addressing	48 bit individual and group	16 or 48 bit individual and group.
8.	Access Protocol	Token Passing	Timed-Token Access: Stations can negotiate token rotation time.
9.	Miscellaneous	-	- Synchronous & asynchronous packets. - Restricted token for DASD devices.
10.	Implementation	Requires full-duplex hardware	Half-duplex hardware adequate.

Fig. 6: A Comparison of IEEE 802.5 and FDDI Rings

ALGORITHMS ARENA

Artificial Intelligence Algorithms

TRACK CHAIR: Prof. Tony Marsland
University of Alberta

Numerical Methods

TRACK CHAIR: Dr. David R. Kincaid
University of Texas at Austin

General Algorithms

TRACK CHAIR: Prof. Paul Purdom
Indiana University

PHASED STATE SPACE SEARCH

T.A. MARSLAND and N. SRIMANI

Computing Science Department, University of Alberta, Edmonton, Canada T6G 2H1.

ABSTRACT

PS*, a new sequential tree searching algorithm based on the State Space Search (SSS*), is presented. PS*(k) divides each MAX node of a game tree into k partitions, which are then searched in sequence. By this means two major disadvantages of SSS*, storage demand and maintenance overhead, are significantly reduced, and yet the corresponding increase in nodes visited is not so great even in the random tree case. The performance and requirements of PS* are compared on both theoretical and experimental grounds to the well known $\alpha\beta$ and SSS* algorithms. The basis of the comparison is the storage needs and the average count of the bottom positions visited.

INTRODUCTION

Phased search is a new variation on a method for traversing minimax game trees. Although based on SSS*[9], phased search has a range of performance which represents a continuum of algorithms from SSS* to $\alpha\beta$[2]. The $\alpha\beta$ algorithm was the first minimax search method to incorporate pruning into game-playing programs, and modified versions of it still predominate, even though more efficient pruning methods exist. For example, SSS* never visits more terminal nodes than $\alpha\beta$, achieving better pruning at the expense of a larger storage requirement. Here better pruning implies fewer terminal node (bottom position) visits, although other measures of performance, such as execution time and storage needs, may be more important. Even so, the number of bottom positions (NBP) visited is particularly relevant, because in any game-playing program the evaluation function spends significant time in assessing these nodes. For this reason, SSS* has the potential to reduce the search time significantly by the virtue of its better pruning. However, for uniform trees with a constant width of w branches and a fixed depth of d ply, SSS* must maintain an ordered list (called OPEN) of $O(w^{d/2})$ entries. Because of this abnormally high memory demand and the considerable time spent in maintaining the OPEN list, SSS* is not widely used, despite its known pruning dominance over $\alpha\beta$.

In its general form, the phased search algorithm, denoted here by PS*, has lower storage requirements than SSS*, but at the same time consistently outperforms $\alpha\beta$ for trees of practical importance[8]. The Phased Search algorithm with k phases, PS*(k), partitions the set of all immediate successors of MAX nodes into k groups (each of maximum size $\lceil w/k \rceil$) and limits its search to one partition per phase. It does not generate all the solution trees simultaneously as does SSS*, generating instead only a subset of them. The algorithm searches the partitions from left to right one at a time. Like SSS*, the search strategy within each phase of PS* is non-directional, but with a recursively sequential partitioning of the MAX nodes. Note that the storage requirement of PS*(k) is $O((\frac{w}{k})^{d/2})$, because PS*(k) searches only w/k successors at alternate levels of the game tree (i.e., at the MAX nodes).

GAME TREES

To provide a formal footing the following definitions are introduced. In a *uniform tree*, T(w,d), every interior node has exactly w immediate successors and all terminal nodes are at the same distance d from the root. The term *random tree* will be applied to those uniform trees whose terminal nodes are assigned *random* values from a uniform distribution. Such trees are commonly used for simulation as well as asymptotic studies of search algorithm performance, because they are regular in structure and are simple to analyze. In *ordered trees* the best branch at any node is one of the first w/R successors. Such a tree is said to be of order R. The higher the value of R the stronger the order. For random trees R = 1, while R = w corresponds to a *minimal tree*, that is, a tree in which the first successor is everywhere best. More useful are ***probabilistically ordered trees*** with parameter (p,R). Here it is only with probability p that the best subtree is among the first w/R successors. These definitions are useful since game tree searching algorithms have been compared on a basis of their effectiveness on random uniform trees[5,7] and on probabilistically ordered trees[3,6].

After generating the list of moves (a set of successor positions), most game playing programs use some knowledge of desirable features to sort the moves in order of merit. Often, the knowledge is quite accurate so the best successor will be found among the first few considered. Thus real game trees are not random, but have been approximated by ***strongly ordered trees***[3]. These in turn are similar to probabilistically ordered trees with p=0.7 and R=w. The experimental results reported here have been obtained from searches of both ordered and random trees, so that the effectiveness of search algorithms can be observed under different conditions. More detailed results are to be found in Srimani's thesis[8].

PHASED SEARCH (PS*) ALGORITHM

Let PS* with k partitions be denoted by PS*(k). For simplicity, it is assumed that the partitions are of equal size. That is, the width w of the uniform search tree is a multiple of the number of partitions. This is not a restriction, since PS*(k) generalizes easily to encompass arbitrary partition sizes.

Let P(n) be the Dewey-decimal identifier of the parent of a node n, let PSIZE be the size of each partition and let V(n) be the static evaluation at a terminal node, n. We will show that PS*(1) has identical performance to SSS*, and PS*(w) is equivalent to $\alpha\beta$. PS* is based on SSS*, but maintains two lists: one is like the OPEN list in SSS*, and the other is a BACKUP list to keep track of partially expanded MAX nodes. OPEN consists of triples (n,s,hi), where *n* is the node identifier, *s* is the status (an element in the set {LIVE, SOLVED}), and *hi* is a bound on the merit of that state (a real number in $[-oo,+oo]$). As in SSS*, the OPEN list is maintained as an ordered list of triples with non-increasing value of hi. The BACKUP list consists of vectors of the form (n,last,low,high), where *n* is the identifier of a MAX node, *last* is the node identifier of the last

CH2345-7/86/0000/0514$01.00 © 1986 IEEE

son of n included in OPEN, and *low* and *high* are the current lower and upper bounds on the value of node n. Whenever a MAX node in the OPEN list is solved or pruned, the corresponding vector is deleted from BACKUP.

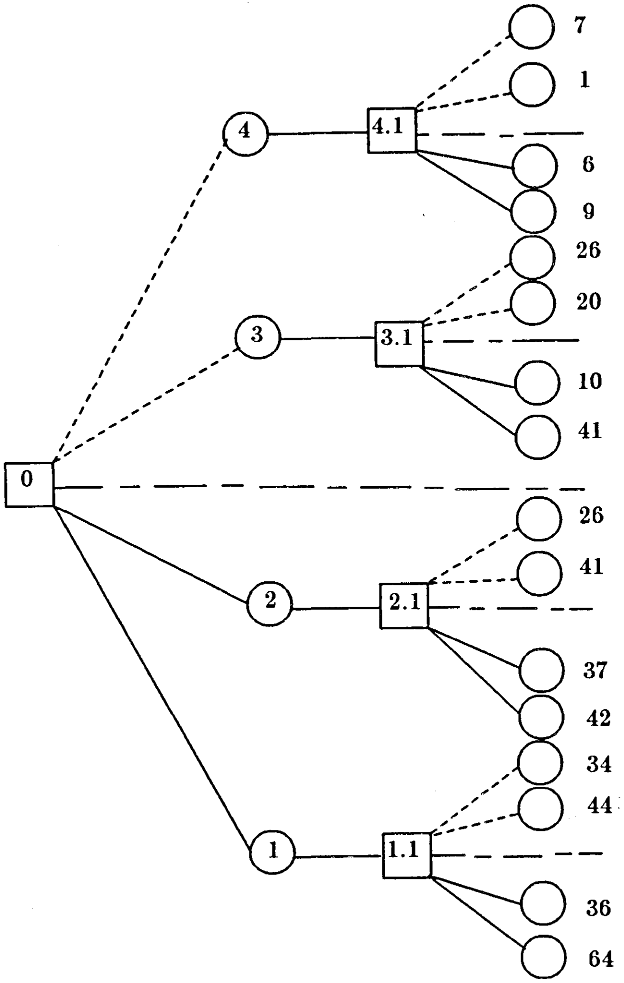


Figure 1: How PS*(2) Partitions a Tree.

The operation of phased search is seen most easily by an example. Figure 1 shows the search of a tree T(4,3) by PS*(2). Note that the successors of MAX nodes are divided into two partitions of equal size, as shown by broken lines in Figure 1. This partitioning is done recursively at each MAX node in the tree, so that the successors of a MAX node in two different partitions are never added to the OPEN list in the same phase of the PS* algorithm. Note also that, as with SSS*, at every MIN node only one successor at a time is included in the search tree. Thus for the example in Figure 1, at any instant no more than four terminal nodes are present in the OPEN list for PS*(2), while in contrast SSS* would have sixteen nodes present in OPEN simultaneously at some points in the search. Finally, note that PS*(2) will always work well if the best successor occurs in the first half of the subtrees at nodes which must be fully expanded (i.e., to use Knuth and Moore's terminology[2], at the type 1 and type 3 nodes).

Description of the Algorithm

Following the lines of Stockman's SSS* algorithm, and using his *GAMMA* operator terminology[9], PS*(k) is formed as follows:

(1) For simplicity, assume that w is a multiple of k and set PSIZE = w/k.

(2) Place the initial state (n=root, s=LIVE, hi=+oo) on the OPEN list.

(3) Repeatedly retrieve the next state (n,s,hi) from OPEN (this node has the currently highest merit, hi) and invoke the *GAMMA* operator, described in Table 1, until the termination condition is reached.

In Table 1, case 1 corresponds to the retrieval of a LIVE interior MAX or MIN node. If a MAX node is found the first partition is added to OPEN, otherwise (for a MIN node) only the first successor is taken. In the case of a MAX node, an entry is also added to the BACKUP list. For a LIVE terminal node, *GAMMA* either inserts n into OPEN with SOLVED status, or inserts the parent of n, P(n), into OPEN with SOLVED status. The choice is made in alternatives 2a & 2b and depends on how V(n), the evaluation of node n, compares to the low bound. For a SOLVED MAX node, n, *GAMMA* purges the successors of P(n) from the BACKUP list, and either adds the next successor of P(n) onto OPEN or prunes by pushing the solved parent node onto the OPEN list, cases 3b & 3c respectively. Similarly for SOLVED MIN nodes, *GAMMA* either adds another partition to OPEN or purges the pruned successor partitions. Here case 4(b) is especially complex, since it must deal with the situation when the parent MAX node has another partition to process. More than any other, it is the actions in case 4 which distinguish PS* from SSS*.

Correctness of PS*

To make it clear that the PS* algorithm always returns the minimax value, the following theorem is provided.

Theorem.
PS*(k), with its state operator *GAMMA*, computes the minimax value of the root for all trees.

Proof: It is necessary to show that

(1) PS* always terminates, and

(2) PS* does not terminate with an inferior solution.

The aim of a game tree search is to find the best solution tree. Each solution tree is a unique subtree of the game tree and is made up of all successors of each MIN node, but only one successor of each MAX node it contains. The minimax value of a game tree is the value of the best solution tree. Hence following the notation of Stockman[9], g(root) $\geq f(T_{root})$, where $f(T_{root})$ is the value of a solution tree and g(root) is the minimax value. Also, if $T0_{root}$ is the best solution tree then g(root) = $f(T0_{root})$. It follows that the algorithm always terminates after a finite number of steps, since there are only a finite number of solution trees, and any subtree once solved or discarded is not searched again.

PS* manipulates its search among different solution trees, in order to find the best one as easily as possible. Within a solution tree it carries out a minimax search, and uses the upper bound stored in the BACKUP list to help prune the search of redundant subtrees. For any solution tree, T_{root}, $f(T_{root})$ also represents the minimax value returned by PS*, provided PS* has searched that solution tree completely. All that remains is to show that PS* finds the best solution tree $T0_{root}$. By contradiction, suppose that, for some k $\geq$ 1, PS*(k) terminates with a solution tree $T1$ which is inferior to $T0$, that is, $f(T1_{root})$ $< f(T0_{root})$. This cannot happen if $T0$ and $T1$ occur in the same partition, since PS* will select the best for the same reason that SSS* does. If $T0$ is in a previous partition, then $T0$ would be SOLVED before $T1$ is encountered, and there would be a triple (n,s,hl_0) for the solution tree $T0$ such that, $hl_0 = f(T0_{root}) \geq$

$f(T1_{root})$. The value hl_0 is held as a lower bound in the BACKUP list, and so prevents $T1$ from being fully expanded and selected. Otherwise, if $T1$ is SOLVED and $T0$ occurs in one of the later partitions, the corresponding state (n,s,hl_0) would appear at the front of OPEN before the root node can be declared SOLVED. When $T0$ appears, the corresponding solution tree would be evaluated fully and found to be better than $T1$. since as the best solution tree, $T0$, cannot be pruned.

From the theorem it follows naturally that if the number of phases in PS* is k, then

for k=1, PS*(k) is equivalent to SSS* and
for k=w, PS*(k) is equivalent to $\alpha\beta$,

as far as nodes visited is concerned, since PS*(w) reduces to a depth-first left to right (directional) search. Also, for k > 1, the space requirement for OPEN is less than the $w^{d/2}$ needed for SSS*, since

the maximum size of OPEN for PS* with k partitions is at most $(\frac{w}{k})^{d/2}$ entries.

Finally the BACKUP list, for partially expanded MAX nodes, requires

$\sum_{j=0}^{\lfloor \frac{d-1}{2} \rfloor} (\frac{w}{k})^j$ entries, which is about $(\frac{w}{k})^{\lfloor \frac{d-1}{2} \rfloor}$ entries.

Thus for PS*(k) the size of BACKUP is about k/w of the size of OPEN, and the two lists together occupy significantly less space than the single OPEN list used by SSS*. Consequently, if S(A) denotes the space needed by an algorithm A, then S(PS*(k)) ≤ S(SSS*) for any k > 1 and for any depth and width of the search tree.

Comparison with other Methods

Let R be the order of the tree being searched, and let PS*(k) denote the Phased Search algorithm with k phases. Using the notation of Roizen and Pearl[7], let I(A) represent the number of bottom positions visited by algorithm A.

(1) For minimal trees (optimally ordered game trees), I(SSS*) = I(PS*(k)) = I($\alpha\beta$), because all algorithms traverse the best branch first and so achieve maximal cut-offs.

(2) For ordered trees, when p = 1 and R ≥ k, I(PS*(k)) ≤ I(SSS*) ≤ I(AB), since the best solution is always among the first w/R branches at every node in the solution tree. Although there may not be many cases where strict inequality holds, PS*(k) is at least as good as SSS* as long as R ≥ k, because the best solution is always found in the first partition. Figure 2 provides an example where I(PS*(2)) < I(SSS*), for a tree of depth 5 and width 4. Only that part of the tree which is enough to demonstrate the point has been presented. Assume that node 2.1 is solved with value 64, so the value of node 2.2 has an upper bound of 64. Consequently, 2.2.1.1.1 and 2.2.1.1.2 are solved with values 18 and 21 respectively. Then 2.2.2.1, 2.2.2.2, 2.2.2.3 and 2.2.2.4 are included in OPEN and solved with values ≥ 64, hence node 2.2.2 is solved. Note that nodes crossed in Figure 2 are visited by SSS* but not by PS*(2).

Table 1: State Space Operator (*GAMMA*) for PS*(k).

<table>
<tr><td colspan="3">k is the partition count, and PSIZE = w/k is the partition size.
Let n be the m-th successor of its parent node i, where i = P(n).
Thus, n = i.m, provided n is not a root node.</td></tr>
<tr><td>Case</td><td>Condition of the input state (n,s,hi)</td><td>Action of <i>GAMMA</i></td></tr>
<tr><td>1.</td><td>s=LIVE, n is interior</td><td></td></tr>
<tr><td>1a</td><td>Type(n) = MAX</td><td>Push states (n.j,s,hi) for all j=1,...,PSIZE onto the OPEN stack in reverse order. Push (n,PSIZE,low,hi) onto BACKUP, where low is the lower bound of n and hi is the upper bound. Note that, if n = root, then low=-oo else low = low of P(i) stored in BACKUP.</td></tr>
<tr><td>1b</td><td>Type(n) = MIN.</td><td>Push (n.1,s,hi) onto the front of the OPEN list.</td></tr>
<tr><td>2.</td><td>s=LIVE, n is terminal</td><td>Set Score = Min(V(n),hi), where V(n) is the value returned by the evaluation function.</td></tr>
<tr><td>2a</td><td>Type(n) = MIN, or Score &gt; low of P(i).</td><td>Insert (n,SOLVED,Score) into OPEN in front of all states of lesser merit. Ties are resolved in favor of nodes which are leftmost in the tree.</td></tr>
<tr><td>2b</td><td>Type(n) = MAX, Score ≤ low of P(i)</td><td>If i is the last node in the current partition at P(i), then Score is changed to low of P(i).
Insert (i,SOLVED,Score) into OPEN, maintaining the order of the list.</td></tr>
<tr><td>3.</td><td>s = SOLVED, Type(n) = MAX.</td><td>Purge all successors of i = P(n) from BACKUP.</td></tr>
<tr><td>3a</td><td>m = w, n = root</td><td>Terminate: hi is the minimax value of the tree.</td></tr>
<tr><td>3b</td><td>hi &gt; low of P(i), m &lt; w.</td><td>Expand: push (i.m+1,LIVE,hi) onto the front of OPEN.</td></tr>
<tr><td>3c</td><td>Otherwise</td><td>Prune: push (i,SOLVED,hi) onto the front of OPEN.</td></tr>
<tr><td>4.</td><td>s = SOLVED, Type(n) = MIN,</td><td>Obtain values of low(i) and high(i) from BACKUP.
Set low(i) = Max(low(i),hi) and update low for all descendants of i on BACKUP.</td></tr>
<tr><td>4a</td><td>If low(i) ≥ high(i)</td><td>Purge all successors of i from OPEN and BACKUP.
Push (i,SOLVED,high(i)) onto the front of OPEN.</td></tr>
<tr><td>4b</td><td>If low(i) < high(i)</td><td>If there are incompletely searched MAX successors (non-immediate) of node i present in BACKUP, then add the next partition of the first such node found in BACKUP to the front of OPEN;
Else Purge all successors of i from OPEN and BACKUP,
and <i>either</i> push the next partition of successors of i onto OPEN <i>or</i>, if there are no more partitions,
Push (i,SOLVED,low(i)) onto the front of OPEN.</td></tr>
</table>

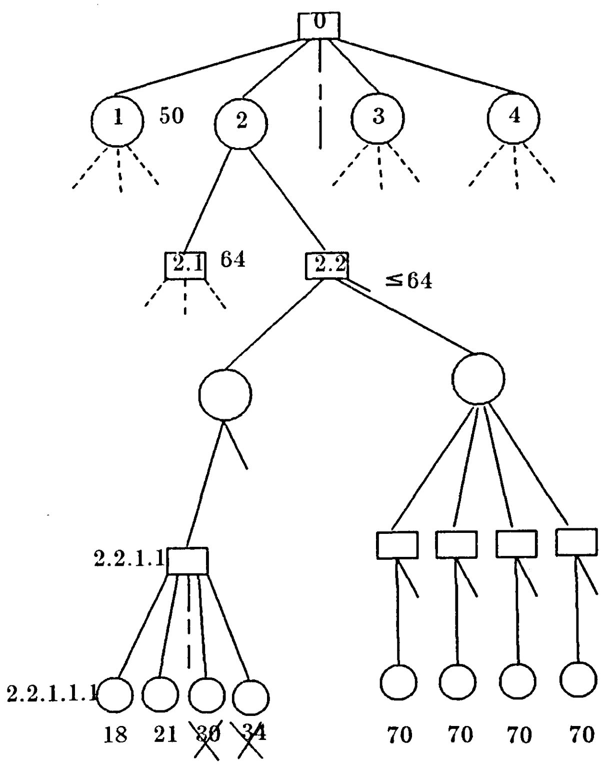


Figure 2: Tree T(4,5) in which PS*(2) is better than SSS*.

(3) For ordered trees, if p = 1 and R < k, I(PS*) can be greater than I(SSS*). Similarly, if the tree is random, then PS*(k) will occasionally evaluate some extra nodes. However, our experimental results show that even when R < k, in most of the cases (including random trees) PS*(k) is still better than the $\alpha\beta$ algorithm[8].

(4) There are trees which are unfavorable for PS*, so that I(PS*(k)) > I($\alpha\beta$). Such trees are statistically insignificant, and are uncommon in typical applications, because they represent a worst first ordering within every partition.

PERFORMANCE COMPARISON

The search algorithms PS*(k), SSS*, and $\alpha\beta$ have been implemented on a VAX 11/780 using the C language. Experimental investigations were carried out with both ordered and random trees, using different combinations of depth, width and tree ordering. Some of the results on minimal, random, and ordered versions of the uniform trees T(8,4), T(16,4), T(24,4), T(32,4) and T(8,6) are presented. For the trees of width 8, 16 and 24, orders R = 2 and 4 were searched and for trees of width 32, order 8 was also studied. For each combination, 100 different trees were generated using a modified version of the scheme developed by Campbell[1], and the average NBP visited by each algorithm are presented in the tables. The maximum amount of space needed is also given in terms of list entries.

Based on the search of 100 different trees, the following observations about the average performance of PS*(k) are possible:

(1) Data in Tables 2 through 6 show that on random trees (R = 1), the average NBP for PS*(2) is much less than for $\alpha\beta$, but more than for SSS*. For trees of order R = 2 and higher, PS*(2) and SSS* have the same performance, but it is clear that PS*(2) needs much less space.

(2) SSS* is always better than $\alpha\beta$ and is statistically better than PS* for both random and probabilistically ordered trees, Table 4. For perfectly ordered trees, each algorithm visits minimum bottom positions.

(3) Table 4 shows the results on both ordered and probabilistic trees of depth=4, width=24 and of orders R=2 and 4. In the probabilistic case, the average NBP are slightly greater, as we would expect, because at every MAX node there is some nonzero probability that the best branch is not found in the first partition searched by PS*.

(4) For most of the trees, I(PS*(i)) < I(PS*(j)), for 1 ≤ i < j ≤ w. That is, PS*(k) visits terminal nodes in increasing number with increasing k. There are some trees for which this is not true[8]. It is also known that the above relation marginally fails to hold for ordered trees (probability p=1), see for example Tables 5 and 6 where PS*(2) and PS*(4) often have statistically insignificant better performance than SSS* (i.e., PS*(1)) on order R = 4 trees.

Table 2: Average NBP on Trees with depth=4 and width=8.

Search method	R = 1 (random)	R = 2	R = 4	R = 8 (minimal)	Space needs
SSS*	439	287	190	127	64
PS*(2)	571	286	190	127	21
PS*(4)	634	375	190	127	7
$\alpha\beta$	689	415	248	127	4

Table 3: Average NBP on Trees with depth=4 and width=16.

Search method	R = 1 (random)	R = 2	R = 4	R = 16 (minimal)	Space needs
SSS*	2250	1637	1146	511	256
PS*(2)	2829	1637	1146	511	73
PS*(4)	3363	2114	1146	511	21
PS*(8)	3743	2388	1496	511	7
$\alpha\beta$	3952	2981	1664	511	4

Table 4: Average NBP on Trees with depth=4 and width=24.

For minimal trees of depth 4 and width 24 NBP=1151.						
		prob=1.00		prob=0.90		
Search method	R = 1 (random)	R = 2	R = 4	R = 2	R = 4	Space needs
SSS*	5805	4423	3206	4702	3513	576
PS*(2)	7345	4423	3203	4956	3690	157
PS*(4)	8650	5718	3201	6460	3940	43
PS*(6)	9207	6222	3950	7126	4649	21
PS*(8)	9753	6652	4300	7517	4938	13
$\alpha\beta$	10602	7437	5031	8364	5660	4

Table 5: Average NBP on Trees with depth=4 and width=32.

Search method	R = 1 (random)	R = 2	R = 4	R = 8	R = 32 (minimal)	Space needs
SSS*	10816	8493	6424	4633	2047	1024
PS*(2)	13989	8478	6422	4632	2047	273
PS*(4)	16464	11089	6420	4632	2047	73
PS*(8)	18512	12782	8313	4631	2047	21
PS*(16)	20145	13966	9330	6209	2047	7
$\alpha\beta$	20836	14665	10046	6974	2047	4

Table 6: Average NBP on Trees with depth=6 and width=8.

Search	R = 1	R = 2	R = 4	R = 8	Space
method	(random)			(minimal)	needs
SSS*	6044	3475	1932	1023	512
PS*(2)	9984	3437	1921	1023	85
PS*(4)	11283	5213	1915	1023	15
$\alpha\beta$	11565	5555	2659	1023	6

Choice of Partition Count

From the previous discussions, it is clear that selection of the partition count, k, is important if PS*(k) is to achieve its maximum benefit. If from some previous knowledge we know that the tree is of order R, we can choose k = R. Then I(PS*(k)) would be the same as I(SSS*), but the storage requirement of PS*(k) would be about $1/(k^{d/2})$ of that of SSS*. Clearly, there is a trade-off between space and bottom positions visited. If k=w, minimum space is required, but NBP will increase to that of an $\alpha\beta$ search. On the other hand, if k=1 the NBP would be low but space needed would be as much as for SSS*. Thus PS* forms a continuum of alternatives between SSS* and $\alpha\beta$. PS* can be made effective by using information about the ordering properties of game trees, since one can choose the parameter k both on the basis of the tree ordering and on the memory space available. Different ordering schemes must be considered, since ordered trees often are more typical of those appearing in applications than are random trees.

Storage needs are also significant. For example, SSS* needs 1024 entries in the OPEN list to search a tree of depth=4 and width=32, whereas PS*(4) requires 64+9 = 73, and PS*(8) needs only 16+5 = 21 for both the OPEN and BACKUP lists. Note that although PS* maintains two ordered lists, the total size of the two lists is much less than that of the single list of SSS*. Also, an ordered list of size 64 or 16 is much cheaper to maintain than a list of 1024 elements. Hence, the time spent by PS* manipulating these overhead lists may be less than that needed by SSS*.

CONCLUSION

The new algorithm PS*(k) can be viewed as a continuum between SSS* and $\alpha\beta$, as it attempts to make use of the best characteristics of both. The $\alpha\beta$ algorithm processes nodes in a game tree much faster than SSS*, but SSS*, making more use of the knowledge gained at earlier steps, prunes better than $\alpha\beta$ and as a result visits fewer bottom positions. SSS* achieves this better pruning at the expense of extra bookkeeping which needs more storage and considerable time for the update process. The phased search algorithm PS* also does some bookkeeping and achieves much better pruning than $\alpha\beta$ in a statistical sense. Since PS* concentrates only on a subset of the solution trees in each phase, it consequently needs smaller storage and may even require less execution time than SSS*. Thus PS*(k) can be comparable to SSS* in performance, especially on *bushy* trees (i.e., trees with w > 20), and yet at the same time has significantly lower storage overhead than SSS*. Because of the built-in flexibility provided by phasing and the possibility for choosing the partition size parameter (PSIZE), PS* is expected to be useful in practice. PS* becomes most efficient if parameter selection can be done using some a priori knowledge of the expected location of the solution.

Experimental results reported here are based on a game tree model, and the algorithm remains to be tested with a typical game-playing program. However, experience with other alternatives to $\alpha\beta$[6] shows that performance on probabilistic uniform trees is a good indicator of performance in a typical application[4]. In the work reported here, the successors of a MAX node in the PS*(k) algorithm are divided into partitions of equal sizes. This is not a restriction, but further work is necessary to determine if unequal partition sizes offer a performance advantage in practice. Certainly for probabilistically ordered trees increasing partition sizes could be useful.

Acknowledgements

Financial support in the form of Canadian Natural Sciences and Engineering Research Council Grant A7902 made the experimental work possible. Discussions with Liwu Li on theoretical points and the independent implementation of PS* by Erik Altmann helped and were appreciated.

References

1. M.S. Campbell and T.A. Marsland, A comparison of minimax tree search algorithms, *Artificial Intelligence 20(4)*, (1983), 347-367.
2. D. Knuth and R. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence 6(4)*, (1975), 293-326.
3. T.A. Marsland and M. Campbell, Parallel search of strongly ordered game trees, *Computing Surveys 14(4)*, (1982), 533-551.
4. T.A. Marsland, Relative efficiency of alpha-beta implementations, *Procs. 8th Int. Joint Conf. on Art. Intell.*, (Los Altos: Kaufmann), Karlsruhe, West Germany, Aug. 1983, 763-766.
5. A. Musczycka and R. Shinghal, An empirical comparison of pruning strategies in game trees, *IEEE Trans. on Systems, Man and Cybernetics SMC-15*, 3 (1985), 389-399.
6. A. Reinefeld, J. Schaeffer and T.A. Marsland, Information acquisition in minimal window search, *Procs. 9th Int. Joint Conf. on Art. Intell.*, Los Angeles, 1985, 1040-1043.
7. I. Roizen and J. Pearl, A minimax algorithm better than alpha-beta? Yes and No., *Artificial Intelligence 21(2)*, (1983), 199-220.
8. N. Srimani, A new algorithm (PS*) for searching game trees, M.Sc. thesis, Computing Science Dept., University of Alberta, Edmonton, July 1985.
9. G.C. Stockman, A minimax algorithm better than alpha-beta?, *Artificial Intelligence 12(2)*, (1979), 179-196.

IMPROVED PARALLEL ALPHA-BETA SEARCH

Jonathan Schaeffer

Computing Science Department,
University of Alberta,
Edmonton,
Canada T6G 2H1

ABSTRACT

Conventional parallelizations of the alpha-beta algorithm have met with limited success. This paper describes a parallel alpha-beta searching program that achieves a high degree of parallelism through the use of four different types of processes: Controllers, Searchers, Table Managers, and Scouts. Improved performance is achieved through decreased synchronization overhead, increased distribution of information, and increased awareness by "scouting" ahead in the tree looking for interesting features. Experimental data is presented showing a 5.7-fold speedup with 9 processors.

1. Introduction

Attempts to parallelize the *alpha–beta* tree search algorithm have been less than successful. Simulation results have been published for several inventive parallel approaches showing tremendous speedups ([1,2] for example). However, there exists a large gulf between theory and practice; actual implementations have shown only modest speedups [3-5]. Parallel programs suffer from efficiency losses for a variety of reasons. For the alpha-beta algorithm, synchronization overhead, the cost incurred by processors becoming idle, and search overhead, the consequence of building a larger tree, appear to be the biggest obstacles to improved performance.

This paper introduces some new ideas for enhancing the performance of a parallel alpha-beta searcher. The ideas are discussed in the context of a computer chess program operating in an environment of a network of loosely coupled processors.

One way of reducing the synchronization cost is to re-assign idle processors to help out busy ones. This paper gives some experimental results of such a scheme and shows that synchronization is reduced at the cost of increasing search overhead. With the aim of reducing the search overhead, increased flow of information between processes is experimented with. Two different means of sharing information were tried: a table manager process to share results among processors (used for the transposition table [6]), and periodic merging of local tables and distributing the combined result (used for the history tables [7]). Search overhead is decreased but at the cost of increased synchronization and communication. The result is a 5.7-fold speedup with 9 processors.

Despite these improvements, beyond a handful of processors, parallel alpha-beta performance rapidly degrades. Fishburn has shown that for N processors, under certain conditions, the possible speedups are $O(\sqrt{N})$ [8]. The speedup achievable with 100 processors may not be much greater than that attained by 10, and may even be less! Accordingly, a different means is required to use profitably additional computing resources. The idea of a Scout process is introduced; a process that searches ahead of the main program in the tree "scouting" for interesting lines. The Scouts are small, fast, stripped-down chess alpha-beta searchers. They look for wins and losses of material and communicate this information back to the main program. Thus while a chess program may only see 7 half-moves (or 7-ply) deep in the tree, the Scouts can often search an extra 2-3 ply deeper. This helps increase the tactical awareness of the program and reduce the horizon effect [9] by allowing a chess program to find wins (and avoid losses) not normally possible within its search horizon.

These ideas have been implemented in the computer chess program *Phoenix* †. Experimental data is presented on the effectiveness of the schemes.

2. Parallel Alpha-Beta

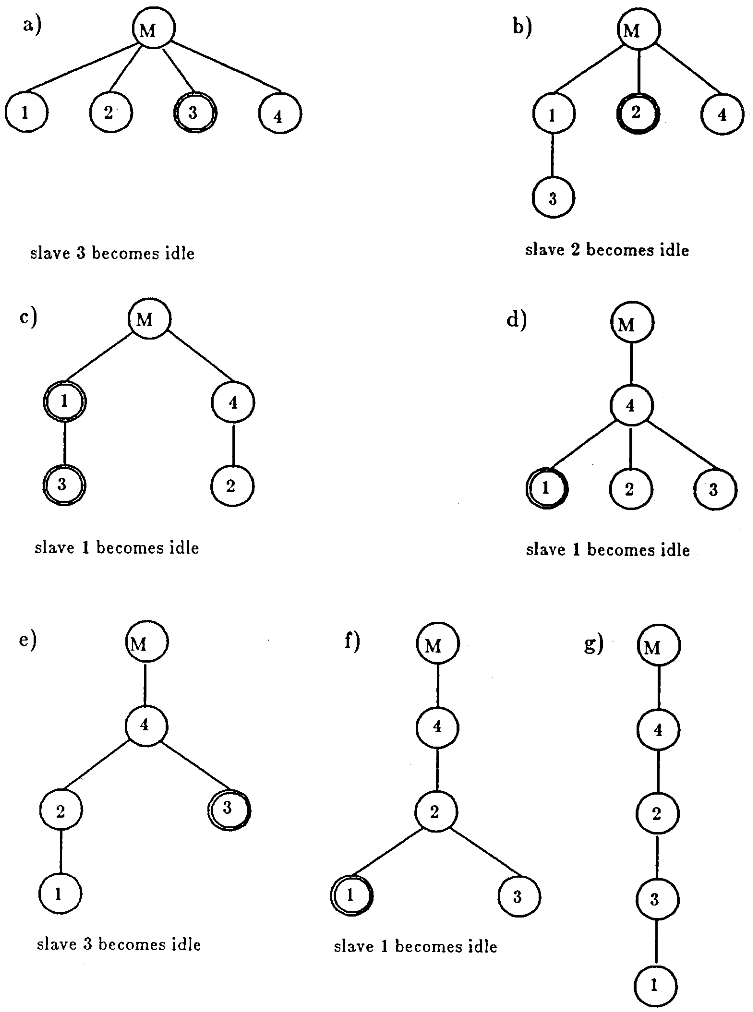
Many different approaches have been discussed in the literature for parallelizing the alpha-beta algorithm. They include parallel aspiration search (limited to a 5-6-fold speedup) [10], tree-splitting (too simple, limited performance) [11], mandatory work first (too much overhead) [2] and principle variation splitting (PVSplit) [12].

The *PVSplit* algorithm applies a recursive form of *tree-splitting* [8] to achieve parallelism. The w sub-trees originating at the root of the tree are in turn assigned to a process to search. However, the searching of the first sub-tree from the root is usually more expensive than the others, perhaps comprising 50% of the work. Accordingly, *PVSplit* applies tree-splitting recursively at interior nodes along the first path from the root to terminal node (i.e. along the *principal variation*). At each point where tree-splitting occurs, the processes must synchronize before continuing. The search is conducted in iterations, searching all moves from the root 2-ply deep, then 3-ply, then 4-ply etc. After each iteration, further synchronization is necessary. The communication structure of the N processes used can be viewed as a process tree of depth 1 and width N. The process distributing the work is called the *Controller*, while the N processes actually doing the work are known as *Searchers*.

The losses in performance of this algorithm are primarily two-fold. First, synchronization is required at several places in the algorithm. At these points, processors become idle waiting for others to complete their work. Experiments have shown that as the number of computers increases, this overhead quickly becomes dominant [3]. The second loss is extra searching as a result of incomplete information. On a loosely coupled network of processors, results computed on one processor are not globally available to others and hence may be re-computed. The search overhead increases steadily but appears to level off eventually [3].

To help reduce the synchronization costs, the *PVSplit* algorithm was modified to allow idle processes to help other processes complete their tasks. This is done by changing each Searcher to maintain a list of its sub-trees that have yet to be searched. When a process becomes available to help out, work can be taken from the top of the list and off-loaded to that process. Thus at any time, any Searcher could be working for any other. The initial process communication structure is a tree of depth 1 and width N (Figure 1a). The Controller process at the root assigns sub-trees for evaluation to the N Searchers. When the Controller has no more work for a Searcher to do, that process is re-assigned to work for another that is still busy. In this way, the tree-like configuration of processes is retained; the tree just changes its shape dynamically. Figure 1 illustrates a sample scenario with

† A competitor in the 1986 World Computer Chess Championships.



slave 3 becomes idle

slave 2 becomes idle

slave 1 becomes idle

slave 1 becomes idle

slave 3 becomes idle

slave 1 becomes idle

Figure 1. Re-Assigning Idle Processes

a Controller and 4 Searcher processes; successive diagrams illustrate the effect as a process becomes idle.

Note that synchronization is not eliminated by this approach; it is only reduced. At points where work becomes divided, the process distributing the work ("employer") must wait until all the work handed out has been completed before moving onto other tasks. In such cases, it is usually only one process, the employer, that must remain idle. For example, in Figure 1c, when process 1 finishes its work, it cannot be re-assigned until process 3 is done. The worst case scenario occurs on the last piece of work. As in Figure 1g, only one process may be busy with the others waiting for its completion. In *PVSplit*, at all synchronization points, $N-1$ processes remain idle waiting for the return of the last result.

This attempt to reduce synchronization is not without problems. First, the amount of communication is increased. Second, the search overhead would be expected to increase. Whereas formerly a subtree would be entirely searched by one processor with all the relevant information locally available, by splitting it up over several processors, not all of this information is available and extra work is performed. Third, the complexity of the program is increased. Although this effect is difficult to quantify, it not only affects the implementation and debugging time, but the execution time of the program as well.

Nodes in a chess tree are not necessarily unique. Transpositions occur, whereby two different sequences of moves can result in the same position. Transposition tables are used by chess programs to remember subtrees that have been searched [6]. On reaching a new position, the program can interrogate the table before doing the search in the hope that the sub-tree has been seen before. Unfortunately, in a multi-processor environment the transposition table information is distributed over all the processors. To facilitate sharing of this information, a Table Manager process has been added that reduces the searching each process does by making the results of other Searchers available. The Table Manager receives table entries from the Searchers and responds to their queries for information from the table.

This attempt to reduce search overhead is not without problems. First, it is achieved at the cost of increased communication in the program. Second, synchronization overhead is increased because table queries are performed synchronously; a process must wait until it gets the result before it can continue. Allowing a process to proceed and eventually be interrupted when the result is available would complicate the implementation and has not yet been tried.

The *history heuristic* is a simple means of accumulating information about a search tree that can be used for deciding the order in which sub-trees should be examined at interior nodes. For sequential chess programs, the algorithm has been successful [7]. In a parallel environment, some of its effectiveness is lost because each processor has only part of the history information. This suggests that processors should share this information. After each iteration, all processors could communicate their history tables to the Controller who accumulates them and then replies to each with the updated global table. In this way search overhead may be reduced at a cost of increased communication.

The above three ideas (re-using idle processors, table managers, and shared history heuristic information) individually can improve the performance of a parallel alpha-beta searcher. However, as the preceding discussion has shown. these enhancements can run counter to each other.

3. Scouts

Whatever depth an alpha-beta search is performed to, there is always the possibility that by searching deeper, a better result may be found. In chess programs this is particularly acute; a program may make a move that, for example, within a 5-ply search looks good, but a 6-ply search may reveal its deficiencies. Considerations such as this have motivated chess programmers to acquire the fastest possible hardware. It is not a coincidence that the best chess programs today can also search the deepest.

To overcome this problem, the idea of a Scout is introduced. Scouts are stripped down versions of Searchers. They are designed to be as fast as possible, scouting ahead in the tree looking for wins and losses of material at a depth beyond what Phoenix could usually search. All the chess expertise that allows Phoenix to play a good game of chess has been removed. It evaluates positions solely on the net material balance of the position.

Because of its simplicity a Scout can execute at twice the speed of a Searcher process. This is not enough; on average, a factor of 3-8 is required to search an extra ply. Since Scouts are only interested in finding tactically interesting lines, some heuristics are added to the search algorithm to eliminate uninteresting lines of play. For example, a line in which a series of moves are made without any threat is classified as uninteresting and ignored. These heuristics introduce error into the search in that some tactically interesting lines may be overlooked. It turns out that these type of positions occur infrequently in practice, whereas the reduction in tree size and resulting deeper searches are significant.

Scouts can be easily implemented in parallel by using the same routines used to parallelize Phoenix. They have their own Controller process to distribute the work and accumulate the results. A set of Scouts with a Controller process is called *Minix* (Mini-Phoenix).

Phoenix and Minix communicate through their Controller processes. Minix searches all the moves to find out which ones are in the set of tactically best moves, those moves that result in the best material advantage for the program. When Phoenix has decided on its move, it tells Minix who checks to see if the choice is tactically good. If so, it gives permission to Phoenix to make the move. Otherwise, Minix will veto Phoenix and order it to make a move that, from Minix's point-of-view, is tactically better.

4. Environment

A collection of workstations, with its network and communications software, may be viewed as a *multi-computer* capable of running distributed and parallel algorithms. The software facility used to implement Phoenix is a multi-computer called the Virtual Tree Machine (VTM) [13]. It is implemented on a network of autonomous VAX-11/780s, SUN-2, and SUN-3 processors each running the 4.2BSD Unix operating system [14]. The user's view of the facility is a collection of processing elements - each with its own local memory and peripherals. The VTM name is a misnomer in that arbitrary interconnects are possible, not just trees. In reality, the VTM consists of ordinary Unix processes with communication paths implemented as virtual connections between processes over a local area network. The user's interface to the machine is a set of procedures, callable from application programs, and a collection of servers that create the nodes in the virtual machine according to a description provided by the user. This description specifies the mapping between virtual processing elements (nodes) and physical processors as well as the interconnections between nodes.

Figure 2 illustrates the process/processor structure on a sample network. Each box represents a machine and each circle, a process running on that machine. Note that for illustrative purposes, both the Controller and User Interface processes reside on separate processors. In fact, they consume few resources and so can share the same processor as the Table Manager.

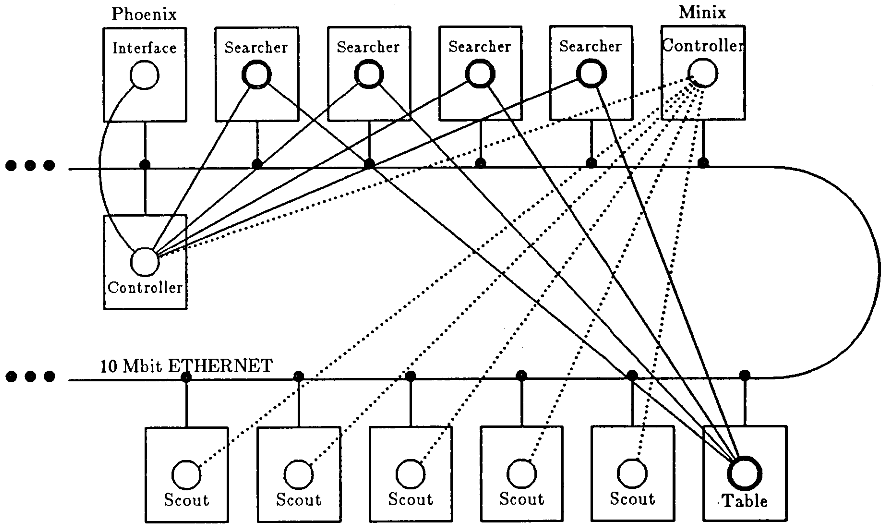


Figure 2. Phoenix and Minix

5. Results

Phoenix's PVSplit algorithm has been enhanced to reduce processor idle time and share history and transposition information. Figure 3 presents some experimental results showing speedups using 1 to 9 processors. The data was obtained by searching a standard set of 24 positions [15] to 7-ply. The *B* line is for PVSplit enhanced by the re-assigning of idle processors, *BH* is *B* with the addition of shared history information, and *BHT*, *BH* with a transposition table manager. *BHT* is able to achieve a speedup of 5.7 using 9 processors.

Previous implementations of PVSplit show the performance of the algorithm tapering off as the number of processors are increased, to a point where additional processors actually *degrade* performance. In [3], their experiments suggested performance stopped improving at 8 processors with a speedup of 4.4. Newborn's results [4] are slightly higher, but also appear to tail off quickly.

The data in Figure 3 shows that through 9 processors, performance is still increasing. The effect of sharing history and transposition information increases the performance of the algorithm without, however, eliminating the decreasing returns for additional computing power. Transposition table managers clearly provide a significant improvement in performance.

Figure 4 shows the total, search, and synchronization overheads for both the *B* and *BHT* variants. Communication overhead was experimentally small enough to be considered negligible. The addition of shared history and transposition table information has reduced the search overhead by more than 50%. Experiments have shown that the re-assigning of idle processors to help out busy ones does reduce synchronization overhead, but increases search

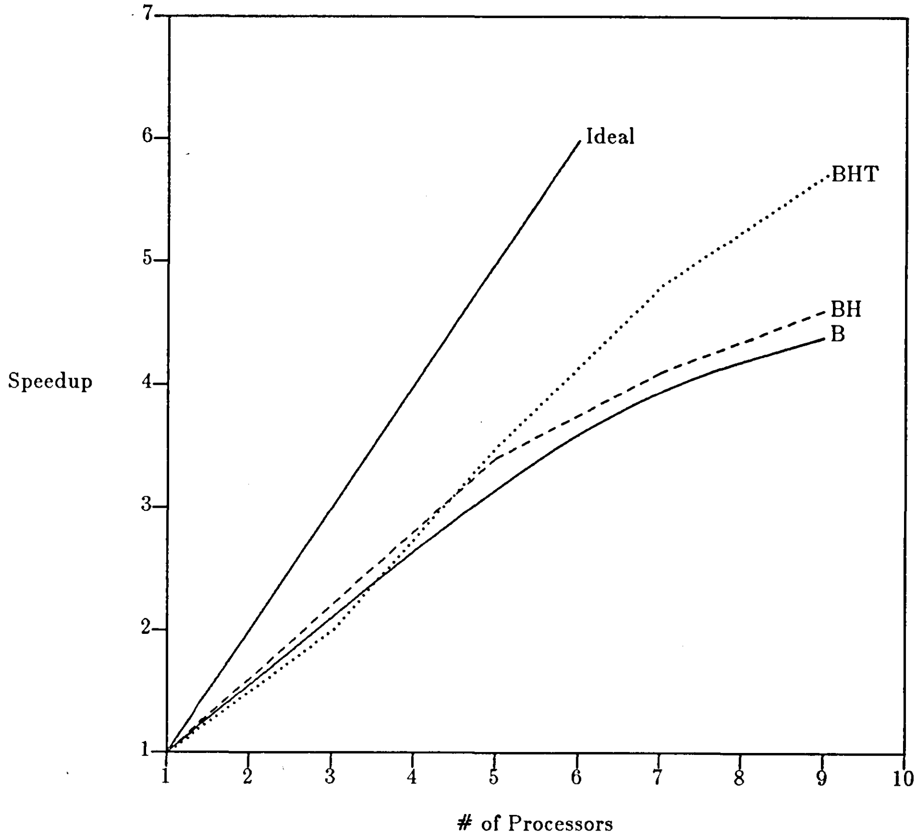


Figure 3. 7-Ply Speedups

overhead almost to the point where it offsets all the gains. Using a Table Manager at the cost of an additional Searcher helps offset this.

An anomaly appears on the graph for the *BHT* overheads with 2-4 processors. For few processors, devoting resources for a Table Manager instead of a Searcher does not pay-off in performance. Instead, with 2-4 processors one does better to use as many Searchers as possible, and for 5 or more, add a Table Manager.

Limited data is available for more than 9 processors. Experiments with 19 processors show a 7.7-fold speedup, with additional processors still providing some benefits. Overhead analysis shows that search overhead levels off, while synchronization increases dramatically. This suggests that the Table Manager process is becoming over-loaded and processors wait increasing amounts of time for response. A possible solution is the addition of a second Table Manager.

Scout processes are new and there is not a lot of experience using them. In quiet positions with few tactical possibilities, Minix can easily out-search Phoenix by 10-ply or more. In more complicated positions, it is usually 1-2-ply ahead. Unfortunately, Minix's performance is difficult to quantify. Many standard sets of chess problems contain a large number of tactical positions and on these Minix performs exceptionally well, finding a many winning moves not found by Phoenix. However, in real tournament games, the number of times that an extra one or two ply of tactical searching makes a difference in the move selected is infrequent. Since these situations do not occur often, one might argue that the computing resources are not well used. On the other hand, these winning and losing moves can be the decisive difference in a tournament game. Given that the use of Scouts results in improving even one move in a game, then they have

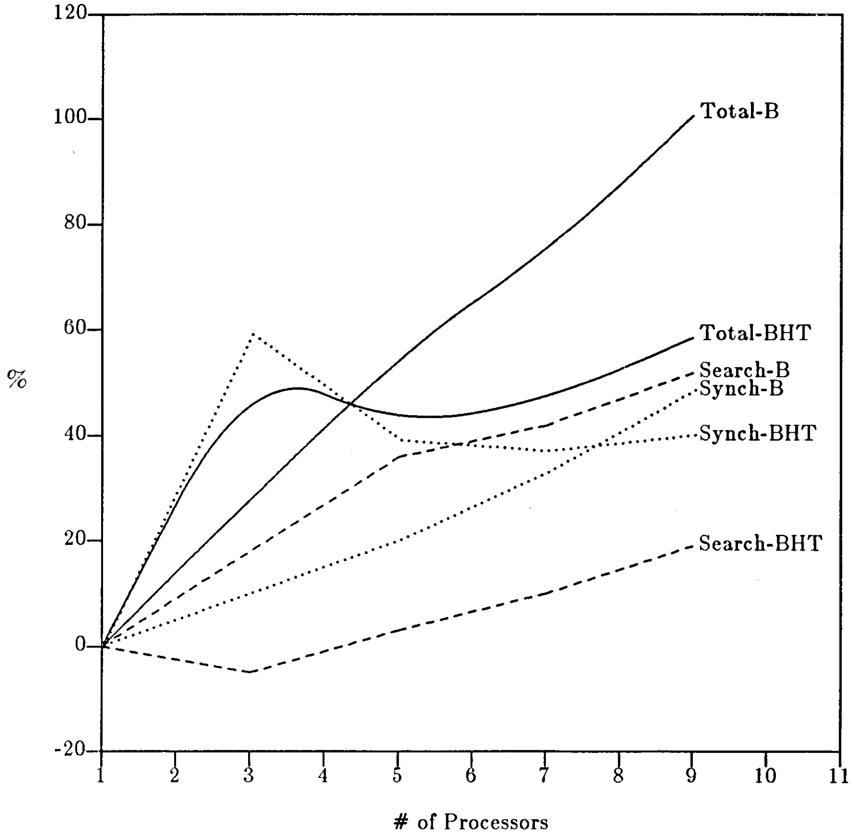


Figure 4. 7-Ply Overheads

made a significant improvement in the program's play.

At the time of this writing, Phoenix has competed in four tournament games with Minix. In only one game did Minix make a difference, and that on a move that did not alter the course of the game. In one practice game played, Minix prevented Phoenix from making a losing move. However, this data is insufficient to draw any conclusions.

6. Conclusions

Parallelizing tree search algorithms for a small number of processors is just a prelude to tackling the problems posed by having 100's or even 1000's of processors. Although no claims can yet be made on the suitability of re-assigning idle processors for reducing synchronization overhead, and globally accessible tables for decreasing search overhead, they both clearly addresses major problems posed by such a hardware configuration. Unfortunately the reduction of one overhead (synchronization for example) can result in the increase of others (communication and search). Improving on this remains an open research question. Given that existing parallelizations of alpha-beta are limited to a few processors, Scouts may provide an alternate means of using available resources profitably.

Acknowledgements

Many of the ideas used in the Scout processes evolved from discussions with Don Beal. The suggestions and help of Marius Olafsson is greatly appreciated.

References

1. G. Lindstrom, The Key Node Method: A Highly-Parallel Alpha-Beta Algorithm, UUCS 83-101, Department of Computer Science, University of Utah, 1983.
2. S.G. Akl, D.T. Barnard and R.J. Doran, The Design, Analysis, and Implementation of a Parallel Tree Search Algorithm, *IEEE Transactions on Pattern Analysis and Machine Intelligence 4*, 2 (1982), 192-203.
3. T.A. Marsland, M. Olafsson and J. Schaeffer, Multiprocessor Tree-Search Experiments, in *Advances in Computer Chess 4*, D. Beal (ed.), Pergamon Press, 1985, 37-51.
4. M. Newborn, A Parallel Search Chess Program, *ACM Annual Conference*, 1985, 272-277.
5. T.A. Marsland and F. Popowich, Parallel Game-Tree Search, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 4 (1985), 442-452.
6. D.J. Slate and L.R. Atkin, Chess 4.5 - The Northwestern University Chess Program, in *Chess Skill in Man and Machine*, P.W. Frey (ed.), Springer-Verlag, New York, 1977, 82-118.
7. J. Schaeffer, *Experiments in Search and Knowledge*, Ph.D. thesis, Department of Computer Science, University of Waterloo, 1986.
8. R.A. Finkel and J.P. Fishburn, Parallelism in Alpha-Beta Search, *Artificial Intelligence 19*, (1982), 89-106.
9. H.J. Berliner, *Chess as Problem Solving: The Development of a Tactics Analyzer*, Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, 1974.
10. G. Baudet, *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, 1978.
11. J. Fishburn, *Analysis of Speedup in Distributed Algorithms*, Ph.D. thesis, Computer Sciences Department, University of Wisconsin-Madison, 1981.
12. T.A. Marsland and M.S. Campbell, Parallel Search of Strongly Ordered Game Trees, *Computing Surveys 14*, (1982), 533-551.
13. M. Olafsson and T.A. Marsland, A Unix Based Virtual Tree Machine, *CIPS Congress 85*, Montreal, June 1985, 176-181.

14. K. Thompson and D.M. Ritchie, The UNIX Timesharing System, *Communications of the ACM 17*, 7 (1974), 365-375.

15. D. Kopec and I. Bratko, The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess, in *Advances in Computer Chess 3*, M.R.B. Clarke (ed.), Pergamon Press, 1982, 57-72.

NEW ADI MODEL PROBLEM APPLICATIONS

Nancy S. Ellner and Eugene L. Wachspress

Department of Mathematics
University of Tennessee
Knoxville, TN 37996-1300

Abstract

Peaceman and Rachford introduced ADI iteration to solve parabolic and elliptic linear systems. Recent generalizations enhance use of the model ADI problem as a preconditioner for solving nonseparable systems discretized by either five-point difference or nine-point finite element techniques. The theory also yields parameters for a new iterative procedure for solving the Lyapunov matrix equation. This application is well suited for parallel computation. Classical Chebyshev minimax theory is restricted to real approximation. Rouche's theorem provides a basis for generalization to the complex field. This is essential for application to systems involving matrices with complex eigenvalues. The particular application which motivated this generalization was computation of impedance boundary conditions for finite element computations by the method of "Infinitesimal Scaling" introduced by H. Hurwitz.

2. INTRODUCTION: THE ADI MODEL PROBLEM

Alternating-Direction-Implicit (ADI) model problem theory arises in diverse areas. It was developed by Zolotareff (1877) and applied by Cauer (1934) to an electrial filter design problem. In 1955 Peaceman and Rachford[9] introduced an alternating-direction-implicit (ADI) iteration for solving elliptic systems arising from five-point discretization of boundary-value problems governed by the differential equation $-\mathrm{div}[D(x,y)\mathrm{grad}u(x,y)] = f(x,y)$. The real, SPD (symmetric, positive-definite) coefficient matrix A is split into H and V where H is the discretization of the x-derivatives and V of the y-derivatives. The two-step iteration

$$(H+w_jI)u_{j-1/2} = -(V-w_jI)u_{j-1} + b$$

$$(V+w_jI)u_j = -(H-w_jI)u_{j-1/2} + b \qquad (1)$$

for $j = 1,2,\ldots,t$ and u_0 prescribed

is now known as Peaceman-Rachford iteration.

A rigorous theoretical analysis of convergence and methods for computing optimum w_j are known for the "model-problem" where H and V commute. In a series of papers published in the sixties, Chebyshev minimax theory was applied to establish existence and uniqueness of optimum parameters as a function of the eigenvalue spectra of H and V[14-18]. A convenient algorithm was devised for computing optimum parameters for the special case of $t = 2^n$. This algorithm is based on a folding technique having certain features not dissimilar to the FFT. W.B. Jordan[18] demonstrated how optimum parameters could be found for any t in terms of elliptic functions and gave simple formulas which yield values quite close to optimum. This solution in terms of elliptic functions had been discovered for the same minimax problem by Zolotareff and used by Cauer to design electrical filters.

Two significant generalizations of the Peaceman-Rachford iteration were introduced during the sixties. First, it was noted that one could replace w_jI by w_jF in Eqs. 1, where F is a diagonal matrix. The model-problem condition is then that $HF^{-1}V - VF^{-1}H = 0$. This is equivalent to a renormalization of H and V in Eq 1. A model-problem is obtained with variable mesh increments and $D(x,y) = 1$ when the difference equations are derived by the box-intergration method and matrix F has as its elements the mesh-box areas. This is true in both Cartesian and cylindrical coordinates[16] .

The second significant generalization was choice of a different value for the iteration parameter in the two sweeps. In the seminal work by Peaceman, Rachford and Douglas, it was noted that this can lead to divergence while convergence is assured with Eqs 1. The crucial point is that the two parameters for each double-sweep are related in a special way. This generalization

*This work was partially supported by General Electric Corporate Research and Development, Electromechanics Branch, Contract #P.O. G0076000.

yields optimum parameters as a function of the distinct spectral intervals of H and V (or of HF^{-1} and VF^{-1} when the parameters are multiplied by F.) This is sometimes a great improvement over parameters based on a single interval encompassing both spectra. Indeed, H or V may be singular or even have negative eigenvalues as long as A is positive definite. The analysis and examples were first given by Wachspress[15] and may now be found in many texts. Let the parameter w_j in Eqs. 1 be replaced by p_j in the first sweep and q_j in the second sweep. Let the spectral bounds be $a \leq s(H) \leq b$ and $c \leq t(V) \leq d$ with $a + c > 0$. The ADI minimax problem is then to find the parameters p_j and q_j to minimize

$$R(J) = \underset{\substack{a < s < b \\ c < t < d}}{\text{maximum}} \quad \prod_{j=1}^{J} \left| \frac{(q_j - s)(p_j - t)}{(p_j + s)(q_j + t)} \right| \qquad (2)$$

Model problems are not often encountered in practice. Nevertheless, model-problem ADI serves an important function as a preconditioner for compound iteration.

Two recent generalizations of ADI model-problem application were described by Wachspress.[18] First, one notes that the eigenfunctions of the diffusion equation are separable when the diffusion coefficient is a separable function of x and y . This generalization of the ADI model problem was described by Young[19]. The five-point difference equations derived by the box-integration method for such separable problems yield a model problem for ADI iteration. Theory relating to approximation of a nonseparable function by a separable one[2] is described by Light and Cheney[7]. Applying this to a non-separable diffusion coefficient leads to an ADI model problem as a preconditioner for a conjugate gradient or Chebyshev iteration which is often a great improvement over a Laplacian preconditioner. Schatz (in prep.) has used such generalized model problems with FPS and FFT preconditioners. Second, in solving finite element problems over rectangular grids one encounters nine-point rather than five-point equations, and no ADI model problem had been known for these nine-point equations. A transformation of variables was discovered that provided a nine-point ADI model problem. A third new application for ADI model-problem theory arises in an entirely different situation which will now be described.

3. THE LYAPUNOV MATRIX PROBLEM

3.1. The Problem and Some Solution Techniques

A recurring problem in linear algebra is the Lyapunov matrix problem where one is given the mxm matrix A , the nxn matrix B , the mxn matrix C and seeks the mxn matrix X for which

$$AX + XB = C \ . \qquad (3)$$

A unique solution is easily demonstrated for the case where s + q = 0 is not satisfied for any eigenvalue s of A and q of B . The ADI application arises when A and B are N-stable (i.e., the real parts of their eigenvalues are positive) and in the slightly more general case where the real part of s + q is positive for any eigenvalues s and q of A and B , respectively. A good survey of alternatives for solving Eq. 3 in use prior to 1972 is given by Rothschild Jameson[11]. Let N be the larger of m and n . Then the number of arithmetic operations for the four techniques they describe varies from $O(N^6)$ for the Geiss "brute-force" direct solution[3] to $O(N^4)$ for Jameson's iterative method[11]. For m = n , a more recent $O(N^4)$ direct method of Bartels and Stewart[1] with modifications introduced by Golub, Nash and vanLoan[4] appears more efficient and robust than the methods considered by Rothschild and Jameson. The bulk of the computation occurs in simultaneous reduction of A to triangular and B to upper Hessenberg form.

The new iterative method for solving the Lyapunov equation described in this paper requires $O(N^3)$ operations and is partucularly well suited for array computation. It is a generalized form of a procedure introduced by R.A. Smith in 1968[12]. Smith considered the case where A and B are N-stable. A concise summary of pertinent parts of Smith's analysis follows. Let E be the unit mxm matrix and I the unit nxn matrix. For any positive scalar q , the identity

$$(qE+A)X(qI+B) - (qE-A)X(qI-B) = 2qC \qquad (4)$$

is easily verified. Premultiply Eq. 4 by $(qE + A)^{-1}$ and postmultiply by $(qI + B)^{-1}$ to get

$$X - UXV = W \qquad (5)$$

where $U = (qE+A)^{-1}(qE-A)$, $V = (qI+B)^{-1}(qI-B)$,

and $W = 2q(qE+A)^{-1}C(qI+B)^{-1}$. By inspection,

$$X = \sum_{k=1}^{\infty} U^{k-1}WV^{k-1} \qquad (6)$$

is a formal solution to Eq. 5. The spectral radii of U and V are less than unity. It follows that the series in Eq. 6 is convergent. Define the matrix sequence Y_r by

$$Y_0 = W , \qquad Y_{r+1} = U^{2^r} Y_r V^{2^r} \ . \qquad (7)$$

It is seen that Y_r approaches X rapidly. Moreover, U^{2^r} and V^{2^r} are obtained by squaring $U^{2^{r-1}}$ and $V^{2^{r-1}}$, respectively. Each step thus requires four matrix multiplications and one matrix addition. This is a quadratically convergent iteration. Computation arithmetic is $\bar{O}(N^3)$.

3.2. A Generalization of R.A. Smith's Iteration

The generalization of Smith's iterative solution to be developed here has several advantages. The number of arithmetic operations remains of $O(N^3)$ but a parallelism not shared by Smith's algorithm is introduced for efficient use of an array computer. The restriction that both A and B be N-stable is relaxed to the condition that the sum of the real parts of eigenvalue p_j of A and eigenvalue q_j of B be positive for all i, j pairs. Fewer iterations are required with the generalization.

A crucial observation is that the identity in Eq. 4 may be generalized to

$$(pE+A)X(qI+B) - (qE-A)X(pI-B) = (p+q)C \ . \qquad (8)$$

Multiply Eq. 8 on the left by $(pE+A)^{-1}$ and on the right by $(qI+B)^{-1}$ to get

$$X - U(p,q)XV(p,q) = W(p,q) \qquad (9)$$

where

$$U(p,q) = (pE+A)^{-1}(A-qE) \qquad (10)$$

$$V(p,q) = (B-pI)(B+qI)^{-1} \qquad (11)$$

$$W(p,q) = (p+q)(pE+A)^{-1}C(qI+B)^{-1} \ . \qquad (12)$$

Then the Smith iteration in Eq. 7 yields

$$Y_0(p,q) = W(p,q) \ ,$$

$$Y_{r+1}(p,q) = U(p,q)^{2^r} Y_r(p,q)V(p,q)^{2^r} \ . \qquad (13)$$

Now note that Y_r is related to the solution X by

$$Y_r(p,q) = X - U(p,q)^{2^r} XV(p,q)^{2^r} \ . \qquad (14)$$

The procedure is convergent if the spectral radii of U and V are less than unity. It is obvious that one should choose p and q to minimize the spectral radii of $U(p,q)$ and $V(p,q)$. This is precisely the ADI minimax problem for $t = 1$.

The ADI model-problem was encountered in solving the linear system $Au = b$ when it was possible to split A into a sum of commuting matrices H and V. The case for which extensive theory was developed was for real spectra of H and V. The analysis is valid for any commuting linear operators H and V which sum to A. Prof. G.J. Habetler (R.P.I.) observed that one may define $H(X) = AX$ and $V(X) = XB$ to obtain $HV = VH = AXB$. It is no coincidence, therefore, that $AX + XB = C$ is a "model problem" for ADI iteration. It is seen that the ADI model-problem theory applies to this iterative solution of the Lyapunov matrix equation even when A and B do not commute! Moreover, the condition that $a + c > 0$ guarantees a unique solution. This is more general than the requirement that A and B both be N-stable.

3.3. An Algorithm for Parallel Computation

Smith's quadratic convergence is obtainable only with a constant p, q doublet. Moreover, two matrix inversions are required to obtain $U(p,q)$ and $V(p,q)$ for each doublet. For computation on a serial computer, it is questionable that one can improve efficiency with variable p and q. The situation is somewhat different in application to parallel processing. It will now be shown that one can iterate simultaneously with a different p,q couplet on each of several arithmetic units and then combine the results to derive some of the benefits of variable parameters.

Let $Y(r,j) = Y_r(p_j, q_j)$ as defined by Eq. 13. Parallel computation may be performed on J arithmetic units. The results may then be combined to yield the same estimate to X as would have been obtained by 2^{r-1} applications with optimum parameters for a cycle of length J. The combining algorithm is as follows:

$$X(r,1) = Y(r,1) \quad , \text{ and for } j > 1 \qquad (15)$$

$$X(r,j) = Y(r,j) + U(j)^{2^r} X(r,j-1)V(j)^{2^r} \ .$$

Each step in Eq. 15 requires four matrix multiplications and one addition. Although this is the same work as each step of the base Smith algorithm, this part of the calculation cannot be done in full parallel mode. One may replace Eq. 15 by a folding algorithm to retain some parallelism at this stage of the computation. The iterates in Eq. 15 satisfy

$$X(r,j) = X - \prod_{j'=1}^{j} U(j')^{2^r} \ X \prod_{j'=1}^{j} V(j')^{2^r} \ . \qquad (16)$$

This may be demonstrated inductively with the aid of Eq. 13. The value of $R(J)^2$ is an upper bound on the relative error $|X(r,J) - X| / |X|$, where $|\ |$ denotes the usual spectral norm.

3.4. Application To "Infinitesimal Scaling"

The Lyapunov matrix equation arises in a scheme devised by H. Hurwitz to compute an impedance boundary condition for finite element problems [6]. Hurwitz calls his method "infinitesimal scaling." An SPD matrix C is found as an impedance boundary condition by solving the nonlinear matrix equation

$$(C + Q)N^{-1}(C + Q^T) = M \qquad (17)$$

when given the real SPD matrices M and N and the real antisymmetric matrix Q .

Let $A_n = (C_n + Q)N^{-1}$. (18)

Then a quadratically convergent Newton-type iteration to solve this equation for C is

$$A_nC_{n+1} + C_{n+1}A_n = M - QN^{-1}Q^T + C_nN^{-1}C_n . \qquad (19)$$

This is the Lyapunov equation with $B = A^T$. Moreover, A is N-stable since its symmetric part is SPD.

Another property of the system is that Q is very small in norm compared to C . This yields a spectrum restricted to lie in a rectangle in the positive-real half-plane centered on the real axis and of height much smaller than its width. This is precisely the kind of spectra where ADI iteration can be expected to be very efficient. It was this problem which motivated generalization of the ADI model-problem theory to the complex domain.

4. ADI MODEL-PROBLEM THEORY FOR COMPLEX SPECTRA

4.1. Polynomial Approximation and Backward Spectrum Analysis

Chebyshev minimax theory for polynomial approximation over the reals has been extended to the complex domain. The alternating extremes property in the real case is replaced by constant absolute value of the error function on the spectrum boundary in the complex domain. Rouche's Theorem relating to the number of zeros and poles inside the boundary is then used to establish the minimax property.[9]

Analysis of the minimax problem for the eigenvalue domain $z = 1 + vi$ with v varying between -1 and 1 reveals some of the subtleties of the problem in the complex domain[5]. The transformation $w = i(1 - z)$ yields an interval of $[-1, 1]$ for w . One may examine the Chebyshev polynomial appropriate for this domain with normalization to unity at $z = 0$ or $w = i$. As observed by Opfer and Schober[9], this does not yeild the optimum polynomial in z . The Chebyshev theory establishing the de la Vallee property of alternating extremes of equal magnitude does not hold for complex-valued functions. However, as the degree of the approximation is increased, the Chebyshev polynomial attains its maximum absolute value on a curve that includes the eigenvalue spectrum and collapses in on this spectrum rapidly. As observed by Trefethen[13], this is an essential property of the optimum polynomial. Determining the optimum polynomial is not a simple task. Fortunately, in practice a prescribed error reduction is achieved only after the polynomial degree is sufficiently large that the Chebyshev polynomial is close to optimum. This theory also applies to the ADI rational approximation in which elliptic functions replace trigonometric functions in the analysis.

One of the more prevalent spectrum domains considered for Chebyshev polynomial approximation is an ellipse with major diameter along the real axis. The zeros of the Chebyshev polynomial are spread out along the real axis when the minor diameter is small compared to the major diameter. As the relative magnitude of the minor axis increases, the zeros coallesce toward the center. When the domain is a circle, the optimum polynomial has a single root at the center of multiplicity equal to the degree of the polynomial.

Rather than seeking optimum functions for a given spectrum, we choose a class of approximating functions and investigate the regions on which these functions are optimal. We consider the "lemniscates" described by Opfer and Schober[9]:

Definition: The G-lemniscate of $F(z)$ continuous on region W in the complex plane C is

$$L = \{z \in W: |F(z)| \le G\} \qquad (20)$$

Consider first polynomial approximation to zero over a complex region. Let u be a fixed value greater than unity at which a polynomial approximation to zero over the real interval $[-1, 1]$ of maximal degree t is normalized. The polynomial which has the least maximum absolute value over $[-1, 1]$ is the Chebyshev polynomial

$P_t(z) = \cos[t \arccos(z)]/\cos[t \arccos(u)]$.

Let $P_t(1) = H$. The H-lemniscate of P in the complex plane identifies an extended spectrum for which this polynomial is optimal. The application of Rouche's Theorem to the minimax problem over the lemniscate is as follows: P has absolute value of H on the H-lemniscate and t roots within the lemniscate. Let R be a "better" polynomial for approximation over this region in the sense that $|R| < H$ over the closed region bounded by the lemniscate. It is easily demonstrated that the winding number of $P - R$ is the same as that of P as one traces a path around the lemniscate. Hence, P and $P-R$ have the same number of roots in this region. P is known to have t roots within the lemniscate. Both P and R are normalized to unity at u outside this region. Therefore, $P - R$, a polynomial of degree at most t , has at least $t + 1$ roots. This is impossible. Hence, P is optimal.

The lemniscate oscillates between the degenerate ellipse which is the line segment $-1 \leq z \leq 1$ and the ellipse centered at the origin with semi-major axis endpoints $\pm$ cosh v and semi-minor axis endpoints $\pm i \sinh v$, where $v = \text{arcsinh}(1)/t$. Now consider the G-lemniscate for any G in the interval (H, 1). As G increases toward unity, the lemniscate expands to become a circle of radius u. The Chebyshev polynomial is optimal for this entire set of lemniscate spectra. Let $v^* = \text{arccosh}(G)/t$ and let $v' = \text{arcsinh}(G)/t$. The G-lemniscate oscillates between the ellipses

$$[\text{Re}(z)/\cosh v^*]^2 + [\text{Im}(z)/\sinh v^*]^2 = 1 \quad (21)$$

and

$$[\text{Re}(z)/\cosh v']^2 + [\text{Im}(z)/\sinh v']^2 = 1. \quad (22)$$

Let the ellipse of Eq. 21 be denoted by E(G). This ellipse is contained in the G-lemniscate. The Chebyshev polynomial has been used for many years to approximate zero over this elliptic spectral region[15]. Manteuffel[8] has shown that it is optimal. As t increases for a fixed eigenvalue spectrum, v^* and v' approach a common value. The G-lemniscate approaches the E(G)-ellipse. This is related to fundamental theorems developed by Trefethen[13] in a more general approximation theory setting.

This theory is applied to polynomial extrapolation in iterative solution of large systems of equations. Let the spectrum z of a base iteration be within the ellipse with major axis between a and b on the real axis (where $b < 1$) and semi-minor axis of length c. One then defines $v^* = \text{arctanh}[2c/(b-a)]$ and $z'(z) = (\cosh v^*)(2z - a - b)/(b - a)$. The optimal polynomial is then $P_t(z')$ with $u = z'(1)$. Let $\cos B = \tanh v^*$. Then the maximum absolute value of P over the ellipse is:

$$G = [\tan^t(B/2) + \tan^{-1}(B/2)]/\cosh[t \,\text{arccosh}(u/\sin B)]. \quad (23)$$

One of the problems arising in polynomial approximation is to find the ellipse containing a given spectrum and normalization point which minimizes G. This problem has been investigated by Manteuffel[8] and others.

4.2. The ADI Minimax Problem With Complex Spectra.

ADI theory has been generalized to complex spectra. Now, the appropriate domain for analysis is still an ellipse but for the logarithm of the spectrum rather than the spectrum itself. The spectrum is first normalized so that the ellipse is centered at the origin. (If the real part of the spectrum varies from a to b, the spectrum is divided by the square root of ab to yield this ellipse. The parameters determined for this problem are then multiplied by the square root of ab. Minimax analysis is invariant to multiplicative but not additive normalization.) Just as for polynomial approximation, as the ratio of the minor to major axis of the ellipse increases, the optimum parameters coallesce toward the origin which corresponds to a normalized eigenvalue of unity. When the spectra of H and V are real, ADI convergence is greatly enhanced by use of variable p and q. This is especially true when H and V have very large condition numbers (more precisely, when the eigenvalue bounds b and d are much greater than a + c.) Just as in polynomial approximation, this enhancement through use of variable parameters diminishes as the imaginary components of the spectrum increase.

W.B. Jordan's bilinear transformation[15] reduces the two-variable ADI minimax problem of Eq. 2 to a problem in one variable (say x) over the real interval [k', 1] with k' determined from the eigenvalues a, b, c and d. When $a + c > 0$, k' is in (0, 1). The variable u is defined by $x = \text{dn}(Ku, k)$, where dn is the elliptic function which varies from 1 to k' as u varies from 0 to 1. The optimum rational function for this problem is

$$Q_t(x) = k_2^{1/2} \text{sn}[(2tu+1)K_2, k_2] \quad (24)$$

where k_2 is a function of t and k'. In most applications, k' is close to zero. Choice of t sufficiently large to attain a significant error reduction yields a value of k_2 close to zero also. The error reduction after t ADI iterations is equal to k_2 (the square of maximum absolute value of Q_t in the interval [k', 1].)

Just as for the polynomial approximation previously discussed, one may generalize to the complex plane by considering the family of $G\sqrt{k_2}$-lemniscates for the function $Q_t(z) = \sqrt{k_2}\,\text{sn}[(2t(u+iv)+1)K_2, k_2]$. A value of $G = 1$ yields a lemniscate which cuts the real axis at the extremes which are the alternating points for the real problem. A $G\sqrt{k_2}$ - lemniscate for $1 < G < 1/\sqrt{k_2}$ and $0 \leq u \leq 1$ identifies an extended spectrum for which the $Q_t(z)$ function is the optimal rational polynomial of that form by an application of Rouche's theorem. The function $Q_t(z)$ has absolute value $G\sqrt{k_2}$ on its $G\sqrt{k_2}$ - lemniscate and is a rational function of the form $q_t(-z)/q_t(z)$. Suppose there exists a better approximation of this form to zero over the lemniscate region. The difference between these two functions is a ratio of polynomials of maximal degree 2t-1 which has t roots between k' and 1, and (by symmetry) t roots which are the negatives of these as well. This difference function also vanishes at the origin, which is a contradiction since this function of maximal degree 2t-1 then has 2t+1 roots.

As in the case of Chebyshev extrapolation, as the degree of approximation increases for a fixed eigenvalue spectrum, the $G\sqrt{k_2}$-lemniscate of $Q_t(z)$ collapses rapidly on its largest interior convex region, which contains the spectrum. However, the largest convex region here is not elliptical. It becomes approximately elliptical under the 'log-symmetric' transformation $z' = \ln(z/\sqrt{k'})$. The approximating ellipse has semiminor axis endpoints $\pm bi = iv^*\ln(4/k')$ and semimajor axis endpoints $\pm a = \ln[\sqrt{k'} \cos b]$. Then for efficient application of ADI iteration, the log-symmetric transformed spectrum must be embedded in an ellipse with the least value of effective spectral radius G^2k_2 of the iteration. The effective spectral radius of the ADI iteration is bounded by $\exp(\pi^2/\ln(k'/4))$ when the spectrum is real[16] and by $\exp(\pi^2/\ln(k'/4) + 2\pi v^*)$ when the spectrum is complex.

4.3. Numerical Results

Suppose the spectrum lies in a rectangle such that the real part of the spectrum lies between a and b , and the imaginary part lies between -c and c , where $0 < a << 1$ and $c < a << b - a$. Then optimum ADI iteration parameters will be obtained from the approximating ellipse (section 4.2) with maximum possible value of $(\pi/\ln(4/k'))-2v^*$ of all those ellipses containing the log-symmetric transformed spectrum. This will occur for some ellipse with boundary containing at least one of the left or right sets of corners of the log-symmetric transformed rectangle. This is a constrained maximization problem in the one variable $w = \ln(4/k')v^*$ which can be shown to have a solution for spectra with a, b, and c as described above. Numerical trials suggest that the solution is unique.

The Lyapunov problem

$$A^TX+XA = W \qquad (25)$$

was solved according to the complex-spectrum ADI method outlined above, and by the ADI method assuming a real spectrum between a and b for comparison. The matrix A was chosen to be a 20X20 block diagonal matrix with eigenvalues equally spaced along the upper and lower sides of

Table 1: Numerical results for two test matrices A using complex-spectrum method parameters and real-spectrum method parameters. Errors measured are $||X_n-X||/||X||$ with the Eucliden norm, where X_n is the nth iterate.

	Spectrum Bounds			Spectrum Bounds		
	a = .01	b = 100	c = .0010	a = .01	b = 100	c = .0019
number of iterations	Complex-spectrum Method		Real-spectrum Method	Complex-Spectrum Method		Real-spectrum Method
	k' = .000045		k' = .0001	k' = .00003		k' = .0001
	v* = .02155 (radians)		v* = 0	v* = .03207 (radians)		v* = 0
2	.7073	Theor. error bound	.6210	.7515	Theor. error bound	.6210
	.2410	Actual error	.2349	.2544	Actual error	.2350
4	.1251	Theor.	.9640 x 10^{-1}	.1412	Theor.	.9640 x 10^{-1}
	.1519 x 10^{-1}	Actual.	.3150 x 10^{-1}	.1992 x 10^{-1}	Actual.	.3208 x 10^{-1}
6	.2212 x 10^{-1}	Theor.	.1496 x 10^{-1}	.2653 x 10^{-1}	Theor.	.1497 x 10^{-1}
	.3946 x 10^{-2}	Actual	.4749 x 10^{-2}	.8628 x 10^{-2}	Actual	.5370 x 10^{-2}
8	.3911 x 10^{-2}	Theor.	.2323 x 10^{-2}	.4982 x 10^{-2}	Theor.	.2323 x 10^{-2}
	.1430 x 10^{-2}	Actual	.8843 x 10^{-3}	.1223 x 10^{-2}	Actual.	.1207 x 10^{-2}
10	.6915 x 10^{-3}	Theor.	.3606 x 10^{-3}	.9360 c 10^{-3}	Theor.	.3606 x 10^{-3}
	.1294 x 10^{-3}	Actual	.1655 x 10-3	.1363 x 10^{-3}	Actual.	.2962 x 10^{-3}
12	.1223 x 10^{-3}	Theor.	.5599 x 10^{-4}	.1758 x 10^{-3}	Theor.	.5599 x 10^{-4}
	.2417 x 10^{-4}	Actual	.3403 x 10^{-4}	.8174 x 10^{-4}	Actual	.7872 x 10^{-4}
14	.2162 x 10^{-4}	Theor.	.8691 x 10^{-5}	.3304 x 10^{-4}	Theor.	.8691 x 10^{-5}
	.5017 x 10^{-5}	Actual.	.7723 x 10^{-5}	.1109 x 10^{-4}	Actual.	.2120 x 10^{-4}
16	.3824 x 10^{-5}	Theor.	.1393 x 10^{-5}	.6206 x 10^{-5}	Theor.	.1349 x 10^{-5}
	.1683 x 10^{-5}	Actual	.1739 x 10^{-5}	.2213 x 10^{-5}	Actual.	.5652 x 10^{-5}

the rectangular spectrum boundary. The results for two different matrices with the same real spectral ranges are given in Table 1. The results of the iterations for the real-spectrum and complex-spectrum ADI parameters were very comparable. However, there is generally a slight improvement when using the complex-spectrum method parameters. Also, the associated theoretical error bounds are more accurate at higher numbers of iterations and larger imaginary eigenvalue components. Errors nearly identical to those in Table 1 were obtained from matrices with four eigenvalues defining a rectangular spectral bound and the remaining eigenvalues distributed randomly within the rectangle. This suggests that, as expected, the corner eigenvalues are determining the actual convergence rate.

An analagous method can be used if the spectrum is known to lie in a convex polygonal region within a rectangle, defined by the real and complex spectral ranges, which satisfies the conditions on a, b, and c stated above. In this case, the tighter bounds on the spectrum should give more rapid convergence. Numerical studies of this case are in progress.

REFERENCES

[1] R.H. Bartels and G.W. Stewart, "A Solution of the Matrix Equation AX+XB = C ," Comm. ACM 15, pp. 820-826, 1972.

[2] S.P. Diliberto and E.G. Straus, "On the Approximation of a Function of Several Variables by the Sum of Functions of Fewer Variables," Pacific J. Math. 1, pp. 195-210, 1951.

[3] G.R. Geiss, V. Cohen, and D. Rothschild, Grumman Res. Dept. Report E-307, 1967.

[4] G.H. Golub, S. Nash, and C. Van Loan, "A Hessenberg-Shur method for solution of the problem AX+XB = C," IEEE Trans. Automat. Control AC24, pp. 909-913, 1979.

[5] L.A. Hageman and D.M. Young, *Applied Iterative Methods*, Academic Press, 1981.

[6] H. Hurwitz, "In finitesimal Scaling - A New Procedure for Modeling Exterior Field Problems," IEEE Trans. Magn. 20, No. 5, pp. 1918-1923, 1984.

[7] W.A. Light and W. Cheney, "Approximation Theory in Tensor Product Spaces," Springer Lecture Notes in Mathematics 1169, 1985.

[8] T.A. Manteuffel, "The Tschebychev iteration for nonsymmetric linear systems," Numer. Math. 28, pp. 307-327, 1977.

[9] G. Opfer and G. Schober, "Richardson's Iteration for Nonsymmetric Matrices," Linear Algebra and its Applications 58, pp. 343-361, 1984.

[10] D.W. Peaceman and H.H. Rachford, Jr., "The Numerical Solution of Parabolic and Elliptic Differential Equations," J. Soc. Indust. Appl. Math. 3, pp. 28-41, 1955.

[11] D. Rothschild and A. Jameson, "Comparison of four numerical algorithms for solving the Liapunov matrix equation," Int. J. Control 11, No. 2, pp. 181-198, 1970.

[12] R.A. Smith, "Matrix equation XA+BX = C," SIAM J. Appl. Math. 16, No. 1, pp. 198-201, 1968.

[13] L.S. Trefethen, "Near-Circularity of the Error Curve in Complex Chebyshev Approximation," J. Approx. Theory 31, pp. 344-367, 1981.

[14] E.L. Wachspress, "Optimum ADI Iteration Parameters for a Model Problem," J. Soc. Indust. Appl. Math. 10, pp. 339-350, 1962.

[15] ______, "Extended Application of Alternating Direction-Implicit Iteration Model-Problem Theory," J. Soc. Indust. Appl. Math. 11, No. 4, pp. 994-1016, 1963.

[16] ______, *Iterative Solution of Elliptic Systems*, Prentice Hall, 1966.

[17] ______, "Two-Variable ADI Minimax Problem", Ph.D. dissertation, Rensselaer Polytechnic Institute, 1968.

[18] ______, "Generalized ADI Preconditioning," CAMWA 10, No. 6, pp. 457-461, 1984.

[19] D.M. Young, *Iterative Solution of Large Linear Systems*, Academic Press, 1971.

FINITE ELEMENT ANALYSIS USING ADVANCED PROCESSORS

Graham F. Carey and E. Barragy

The University of Texas at Austin
Austin, Texas 78712

Abstract

The evolution of advanced scientific processors is briefly summarized and the impact of the microelectronics revolution on vector and parallel scientific computation is examined. These radical changes in computer architectures are having a profound influence on numerical algorithm development and the implementation of numerical approximation techniques. We consider specifically the finite element method for solution of boundary-value and initial-value problems in engineering and science. The main steps of mesh generation, solution, and post-processing in a finite element analysis are outlined for implementation in a concurrent processing environment. We also give a specific example for parallel processing with a residual-based iterative method on a partition of subdomains.

Advanced Processors

Since the earliest attempt by Babbage in 1822 to develop a mechanical 'analytical engine,' the need to solve scientific and engineering problems has naturally been a dominant factor in the evolution of computers and calculating methods. However, modern scientific computation dates from the introduction of electronic computers, such as the ENIAC in 1946. Phenomenal advances in electronics since that time have led to the emergence of succeeding generations of advanced computers such as the IBM 7090, ILLIAC, CDC 6600 and CRAY 2. Although the term 'supercomputer' has been applied only recently to the current generation vector computers, the term generally describes the leading scientific computers of a given generation. Thus, today's supercomputer will, in due course, be superceded by the advanced processors of the next generation. The revolution in microelectronics has been at the center of technological advances in computing. Large-scale integrated circuits on a tiny chip now have more computing power than the mainframe computers of the 1960's. These advances are all the more impressive when we consider the short period over which they have occurred. One is tempted to imagine that this development can continue unabated at the same rate and in the same manner through miniaturization. However, at the present sub-micron level at which chip research is being conducted, the distances are so small that fabrication difficulties are formidable. Moreover, the speed of light becomes a very real constraint at this scale. The prevailing wisdom is that a major scientific breakthrough using, perhaps, ideas related to "quantum tunneling" will be necessary for the current rate of progress in chip design to be maintained.

There is another aspect of the microelectronics industry that has a very direct bearing on current trends in advanced computers. This is the fact that computer chips can be manufactured in large quantities at very low cost. As a result, the familiar desk-top personal computer has more computing power than an IBM 7090 at a minute fraction of the cost. In fact, a processor for the recently announced ETA 10 supercomputer would fit on a desk top. The mass production of chips has led not only to processors for desk-top computers but also to the development of systems in which processors may be used concurrently in parallel, either independently or jointly on a given task. The ideas of exploiting parallelism are not new, and various forms of parallelism exist to differing degrees in all computers at the arithmetic and storage levels. Historically, parallelism at the processor level was an integral part of the early ILLIAC design but was obviated by progress in microelectronics. The difficulties confronting smaller chip design and the economy of mass production of chips has led to a renaissance in parallel processing.

Current generation supercomputers are configured with a few vector processors in parallel. Examples are the CRAY XMP (2 or 4 processors), CRAY 2 (8 processors) and ETA 10 (8 processors). Much of the scientific calculation performed using these computers is still in serial mode with the processors used independently on different problems to increase throughput. Opportunities to multitask (or microtask) a single large-scale problem exist, and this is a topic of current study. Rather than use a few very sophisticated processors, one can also consider linking many simpler processors into a highly parallel distributed network. Each individual processor costs little but is far slower than the parallel supercomputers mentioned above. The essential idea is to partition the calculation among many processors uniformly to achieve high computation speeds.
The hypercube design has been developed as an

CH2345-7/86/0000/0535$01.00 © 1986 IEEE

effective means of linking processors in a parallel computing system. For example, the INTEL IPSC Hypercube became available in 1985, and is presently configured in 32, 64 and 128 processor versions. In April 1986, the FPS-T series appeared, also configured using the hypercube concept, but involving more advanced vector processing units. Other parallel computers, such as the Balance 8000, are also in research use.

The introduction of vector processors and now of parallel processing has led us to re-examine numerical methods and algorithms that were previously developed and refined for sequential scalar calculations. Features of a numerical method and the associated algorithm, that proved inefficient for sequential scalar calculations, may now be quite attractive (and vice versa; see, e.g., Fox and Otto [1984], Voigt [1985]). In the next section, we examine this issue further in the context of finite element methods and algorithms.

Finite Elements

The finite element method grew out of the need to solve accurately structural analysis problems in the aerospace industry during the late 1950's. Although the mathematical ideas had been outlined earlier in the appendix of a paper by Courant [1943], it was not until advanced electronic computers were widely available that the method could be effectively applied for routine engineering computations. Since this period, finite element techniques have evolved rapidly and are now established as a basic method for solving boundary and evolution problems in science and engineering. In a finite element analysis, the problem is formulated in a weighted integral sense and a piecewise-polynomial approximation sought on a discretization of the domain composed of 'elements' of simple shapes. Contributions of each element to the associated discrete system are computed independently and the resulting numerical linear algebraic problem solved using sparse matrix methods.

Point iterative methods were used initially for finite element system solution due to the limited fast memory available. Since engineering design problems in structural analysis frequently involve many right-hand side vectors, factorization of the system matrix followed by repeated substitution sweeps was preferable. This has become the prevalent approach for system solution and is used widely with such sparse solution schemes as simple banded solvers, envelope (or profile) solvers and frontal solvers. For problems with few right-hand sides, iterative techniques may be a viable alternative, particularly when very large-scale problems are considered and memory limitations again become a significant constraint. On the other hand, iterative methods are most effective when the matrix is symmetric positive definite. While iterative methods may also be successful for other matrices, special preconditioning strategies are frequently required; in certain important classes of problems including, for instance, convection-dominated transport the methods fail or may not be practical.

In finite element analysis of time-dependent problems using implicit integration or of nonlinear problems, sparse linear system solution is still required, and the above considerations again apply. The availability of very large fast memory on the CRAY 2 permits sparse elimination solution of very large three-dimensional problems, but this is generally not feasible with other existing single or parallel systems.

There are several levels at which one can identify parallelism, ranging from concurrent physical processes through the choice of mathematical model to the selected algorithms for numerical solution. These ideas and several strategies for concurrent processing are discussed in a previous study (Carey [1986]) and detailed studies presented by several contributors to the same special journal issue on "PDE's and Algorithms on Advanced Processors." (See, for instance, the articles by Seager [1986] on conjugate gradient multitasking, Rodrigue [1986] on subdomain splitting, Adams [1986] on multicolor iterative methods, Kincaid et al. [1986] on vectorized iterative methods and McBryan and Van de Velde [1985] on elliptic methods, among others. The earlier ASME monograph, edited by Noor [1983], gives a broad overview of other developments in this area.

In the present study, we will focus specifically on the inherent structure of the finite element method as derived from the integral statement of the problem and the independence of element calculations. This philosophy can also be successfully incorporated in the solution procedure, by re-designing the solution algorithms appropriately.

Let us consider the three main steps in a finite element scheme:

(i) Preprocessor - Here, we define the domain, problem data, control variables and then generate the mesh. The significant computation is the mesh generation, and various strategies may be used to this end. These range from interpolatory mapping using the element basis to solution of Laplace-type problems for the nodal coordinates. Since the latter procedure can itself be posed as a finite element problem on a rectangular domain, the arguments in (ii) for vector and parallel calculations also apply to this subsidiary problem.

An effective procedure that lends itself to concurrent computing is the block method of generating a mesh: The global domain is partitioned to a number of major subdomains 1, 2, ..., NS where NS is related to the number of processors NP available (e.g., NS = m NP , integer m). The partition topology of the mesh is resident in each processor (or in a control processor). Subdomain meshes are generated by each respective processor to yield the desired nodal coordinate data and element data. Across the interface between adjacent blocks, shared nodal point information and element connectivity information is "exchanged." Hence, communication is proportional to the number of interfaces in the partition and processing time to the density of the mesh in each subdomain, as well as the number of subdomains NS and number of processors available NP .

(ii) Processor - In the processor, the main element computations and system solution are carried out. Recall that in standard practice the element matrix and vector contributions are calculated sequentially for each element and "assembled" to the global finite element algebraic system. Since the element calculations are independent, they may be made concurrently in parallel. There are two main choices. First, elements can be processed NP at a time until the element list is exhausted. (The elements are dealt out to the processors in any order.) Alternatively, we can retain the subdomain format and again assign processors to specific subdomains: Processor 1 computes all element contributions associated with subdomain 1, and so on.

The assembly step must also be considered. If each processor has sufficient local memory, then the element contributions can be assembled locally, independently and concurrently. Following this, global communication across the interfaces between subdomains can be introduced to complete the assembly of the interface equations. This yields a global sparse matrix problem which can be partitioned in various ways for efficient parallel elimination or iterative solution.

Rather than assemble to the global system and solve as indicated, one can instead retain the original subdomain-processor format. Using, for instance, the idea of substructure analysis, we can pre-eleminate the interior unknowns in each subdomain to construct, in essence, subdomain "super-elements." These super-elements can be assembled in the standard way to give a reduced system corresponding to the unknowns on the interface. Assembly again requires exchange of information at the common interfaces and accumulation of nodal equations, with some overlap of communication and computation.

Iterative methods can also be combined with the above strategy as, for instance, in block iteration for each of the subdomain blocks. Here, all the internal degrees of freedom are retained in each block matrix. For a residual-based iterative method, the block matrix operations can be made concurrently, with exchange of information concerning residual vectors at the interfaces only. We discuss this scheme in more detail later.

(iii) Post-Processing - The finite element vector of nodal unknowns (displacements, potentials, temperature, etc.) may be used to determine subsidiary results (stresses, velocities, fluxes, respectively) of engineering interest. Furthermore, in adaptive refinement procedures, the finite element solution on a given mesh is post-processed to compute the error indicator of each element for the refinement algorithm (see, e.g., Carey and Oden [1984]). Both the indicator calculation and the refinement procedure can be carried out by different processors on separate subdomains. However, some caution must be exercised again at the interfaces to ensure compatible refinements here between adjacent subdomains and the problem complexity is significantly increased. Energy integrals on the entire domain and similar global scalar quantities must be accumulated and, hence, requires communication among processors.

Summarizing, one variant of the parallel finite element scheme proceeds as follows:

(1) The domain is partitioned to major blocks;
(2) Mesh generation is carried out concurrently on blocks (subdomains);
(3) Element matrix calculations are carried out sequentially within blocks but in parallel between blocks;
(4) (a) Pre-eliminate internally within blocks, concurrently, or
    (b) Block matrix-vector products for iteration concurrently;
(5) (a) Solve reduced system by parallel-vector elimination, or
    (b) Update residuals at interfaces during iteration;
(6) Post-process within blocks for element derivatives, integrals, error indicators, etc.

Remarks: (1) For "massively parallel" architectures, the subdomain size will be relatively small and, hence, the amount of computation on each processor proportionally less. For some of the steps above, e.g., (3), (4) and (6), little or no processor communication is involved. Step (5), however, requires exchange of data between processors at the block interface and communication considerations become more significant (see also Gannon and von Rosendale [1984).

(2) As indicated previously, the scheme above can be extended to treat nonlinear problems and time-dependent problems. For instance, in a successive approximation or Newton scheme, the coefficient matrix and right-side vector of a linear algebraic system are recomputed element by element within subdomains in each step of the nonlinear iteration. Hence, steps (3)-(5) above are then simply embedded within the outer nonlinear iteration. In the time-dependent case, if an

implicit scheme (e.g., Crank-Nicolson) is used, we again have a linear (or nonlinear) system to solve at each time step, so steps (3)-(5) are carried out in each time step. Adaptive stepsize control based on, say, truncation error estimates will require minimal communication between processors. Explicit time integration is, of course, simpler to implement and vectorize within subdomains. Using differing time steps in different subdomains increases the complexity and communication an order of magnitude.

Example: Subdomain Block Iteration. As an illustrative example, we consider the use of a block iterative solution scheme 4(b) and 5(b) for system solution. Let the domain be partitioned into NS = NP subdomains where NP is the number of processors. The subdomain and processor interconnection topology are common. Let $e_1, e_2, \ldots, e_{NS}$ be the number of elements generated in each subdomain.

The main calculation in a residual-based iterative solution (e.g., conjugate gradient solution) of the linear system

$$\utilde{A}\utilde{u} = \utilde{b} \qquad (1)$$

is the matrix vector product in the residual calculation for iterate k

$$\utilde{r}_k = \utilde{A}\utilde{u}_k - \utilde{b} \qquad (2)$$

Let us consider the product $\utilde{A}\utilde{v}$ for $\utilde{v} = \utilde{u}_k$ with our block partition. We can write

$$\utilde{A}\utilde{v} = \left(\sum_{s=1}^{NS} \hat{\utilde{A}}_s\right)\utilde{v} = \sum_{s=1}^{NS} \left(\hat{\utilde{A}}_s \hat{v}_s\right) \qquad (3)$$

where $\hat{\utilde{A}}_s$ is the "expanded" matrix obtained by accumulating all element contributions to subdomain s with all other entries of $\hat{A}_s$ (corresponding to the other blocks) zero. Similarly, $\hat{\utilde{v}}_s$ is an "expanded" vector of nodal values associated with the nodes of block s and with all other components zero. Clearly, the product in (3) can be obtained by computing only the block matrix product $\bar{\utilde{A}}_s \bar{\utilde{v}}_s$ where $\bar{\utilde{A}}_s, \bar{\utilde{v}}_s$ are the restrictions of $\hat{\utilde{A}}_s, \hat{\utilde{v}}_s$ to block s .

Hence, the main steps in the algorithm involve:

(1) the sequential calculation of the element contributions $\utilde{A}_e$, $\utilde{b}_e$ for element e = 1, 2, ..., e_s with accumulation to $\bar{\utilde{A}}_s$, $\bar{\utilde{b}}_s$ in each block s ;

(2) the sparse vectorized block matrix products $\bar{\utilde{A}}_s \bar{\utilde{v}}_s = \bar{\utilde{w}}_s$ in each block s to obtain the block residual $\bar{\utilde{r}}_s = \bar{\utilde{b}}_s - \bar{\utilde{A}}_s \bar{\utilde{v}}_s$;

(3) accumulation of the residual component r_{is} for nodes i on the interface of subdomain s , requiring communication between "adjacent" processors. (Fetch and add r_{is} from subregions s adjacent to node i , $r_i = \sum r_{is}$, then send r_i back to r_{is} for adjacent subregions);

(4) update solution vector $\utilde{u}_s$ on each subdomain s , accumulate solution components u_{is} for nodes i on the interface of subdomain s and send the revised interface values back to the subdomains;

(5) Repeat steps (2)-(4).

Remark: Several variants of this scheme are possible, including, in particular, element-by-element accumulation of residuals within each subdomain using the strategy of Carey and Jiang [1986]. This necessitates slightly more computation on each processor, but storage is now negligible so the approach is suitable for architectures with small, fast local memory.

Acknowledgements

We express our appreciation to M. Seager and B. N. Jiang for related discussions.

References

1. Adams, L., "Reordering Computations for Parallel Execution," J. Comm. Appl. Num. Meth., pp. 263-272, 2, 3, 1986.

2. Carey, G. F., "Parallelism in Finite Element Modelling," J. Comm. Appl. Num. Meth., pp. 281-288, 2, 3, 1986.

3. Carey, G. F. and B. N. Jiang, "Element-by-Element Linear and Nonlinear Solution Schemes," J. Comm. Appl. Num. Meth., pp. 145-153, 2, 2, 1986.

4. Carey, G. F. and J. T. Oden, Finite Elements: Computational Aspects. New York: Prentice-Hall, 1984.

5. Courant, R., "Variational Methods for the Solution of Problems of Equilibrium and Vibrations, Bull. Am. Math. Soc., pp. 1-23, 49, 1943.

6. Fox, G. C. and S. W. Otto, "Algorithms for Concurrent Processors," Phys. Today, pp. 50-59, 37, 5, 1984.

7. Gannon, D. B. and J. von Rosendale, "On the Impact of Communication Complexity in the Design of Parallel Numerical Algorithms, IEEE Trans. Comp, pp. 1180-1194, C-33, 1984.

8. Hageman, L. and D. Young, Applied Iterative Methods. New York: Academic Press, 1981.

9. Kincaid, D. R., T. C. Oppe and D. M. Young, "Vectorized Iterative Methods for Partial Differential Equations," J. Comm. Appl. Num. Meth., pp. 289-296, 2, 3, 1986.

10. McBryan, O. and E. Van de Velde, "Parallel Algorithms for Elliptic Equations," Comm. Pure Appl. Math., pp. 769-795, 28, 1985.

11. Rodrigue, G., "Some Ideas for Decomposing the Domain of Elliptic Partial Differential Equations in the Schwarz Process," J. Comm. Appl. Num. Meth., pp. 245-250, 3, 4, 1986.

12. Noor, A. K. (Ed.), "Impact of New Computing Systems on Computational Mechanics," ASME Monograph, 1983.

13. Seager, M., "Overhead Considerations for Parallelizing Conjugate Gradient," J. Comm. Appl. Num. Meth., pp. 273-280, 2, 3, 1986.

14. Voigt, R. G., "Where Are the Parallel Algorithms," ICASE Rept., 85-2, NASA Langley, 1985.

PARALLELISM IN SOLVING PDES

John R. Rice*

Department of Computer Science, Purdue University
West Lafayette, IN, 47907

ABSTRACT

This paper examines the potential of parallel computation methods for partial differential equations (PDEs). We first observe that linear algebra does not give the best data structures for exploiting parallelism in solving PDEs, the data structures should be based on the physical geometry. There is a naturally high level of parallelism in the physical world to be exploited and we show there is a natural level of granularity or degree of parallelism which depends on the accuracy needed and the complexity of the PDE problem. We discuss the inherent complexity of parallel methods and parallel machines and conclude that dramatically increased software support is needed for the general scientific and engineering community to exploit the power of highly parallel machines.

I. INTRODUCTION AND SUMMARY

This paper examines the potential for the use of parallelism in the solution of partial differential equations (PDEs). There are six principal points made as follows:

1. Linear algebra is not the right model for developing methods for PDEs and it is particularly inappropriate for parallel methods.
2. The best data structures for PDE methods are based on the physical geometry of the problem.
3. Physical phenomena have large components that inherently parallel, local and asynchronous. Parallel methods can be found to reflect and exploit this fact.
4. There is a natural granualarity associated with parallel methods for PDEs. The best number of "pieces" and processors depends on the complexity of the physical problem, the accuracy desired and properties of the iteration used.
5. Parallel machines are very messy and it is essential for most users that one have very high level PDE systems to hide this mess.
6. There is much to be gained to using regularity in parallel methods, but one should not carry this to extremes.

II. LINEAR ALGEBRA DOES NOT GIVE THE BEST DATA STRUCTURES

In recent years there have been numerous papers written about linear algebra on parallel/vector machines (see Hwang[2], Sameh[5] and Ortega and Voight[3] for surveys and further references). Many machines have been designed to provide very high performance for linear algebra computations (see Hwang and Briggs[1] and Hwang[2] for surveys and further references). Most of this work is motivated or justified in some part by applications to solving PDEs. Solving large linear problems is an inherent step in solving PDEs and it is usually the most expensive step, yet the thesis of this section is that most linear algebra approaches can be misleading for exploiting parallelism in solving PDEs.

A case in point is *nested dissection*. This was a breakthrough in solving PDEs, one that many people (including myself) had searched for over a period of decades. The original presentation by Alan George of nested dissection was inscrutable. If one starts (as everyone did) with the linear algebra problem $Ax = b$, then to discover nested dissection, one had to see that a matrix rearrangement such as shown in Figure 1 was the "right" way to eliminate the unknowns. However, if one expresses the reordering in terms of the underlying geometry of the PDE, one sees that nested discretion is a natural divide and conquer algorithm. It is then easy to understand why the method works so well, to see how to extend it to nonrectangular domains or to 3 dimensions or to finite element methods.

If one starts with a conventional matrix/vector representation of a PDE computation it is much harder to find efficient methods because the inherent structure of the PDE problem is so distorted by conventional matrix/vector representations. This is further illustrated in Figure 3 which shows the conventional matrix structure obtained by discretizing a second order PDE with derivative boundary conditions on the domain shown there. It is a computational tour-de-force to recover from Figure 3 the information that is superficially apparent in the domain picture.

* This work supported in part by Air Force Office of Scientific Research grant AFOSR-84-0385.

The shortcomming of the conventional linear algebra approach is that the right data structure is not used, instead one should base the data structure on the underlying physical geometry. Figure 4 shows a domain which has been "exploded" to group "like-kinds" of elements together in a PDE problem. A method that is really successful in exploiting parallelism in this problem must "know" this structure, the most practical way to know it is to have it given explicitly in the data structure. More complex problems have other structure (interfaces, singular points, etc.) that can be incorporated in a similar way. It is not just parallelism in the computation that needs information such as seen in Figure 4, the control of numerical methods also need it. Numerical models need to be more accurate (e.g., grids need refining) near special locations. The partitioning of the computations for rapid convergence in iterative methods is strongly influenced by this information.

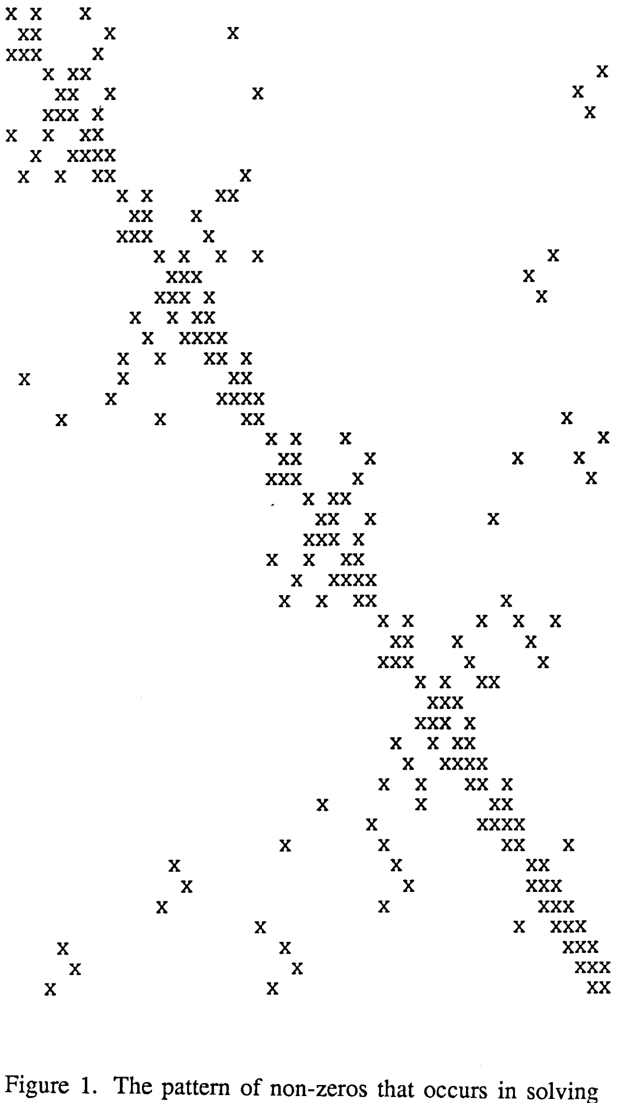


Figure 1. The pattern of non-zeros that occurs in solving Laplace's equation using the nested dissection ordering of the conventional matrix formulation using finite differences.

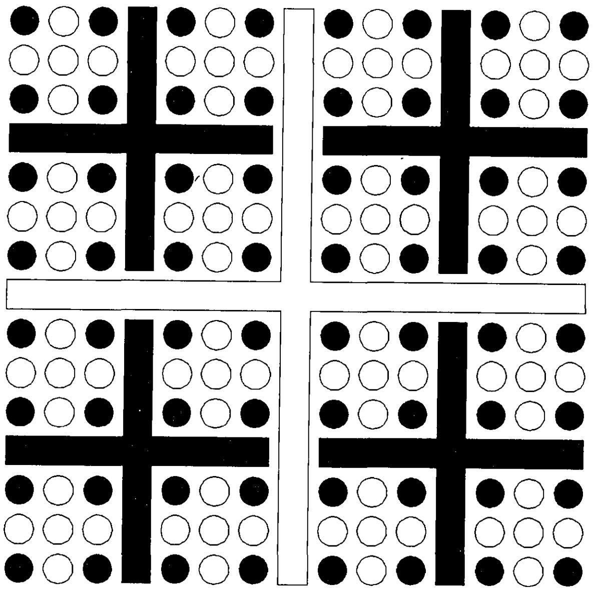


Figure 2. A visualization of the nested dissection ordering shown on a two dimension grid. Points are grouped in isolated blocks, one point per block at the first level, then 9, then 49, then 225 and so on.

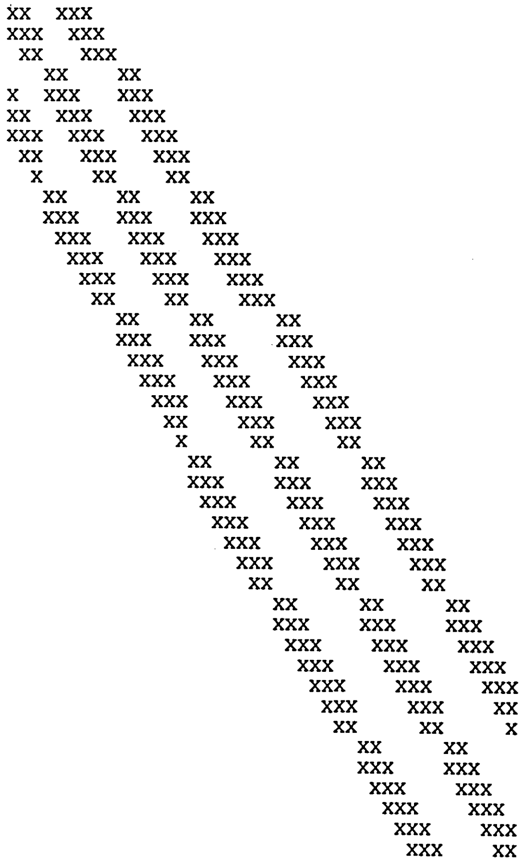


Figure 3. The conventional matrix structure obtained from a 9-point finite difference discretization on the domain of Figure 4.

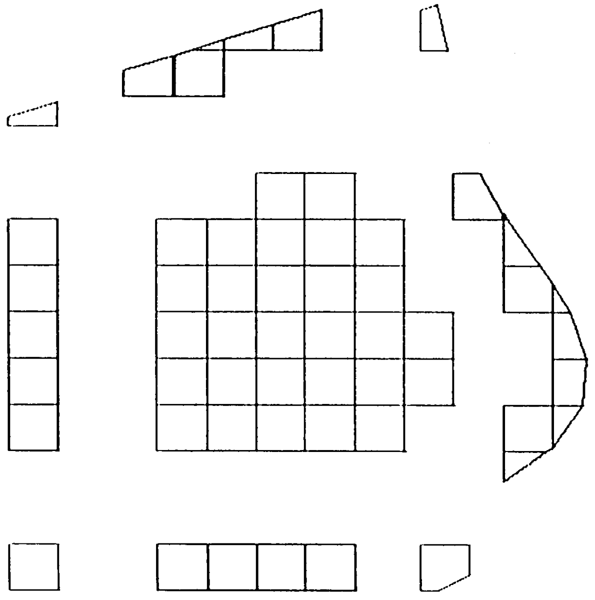


Figure 4. An exploded view of a physical domain which shows the elements of a "like" nature grouped together. The groupings are the first step in determining an appropriate structure in the problem of an efficient parallel method.

III. PARALLELISM IS (ALMOST) UNLIMITED IN SOLVING PDEs

We claim that the physical phenomena that PDEs model are inherently local in space and asynchronous. Locality means that the phenomena are inherently amenable to parallel methods, the computation done at point A does not depend on anything being done at the physically distant point B. There are logical limits to the potential parallelism, we do not foresee much parallelism in time (as opposed to space) except for very special situations. There is also some sequentiality in local computations, one must compute values of coefficient functions in an equation before one can use the equation. For specific applications one can often reduce the sequential work dramatically by preprocessing computations (i.e., computing everything possible as soon as possible).

The preceding observations are based on asymptotic considerations, i.e., if the physical domain is big enough and the accuracy required is high enough then any fixed number N of processors can be used profitably. We argue, however, that there is natural optimal or appropriate granularity and number N of processors associated with any particular PDE computations. We measure granularity in terms the number N of *elements* of the computation or model of the physical object. For simplicity we ignore any cases where computational elements do not correspond naturally with physical elements. The two extremes are:

(i) $N = 1$ processor gives 1 element which gives a sequential computation which gives very limited speed.

(ii) N very large (one Cray 2 per atom in a river?) gives a huge number of elements which gives very high parallelism which gives almost unlimited speed.

There are four considerations (at least) besides cost which lead to the existence of an optimal granularity, they are

1. Every problem has an *acceptable solve time* beyond which solving it faster does not matter.
2. Every problem has an *acceptable accuracy* beyond which more accuracy does not matter.
3. For a fixed physical problem, the number of interfaces between elements grows with the number of elements, thereby increasing the complexity and communication requirements of the computation. This growth might be very slow.
4. For a fixed problem and method, the total work might eventually grow faster with N than parallelism reduces it because of slower convergence of iterative methods, etc.

Having identified granularity with N, we see that the independent variables in an application design are N, the desired elapsed time T and the required accuracy ε. Assume now that ε behaves in a known way, that it is fixed and we only consider choosing N to achieve a specified T value. Figure 5 shows an idealized plot of cost versus time to solve a particular problem using a fixed number N of processors. The key points are that there is a lower limit on time (because processors can go only so fast) and that cost quickly reaches a plateau as the time increases. Figure 6 shows a different view of the situation, cost versus N for a fixed time to solve a particular problem. Again there is a lower limit because processors can go only so fast, but there is also an optimum. As N increases the cost starts to increase because of idle processors and/or increased communication (overhead) costs.

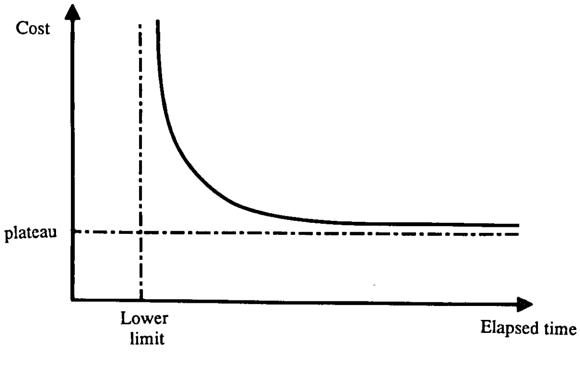


Figure 5. Cost versus elapsed time to solve a particular problem using N processors.

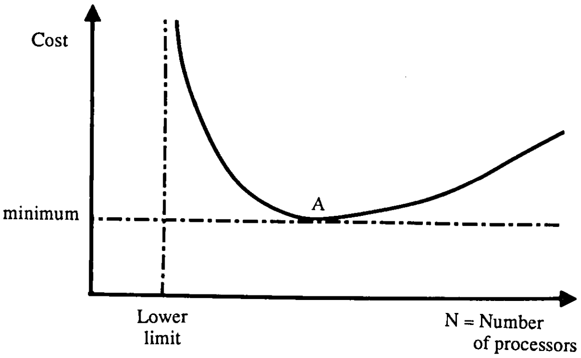


Figure 6. Cost versus the number of processors N used to solve a particular problem in a fixed elapsed time. The point A gives the minimum cost using an optimal number of processors.

We can replot the information of Figures 5 and 6 in the (N, T) plane and show two curves: the limiting curve of what is possible and the curve of optimal combinations of T and N. This is shown in Figure 7, the shapes are purely conjectural, one does not know what they are. It is true that cost decreases monotonically from point C to D.

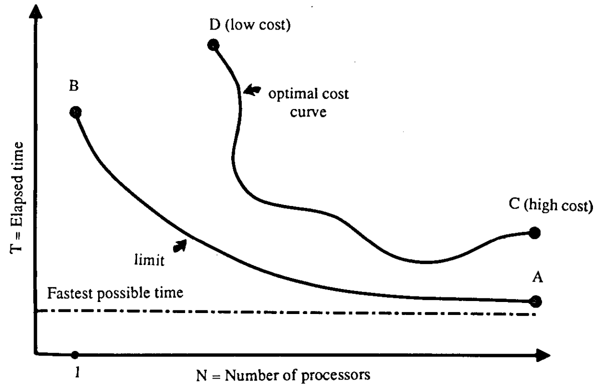


Figure 7. The (N, T) plane showing the limiting curve (A to B) of what is possible and the locus (C to D) of optimal cost combinations of N and T.

Thus we see that while in principle there might be no limit on the amount of parallelism that can be used in solving PDEs, there is definitely such a limit for any fixed application. Very little is known about actual values for real problems. I believe we are very far from the methods that give optimal time or cost in solving PDEs. On the other hand, I find it very convincing to argue that many real problems are very complex and that to achieve ''engineering'' accuracy and ''reasonable'' elapsed time with even a low cost method (never mind optimal cost) will use thousands of processors.

IV. PARALLEL METHODS REQUIRE NEW SOFTWARE SYSTEMS

Parallel machines are already rather complex, much more so than previous computers. They will become even more complex as it is discovered that a mixed set of capabilities provides more efficient computing. There will be variety is everything: processors (integer, floating point, graphics, vector, FFT, ...), memory (local, global, cache, archival, read only, ...), I/O (keyed, text, graphics, movies, acoustical, analog, ...), communication (message passing, packets, buses, synchronous/asynchronous, hypercubes, high/low speed, long haul, ...). The difficulty in managing (programming) this complexity is easily an order of magnitude higher than for present machines. The difficulty is compounded by the fact that changes in the capabilities available will become much more frequent.

The current programming methodology for solving PDEs is that of Fortran. One has a fairly intelligible language where one can exert fairly direct control of the machines resources. Each Fortran statement is typically implemental by 5-10 machines instructions. There must, I believe, always be such a language and I believe that Fortran will be expanded to handle the greater complexity of the machines. It might also be replaced by another moderate level language with such capabilities, e.g. Ada or C suitably enhanced. However, it will no longer be reasonable to expect the end-user scientists and engineers, the people who solve PDEs, to learn how to manage this complex computational environment. They will generally not do a very good job of it and, even if they did a good job, it would be a great waste of talent and duplication of effort. The potential benefits of parallel computation will not be achieved if every user has to master (even partially) how to manage such complex machines.

The solution to this problem is to substantially raise the level of the user's ''programming'' language. He must be able to say in a natural and succinct way what is to be done. In the PDE context they should be able to say things like:

1. Solve
   $(1+x^2)u_{xx} + u_{yy} - \sin(\alpha y)u = \text{Force}2(x, y)$
   on the Domain #12
   with $u = 1$ on the boundary.
2. Use finite differences with a 40 by 40 grid
   plus SOR iteration

3. Show me plots of u , u_x and u_y

In fact, we must aim eventually for the situation where statement 2. is replaced by

2a. Obtain an accuracy of about 0.5 percent

Then, between such a program and the Fortran level is a layer of software which has two components. The first is a set of *problem solving modules* written by people who are relatively expert in solving the problems at hand and experienced in how parallelism (or other special capabilities available) can be exploited. There will be different methods (or, at least, different implementations) in the modules suitable for important subclasses of machines.

The second component of this layer is a set of *computation management facilities* written by people who are relatively expert in memory management, network scheduling, program transformations, etc. They have spent the time to learn how to provide such facilities well and have embedded much of their expertise into their software. These two components are then integrated to provide a bridge between the high level user input and a Fortran-like program targeted for the particular machine (or machines) to be used to solve the problem.

The obvious advantage of this methodology is that, if it works, there is a dramatic reduction in programming effort. This is, of course, the goal of introducing the methodology. Note that this not being done just to reduce software costs, the "mass-market" viability of parallel computation depends on introducing a methodology which hides the underlying complexity from most users.

The obvious disadvantage of this methodology is that the intermediate layer might introduce so much in efficiency that the power of parallelism is seriously weakened or even lost. It is clear that no foreseeable software for managing a computation can be as clever, resourceful and effective as clever, experienced people. This fact is a smokescreen that obscures a much more relevant "fact": people, even clever and experienced ones, almost never get close to "optimal" computations because they do not take the time to do it, it is inordinately expensive to do so. The result is that a good software system, one with many flaws which does many obviously stupid things, consistently can produce moderately good implementations which are significantly better than the ones people consistently produce. Scientific evidence to support this fact is scarce, but there is one solid data point.

Figure 8 shows a program written in DEQSOL, a high level PDE problem solving language under development at Hitachi. No attempt is made here to explain DEQSOL. Hitachi has two PDE application programs that were written in FORTRAN prior to their vector supercomputer and DEQSOL efforts. These programs were brought into their vectorizing Fortran compiler environment and hand tuned to run well on their machines. The problems being solved were later reprogrammed in DEQSOL which produces a Fortran program which then use the vectorizing Fortran compiler but no hand tuning. The results of this experiment are shown below.

```
dom     x = [0:1] ,          /* 3D DIFFUSION PROBLEM */
        y = [0:1] ,
        z = [0:2] ;
tdom    t = [0:5] ;
mesh    x = [0:1:0.1] ,
        y = [0:1:0.1] ,
        z = [0:2:0.1] ,
        t = [0:5:0.001] ;
var     T ;                  /* Temperature */
const
  rho = 1 ,          /* Density */
  c = 1 ,            /* Constant */
  k = 1 ,            /* Diffusion Constant */
  u = 0 ,            /* x-axis Velocity */
  v = 0 ,            /* y-axis Velocity */
  w = 5*(1.0-x**2)*(1.0-y**2) ,      /* z-axis Velocity */
  S = exp(-x**2-x**2-(1.0-z)**2) ;
                             /* Source Distribution */

cvect   V = (u, v, w) ;      /* Velocity Vector */
region
  In = (*, *, 0) ,         /* In */
  0 = (*, *, 2) ,          /* Out */
  X0 = (0, *, *) ,         /* Left */
  X1 = (1, *, *) ,         /* Right */
  Y0 = (*, 0, *) ,         /* Bottom */
  Y1 = (*, 1, *) ,         /* Top */
  R = ([0:1], [0:1], [0:2]) ; /* Whole Region */

equ     rho*c*(dt(T)+V..grad(T)) = k*lapl(T)+S ;

bound   T = 0     at   In+X1+Y1 ,
        dz(T) = 0   at  0 ,
        dx(T) = 0   at  X0 ,
        dy(T) = 0   at  Y0 ;
init    T = 0  at  R ;

ctr     NT ;   /* Iteration Counter */

scheme ;
  iter NT until NT gt 200;
   T<+1> = T+dlt*((k*lapl(T)+S)/(rho*c)-V..grad(T)) ;
   print T at Y0 ;
   disp T at Y0 every 100 times ;
  end iter ;
end scheme ;
end ;
```

Figure 8. A DEQSOL program for solving a partial differential equation. This is a three dimensional, time dependent diffusion equation.

	A	B
FORTRAN:		
lines of code	1361	1567
execution time (sec.)	2.3	5.8
DEQSOL:		
lines of code	127	132
execution time	0.6	1.8
speed up factor	3.8	3.2

We see that not only was the programming effort reduced by the least an order of magnitude, but there was also a very worthwhile *gain* in execution speed. Keep in mind that a speedup of 3 or 4 is the typical total benefit achieved from using vector hardware on Cray and Cyber 205 machines.

We illustrate the power that can be achieved using such high level languages by considering the Plateau problem:

$$(1 + u_x{}^2)u_{xx} - 2u_x u_y u_{xy} + (1 + u_y{}^2)u_{yy} = 0$$
$$u(x, y) \text{ given on the boundary of a region } R \qquad (1)$$

This is classical difficult PDE problem, its solution is the surface that a soap film takes on for a wire frame bent according to the value specified on the boundary of R. We solve this problem for the domain R (shaped like a piano top) explicitly defined in Figure 9. The high level language used is that of ELLPACK[4] one that provides modules and facilities for solving linear PDEs.

Figure 9 shows an ELLPACK program to implement Newton's method for (1). We do not explain the ELLPACK language here. A simple initial guess is made and the convergence is quite rapid in spite of the fact that the solution has a singularity (the wire has a sharp bend) along one side. The maximum differences between iterates are: 1.24, .30, 9.6×10^{-3}, 2.4×10^{-5} and 5×10^{-7}. The round off level (on a VAX 11/780) is reached at five iterations.

Our final point in the software and programming area concerns the role *regularity* in data structures, in algorithms and in programs. Clever programmers and hardware designers can do a lot of special things to exploit special situations. This exploitation is usually achieved at the cost of more complex software and hardware. Thus there must be a balance between the execution time costs and the design costs of software and hardware. While it is hard to defend general statements on the matter, we believe that the optimum lies nearer to regularity and its attendant simplicity than it does to irregularity and its attendant complexity. However, we feel *extreme* simplicity is not the best approach either.

This view is illustrated by an example in discretizing a domain. Figure 10 shows a physical domain that has been partitioned in six ways for a problem with difficulties near the right boundary:

(A) A fine triangulation of a common type

(B) A fine, uniform, rectangular overlay grid

(C) Mapping the domain to a rectangle and inducing a logically rectangular partition

(D) Triangulation adapted to the difficulty

(E) Rectangular overlay grid adapted to difficulty

(F) Logically rectangular partition adapted to difficulty

```
EQUATION.
    (1.+uy(x,y)**2) uxx + (1.+ux(x,y)**2) uyy                  &
       - 2.*ux(x,y)*uy(x,y) uxy                                &
    + 2.*(ux(x,y)*uyy(x,y) - uy(x,y)*uxy(x,y)) ux              &
     + 2.*(uy(x,y)*uxx(x,y) - ux(x,y)*uxy(x,y)) uy             &
  =    2.*(ux(x,y)*uyy(x,y) - uy(x,y)*uxy(x,y))*ux(x,y)         &
    + 2.*(uy(x,y)*uxx(x,y) - ux(x,y)*uxy(x,y))*uy(x,y)
BOUNDARY.
    u = bound(x,y) on x = 1.0,                             &
                y = 0.5 + p        for p = 0.0 to 3.5
    u = bound(x,y) on x = 1.0 + p,                         &
                y = 4.0            for p = 0.0 to 3.0
    u = bound(x,y) on x = 4.0 + .1*p*(p-4.5)**2,           &
                y = 4.0 - p        for p = 0.0 to 4.5
    u = bound(x,y) on x = 4.0 - p,                         &
                y = -0.5 + p/3.    for p = 0.0 to 3.0

GRID.
    9 x points 1.0 to 5.5  $  9 y points -0.5 to 4.0

TRIPLE.  set ( u = gessu )

FORTRAN.
   do 100 it = 1, 5
           call save(r1unkn,i1neqn)
discretization.    collocation
solution.          band ge
output.            max(diffu)

FORTRAN.
   100 continue
OUTPUT.  table(u)  $  plot(u)
```

Figure 9. An ELLPACK program for applying Newton's method to solve the Plateau problem. The procedures *bound*, *diffu*, *gessu* = *xy* /4 and *save* are defined by 22 lines of Fortran not shown here.

We believe that the irregular triangulations do not provide any execution time advantage over the more regular partitions (one can do a regular triangulation if one wants). On the other hand, we also believe that the uniformly spaced partitions are too simple and have too large an execution time penalty. We believe the adaption will pay off. The logically rectangular discretization is the simplest to program but the relative execution efficiencies resulting from (E) and (F) are not clear. Thus we believe that the search for the "best" method should be concentrated on partitions like (E) and (F) but there are still many undetermined degrees of freedom.

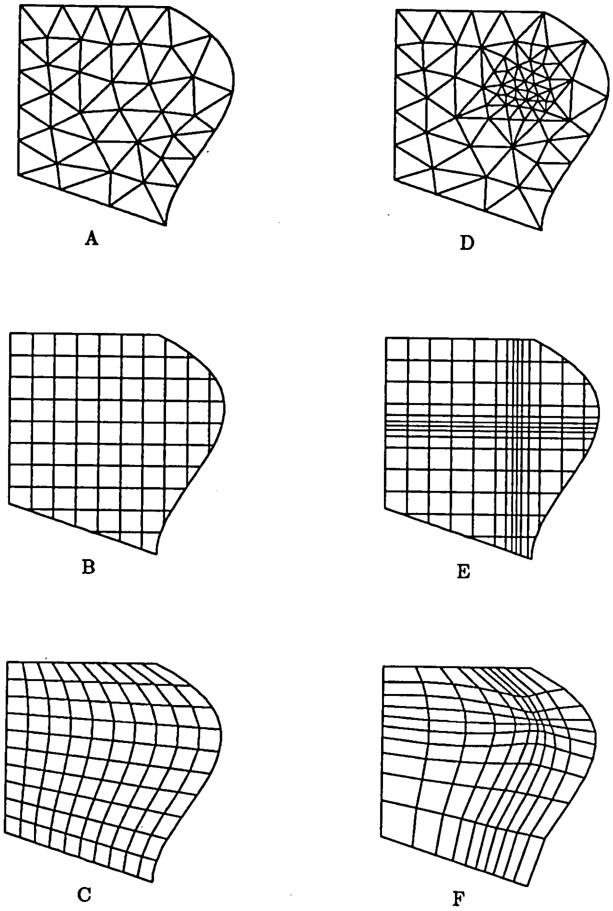


Figure 10. Six ways to partition a domain showing ways to achieve regularity and to adapt to a difficulty. The letters A through F refer to the discussion in the text.

V. REFERENCES

[1] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing,* New York, McGraw-Hill, 1984.

[2] K. Hwang, *Supercomputers: Design and Applications,* Silver Spring, IEEE Publication EH0219-6, 1984.

[3] J. Ortega and R. Voight, ''Solution of Partial differential equations on vector and parallel computers,'' SIAM Review, pp. 149–240, 1985.

[4] J. Rice and R. Boisvert, *Solving Elliptic Problems Using ELLPACK,* New York, Springer-Verlag, 1985.

[5] A. Sameh, ''An overview of parallel algorithms for numerical linear algebra,'' *First Int. Colloquium on Vector and Parallel Computing in Scientific Applications,* Paris, 1983.

GEOPHYSICAL MODELING - MIGRATION VIEWED AS A SPECTRUM OF SUPERCOMPUTER APPLICATIONS

Olin G. Johnson
Oliver Lhemann

Houston Area Research Center and University of Houston
Cray Research Inc., Minneapolis, Minnesota

ABSTRACT

This paper develops the thesis that no single algorithm is appropriate for all modeling-migration problems. Restricting attention to a single 3D problem size of 200^3 space points and 2,000 time points, the amount of computing required for migration can vary by two orders of magnitude depending on the complexity of the problem.

A review of reported timing results over the past four years leads to the identification of three classes of algorithms appropriate for three corresponding migration problem complexity levels. Both the computational and I/O requirements of these algorithm classes are discussed. Trade-off techniques for swapping I/O for computing are discussed and examples are given for the resulting CPU times. Graphical results from a variety of 3D problems are given.

INTRODUCTION

The successful replication of the methods of reflection seismology in scaled modeling tanks over the past few years has at last turned exploration geophysics into a laboratory discipline. Pioneering organizations such as the Seismic Acoustics Laboratory (SAL) at the University of Houston now have extensive catalogues of seismic sections in both 2D and 3D corresponding to layered structures of known dimensions and acoustic properties.

These data sets are invaluable in research efforts directed to duplicating these techniques with computerized models and in testing seismic codes. The Keck Research Computation Laboratory (RCL), a sister laboratory to SAL, primarily pursues research in the use of supercomputers, array processors and parallel computer ensembles in the 3D exploration geophysics area.

Several modeling and migration algorithms have been studied, programmed and tested for accuracy of results using the SAL data.

Two models and their respective data sections are used here. The Wedge Model, pictured in Figure 1, is a simple three layer model with a silicon rubber wedge setting on a plexiglass plate, all submerged in water. Figure 2 exhibits a 3D seismic section of the SALNOR7 model that illustrates a typical complicated North Sea geological structural sequence. The model consists of seven horizons that represent the top Paleocene, top Cretaceous, J-Unconformity, top Brent, base Brent, top Statfjord and base Statfjord. A 3D data set consisting of 240 traces on 240 lines has been collected. Each trace contains 3000 samples at 1 ms. A complete description of this model is given in (Nelson et al., 1982).

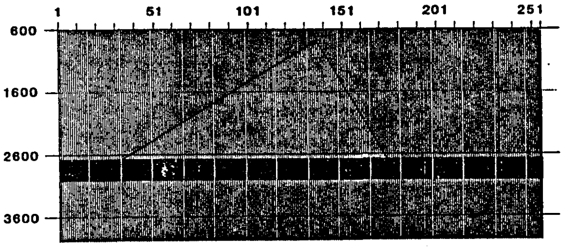


Figure 1. Structure of the Buried Wedge Model.

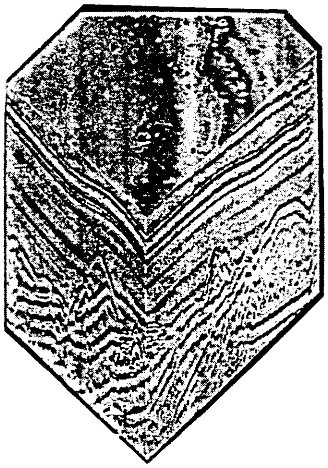


Figure 2. The SALNOR7 Model Time Section.

CH2345-7/86/0000/0547$01.00 © 1986 IEEE

The computational methods reported here fall into two categories: phase shift techniques which allow no lateral variation in velocity and phase shift plus some type of correction which allow "mild" lateral variations. Methods which allow arbitrary lateral variations are available[7]. In the 3D case of size 200^3 space points and 2000 time points, such techniques require several hours of computation on present generation supercomputers.

As will be shown, the computational requirements for these three algorithms can vary by two orders of magnitude.

A primary objective of the RCL program has been to construct the fastest 3D migration program possible. This is accomplished by limiting the data to a single velocity function and optimizing and parallelizing the data handling sections of the code.

HYPERSPEED MIGRATION

In Lhemann[8] a code for 3D migration based on the Phase Shift and PSPI algorithms is described which took less than three minutes of supercomputer wall-clock time to migrate the North Sea data cube of size 256x256x2048.

The Phase Shift migration method is based upon the scalar wave equation

$$\frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} \qquad (1)$$

where $p = p(x,y,z,t)$ is the pressure field, x and y are the midpoint variables, z is the depth, t is the two-way travel time, and V is the half velocity. Assuming the V does not depend on x or y, the Fourier transform with respect to x, y and t leads to the following equation

$$k_x^2 P + k_y^2 P + \frac{\partial^2 P}{\partial z^2} = \frac{w^2 P}{V^2} \qquad (2)$$

with $P(k_x,k_y,z,w) = FFT_{x,y,t}[p(x,y,z,t)]$. Grouping all terms depending on P on one side, one gets

$$\frac{\partial^2 P}{\partial z^2} = (\frac{w^2}{V^2} - k_x^2 - k_x^2)P$$

Let $$k_z = \pm \sqrt{(\frac{w^2}{V^2} - k_x^2 - k_y^2)}$$

then $$\frac{\partial^2 P}{\partial z^2} = - k_z^2 P. \qquad (3)$$

The non-negative (complex) value of k_z is taken reflecting the downward extrapolation of the recorded seismic data in the migration process.

This last equation (3) has an analytical solution

$$P(k_x,k_y,z+\Delta z,w) = P(k_x,k_y,z,w)\exp(ik_z\Delta z) \qquad (4)$$

In practice, a phase shift migration first reads the seismic data $p(x,y,z=0,t)$ and performs a triple Fourier transform to get $P(k_x,k_y,z=0,w)$. The Phase Shift Factor (PSF) $\exp(ik_z\Delta z)$ is then calculated, and downward extrapolation is applied as given by equation (4). The migrated result (in the space domain) is obtained after $P(k_x,k_y,z,w)$ has been calculated for all w by summing over all w[7].

$$P(k_x,k_y,z,t=0) = \sum_w P(k_x,k_y,z,w). \qquad (5)$$

The final section $p(x,y,z,t=0)$ is obtained by performing a double inverse Fourier transform in the space domain.

The Phase Shift algorithm has a lot of practical advantages, it is unconditionally stable, accurate and the amount of computation involved is limited. Gazdag's[8] generalization of this algorithm, Phase Shift Plus Interpolation (PSPI), treats laterally varying velocity fields and consists of two steps. In the first step, the wave field is extrapolated as described above, using n different laterally uniform velocity fields. In the second step the actual wave field is computed by interpolation from the n reference wave fields. In the usual case where n is equal to 3 (minimum, maximum and average velocities) one obtains three fields P1, P2 and P3 at depth $z+\Delta z$. To get the actual field $P(k_x,k_y,z+\Delta z,w)$ one must perform an interpolation between the 3 previous fields. Since this interpolation depends on the velocity structure, a double inverse Fourier transform must be performed for each of the fields ($P1(k_x,k_y,z+\Delta z,w)$, $P2(k_x,k_y,z+\Delta z,w)$ and $P3(k_x,k_y,z+\Delta z,w)$ to get $p1(x,y,z+\Delta z,w)$, $p2(x,y,z+\Delta z,w)$ and $p3(x,y,z+\Delta z,w)$.

This is then followed by a three-point Lagrangian interpolation.

$$p(x,y,z,t = \frac{(\frac{1}{V} - \frac{1}{V3})(\frac{1}{V} - \frac{1}{V2})}{(\frac{1}{V1} - \frac{1}{V3})(\frac{1}{V1} - \frac{1}{V2})}\, p1(x,y,z,t)$$
$$+ \frac{(\frac{1}{V} - \frac{1}{V1})(\frac{1}{V} - \frac{1}{V3})}{(\frac{1}{V2} - \frac{1}{V1})(\frac{1}{V2} - \frac{1}{V3})}\, p2(x,y,z,t)$$
$$+ \frac{(\frac{1}{V} - \frac{1}{V1})(\frac{1}{V} - \frac{1}{V2})}{(\frac{1}{V3} - \frac{1}{V1})(\frac{1}{V3} - \frac{1}{V2})}\, p3(x,y,z,t)$$

The velocities V1, V2 and V3 are the reference velocities defined previously. $V = V(x,y,z)$ is the actual 3D velocity field. The migrated result at depth $z+\Delta z$ is obtained by summing over w all the planes $p(x,y,z+\Delta z,w)$. To go deeper one must take the double Fourier transform of the previous plane and repeat the process just described.

The PSPI program requires a complete 3D velocity model and is much slower than simple phase shift. However, phase shift is quite adequate for sedimentary sections and can serve as the basis of a hyperspeed code.

A first estimate of the amount of I/O required by the Phase Shift algorithm to migrate the SALNOR7 model gives ($4*10^6$ sectors) and so, assuming the average transfer time of a sector to be 0.7 ms, results in a time of 2800 s for the I/O. This number is much bigger than the estimated CPU time (around 140 s) required by present supercomputers.

A second look at the algorithm indicates that the minimization of disk transfers is equivalent to keeping in memory, as long as possible, each plane $P(k_x,k_y,z,w)$. An obvious solution simply consists in switching the frequency and the depth loops. $P(k_x,k_y,z=0,w)$ is loaded only once for a fixed w, and all extrapolation steps can be performed in core to get $P(k_x,k_y,z=z+\Delta z,w)$, $P(k_x,k_y,z=z+2\Delta z,w)$ ... up to the maximum depth $P(k_x,k_y,z=z+NZ\Delta z,w)$.

If there is not enough space in main memory, the previous planes will have to overwrite each other during the execution of the program. But before doing this each of them must be added as described in equation (5). Since present supercomputers do not have sufficient memory to treat large 3D problems in core, the trick is to work with several frequencies at the same time to reduce the I/O transfers in proportion.

If 16 frequencies are grouped together, a new estimation of the amount of I/O required by the modified algorithm is now around 175 s, instead of 2800 s. This can be reduced further by simultaneous I/O transfers. Hence, the resulting 3D migration code is compute bound.

The SALNOR7 model was padded with null traces to get 256 lines of 256 traces each for use with the FFTs. For the same reason all traces were truncated to keep only 2048 samples at 1 ms since all relevant information is at less than 2 s.

The size of the data cube is now equal to 256x256x2048, which represents a volume of 64 M words.

The maximum depth of the migrated section is equal to NZSTEP*ΔZ, where NZSTEP is the number of iterations of the depth loop. To get a good resolution, ΔZ = 10 meters and NZSTEP = 128 were used to reach the bottom of the model at Z = 1280 meters.

As seen previously, the Phase Shift migration algorithm uses a one-dimensional velocity structure. Rather than choosing average values as one would usually do, we have selected a particular section (line 120) and taken for the velocity function the actual values at a position corresponding to the middle trace. This was done in order to compare the output with previously migrated results for this particular section.

The frequency spectrum of seismic data is very limited, usually under 70 Hz. In the case of the SALNOR7 model the data were filtered by a 6 - 60 Hz band-pass filter. Since Δt is 1 ms and the number of samples is 2048, it follows that ΔHz = 1/2.048. So it is necessary to sum the computed results for the first 128 frequencies only.

To migrate the entire SALNOR7 model on presently available supercomputers, it only takes 131 s of CPU and 11 s of I/O, which gives a total run-time of 142 s. The amount of memory required by this migration program is 3.2 M words.

WAVE EXTRAPOLATION EQUATIONS

In the presence of lateral inhomogeneities in the velocity field, the extrapolation equations obtained from the Phase Shift algorithm are usually approximated and solved using finite difference extrapolators. A review of some of these approximations is given below. To simplify notation, equations are developed only for the two dimensional (2D) case. The extension to the 3D case is straight forward.

The theory of wave extrapolation is based on the assumption that the zero offset pressure data, defined in the (x,z) domain, satisfy the wave equation

$$\frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial z^2} \qquad (6)$$

with p = p(x,z,t) where x is the distance along the surface, z the depth, t the two-way traveltime, and v the half velocity.

In the case where the velocity v does not vary laterally, a double Fourier transform with respect to x and t is performed on the previous equation. We obtain

$$\frac{w^2}{V^2} P = K^2xP + \frac{\partial^2 P}{\partial z^2} \qquad (7)$$

with $P(Kx,z,w) = FFT_{x,t}[p(x,z,t)]$. From this equation, an analytical solution to the downward extrapolation problem can be deduced[6].

$$P(Kx,z+\Delta z,w) = P(Kx,z,w) * EXP(iKz\Delta z) \qquad (8)$$

$$\text{with } Kz = \pm \, SQRT(\frac{w^2}{V^2} - K^2x).$$

Among the two possible solutions, only the one with the positive sign in the exponential (or phase shift) factor is relevant. This comes from the fact that in the migration process we are only interested in waves moving in the reverse time direction.

In the case where the velocity v is a function of both x and z, the analytic solution given in (8) is no longer valid. This comes from the fact that in taking a double Fourier transform of (6), with respect to t and x, we do not get equation (7) anymore but the following

$$\frac{\partial^2 P}{\partial z^2} = w^2 \text{ FFTx } (\frac{1}{V(x)^2}) \oplus p - K^2 xp \qquad (9)$$

where ⊕ is defined as the convolution product.

The product of the inverse of the velocity by the pressure field P is now replaced by a product of convolution which rules out all chances of finding a simple analytical solution. A common method employed to circumvent this problem is to use the analytical solution derived in the constant velocity case and modify it in such a way that it can accommodate lateral velocity variations.

For this purpose, one uses the following equation

$$\frac{\partial P}{\partial z} (Kx,z,w) = i(\frac{w}{v}) \; [1-(\frac{Kxv}{w})^2]^{1/2} \; P(Kx,z,w) \qquad (10)$$

which has as solution equation (8) and is thus the exact extrapolation equation for the constant velocity case. Lateral variations of the velocity are possible if we inverse transform over x by letting iKx = Δx. Regrettably, this transform can only be performed if the square root is regarded as some kind of truncated series expansion. The square root can be developed into a Taylor series expansion[5], or as it was first suggested by Francis Muir[4], into continued fractions. In this second method, the square root

$$R = (1 - K^2)^{1/2} \qquad (11)$$

is expanded according to the following recurrence relation

$$R_{n+1} = 1 - \frac{K^2}{1 + R_n} \qquad (12)$$

in which R_n is the nth order approximation. Starting from $R_o = 1$, we obtain

$$R_1 = 1 \; \frac{K^2}{2} \qquad (13)$$

and

$$R_2 = 1 - \frac{K^2}{2 - K^2/2} \qquad (14)$$

Francis Muir showed that the original 15 degree and 45 degree methods developed by J. Claerbout[3] correspond to the expansion defined by R1 and R2 respectively.

The substitutions K = Kxv/w and Rn = Kzv/w give us the dispersion relations corresponding to these expansions. For example, the dispersion relation of the 45° method is

$$Kz = \frac{w}{V} - \frac{(v/2w)K^2x}{1 - (Kxv/2x)^2} \cdot \qquad (14a)$$

This expression can be used to replace the square root term in (10) to give us an approximated extrapolation equation

$$\frac{\partial^2 P}{\partial z^2} = i \; [\frac{w}{V} - \frac{(v/2w)K^2x}{1 - (Kxv/2w)^2}] * P. \qquad (14b)$$

Such an equation can be solved using a finite-difference scheme in the (x,w) domain. A method of resolution is discussed in Gazdag and Sguazzero[6].

These are several problems associated with finite-difference migration methods, including numerical errors resulting from the finite-different approximation used and predisposition to instability. Numerical erros can be reduced by using a small integration step ΔZ and the stability problem, is kept under control by using implicit methods. Unfortunately, such improvements appear to be quite costly in CPU time, especially for 3D problems.

THE PHASE SHIFT PLUS CORRECTION ALGORITHM

In this section, a migration algorithm suitable for variable velocity fields is introduced. Basically, the idea is first to perform a downward extrapolation of the pressure field using the phase shift algorithm and a constant reference velocity v̄. A correction term is then introduced, to take into account the lateral velocity changes in the velocity field.

If v is the actual velocity field, it is known from the previous section that the exact dispersion relation in the two dimensional case can be written in the form

$$Kz(v) = (\frac{w}{v}) * \text{SQRT } [1 - (\frac{Kxv}{w})^2] \qquad (15)$$

which, if we use the relation

$$Kz(v) = Kz(\bar{v}) + (Kz(v) - Kz(\bar{v})] \qquad (16)$$

is also equivalent to

$$Kz(v) = (\frac{w}{\bar{v}}) * \text{SQRT } [1 - (\frac{Kx\bar{v}}{w})^2] +$$

$$[(\frac{w}{v}) * \text{SQRT}[1-(\frac{Kxv}{w})^2] - (\frac{w}{\bar{v}}) * \text{SQRT}[1-(\frac{Kx\bar{v}}{w})^2]]. \qquad (17)$$

As in the previous section, velocity variations will be introduced after inverse transforming over x by Kx = Δx. This transformation is possible if the second term is developed as some kind of truncated series. Using a Taylor development and keeping only the first three terms of the development, equation (17) becomes

$$Kz(v) = (\frac{w}{\bar{v}}) *SQRT[1-(\frac{Kx\bar{v}}{w})^2] + (\frac{1}{v} - \frac{1}{\bar{v}})w \tag{18}$$

$$-(v - \bar{v}) \frac{K^2x}{2w} - (v^3 - \bar{v}^3) \frac{K^4x}{8w^3}.$$

To see how this expression can be used practically, let us first consider the case where only the firt term of the expansion is kept (first order PSPC).

$$Kz(v) = (\frac{w}{\bar{v}}) *SQRT[1 - (\frac{Kw\bar{v}}{w})^2] + (\frac{1}{v} - \frac{1}{\bar{v}})w. \tag{19}$$

After substituting this dispersion relation into equation (8), we obtain the following expression

$$P(Kx,z + \Delta Z,w) = \tag{20}$$

$$P(Kx,z,w) *EXP \{i(\frac{w}{\bar{v}})[1 -$$

$$(\frac{Kx\bar{v}}{w})^2]^{1/2} \Delta Z\} *EXP \{i(\frac{1}{v} - \frac{1}{\bar{v}})w\Delta Z\}.$$

So, by defining

$$P^*(Kx,z + \Delta Z,w) = \tag{21}$$

$$P(Kx,z,w) *EXP \{i(\frac{w}{\bar{v}})[1 -(\frac{Kx\bar{v}}{w})^2]^{1/2} \Delta z\}$$

(20) becomes

$$P(Kx,z + \Delta Z,w) = \tag{22}$$

$$P^*(Kx,z + \Delta Z,w) *EXP \{i(\frac{1}{v} - \frac{1}{\bar{v}})w\Delta Z\}$$

After inverse transforming this expression into the real domain, one obtains an expression in which we can vary the velocity,

$$p(x,z + \Delta Z,w) = p^*(x,z + \Delta Z,w) *EXP\{i(\frac{1}{v} - \frac{1}{\bar{v}})w\Delta Z\}. \tag{23}$$

As it is seen from this equation, the extrapolation is performed in two steps. The Phase Shift algorithm is first applied in the wave-number domain with a constant reference velocity $\bar{v}$ in order to obtain P^*. A correction term is then applied in the real domain to take into account lateral variations in the velocity field. In the first order scheme, the correction term is merely equivalent to a time-shift, no correction is performed on the diffraction term.

Let us now concentrate on the second order approximation (the development of the third order expression is similar and it will not be treated), where the dispersion relation reads

$$Kz(v) = (\frac{w}{\bar{v}}) *SQRT [1 - (\frac{Kx\bar{v}}{w})^2] + \tag{24}$$

$$(\frac{1}{v} - \frac{1}{\bar{v}})w\Delta Z - (v - \bar{v}) \frac{Kx^2\Delta Z}{2w}$$

Using the same notation as in the previous case, the downward extrapolation equation is now of the form

$$P(Kx,z + \Delta Z,w) = P^* (Kx,z + \Delta Z,w) \tag{25}$$

$$*EXP\{i(\frac{1}{v} - \frac{1}{\bar{v}}) w\Delta Z\} *EXP\{-i(v - \bar{v}) \frac{Kx^2\Delta Z}{2w}\}$$

Because of the Kx^2 term in the last exponential, we cannot perform an inverse transform as previously to go back in the x-domain. The solution is to replace the exponential by its first order Taylor development. In other words, we replace

$$EXP\{-i(v - \bar{v}) \frac{Kx^2\Delta Z}{2w}\} \tag{26}$$

by

$$1 - i(v - \bar{v}) \frac{Kx^2\Delta Z}{2w}. \tag{27}$$

This is valid only if the value of the expression inside the exponential is small enough. In order to evaluate it, let us write it in the following form

$$(v - \bar{v}) \frac{Kx^2\Delta Z}{2w} = (\frac{v}{\bar{v}} - 1) (\frac{Kx\bar{v}}{w}) (\frac{Kx\Delta Z}{2}). \tag{28}$$

If the following condition for the velocities is assumed

$$0.5 \leq \frac{v}{\bar{v}} \leq 1.5 \tag{29}$$

we have

$$|\frac{v}{\bar{v}} - 1| \leq 1/2. \tag{30}$$

The inequality assumed in (29) puts some restrictions on the velocity field. More specifically, the method is applicable only if the velocity field is such that, for each depth step, the ratio of the largest over the smallest velocity is less or equal to 3. Let us now concentrate on the second term of (28). We show that we have the following inequality

$$(\frac{Kx\bar{v}}{w}) \leq 1. \tag{31}$$

This inequality is related to the problem of the elimination of the evanescent energy. For a laterally uniform medium $\bar{v}$, the evanescent solutions are defined by the relation[7].

$$Kx \le \frac{\bar{w}}{v}. \qquad (32)$$

Thus, when P^* is computed with the constant velocity $\bar{v}$, we keep only the waves such that

$$Kx \le \frac{w}{\bar{v}}. \qquad (33)$$

When the velocity field varies laterally, the identification of the evanescent field becomes less clear cut. However, in our case, we choose to apply inequality (33) everywhere, and so (31) is verified. It has thus been shown that

$$|(v - \bar{v}) \frac{Kx^2\Delta Z}{2w} | \le \frac{Kx\Delta Z}{4} \qquad (34)$$

or

$$|(v - \bar{v}) \frac{Kx^2\Delta Z}{2w} | \le \frac{\pi\Delta Z}{8\Delta X} \qquad (35)$$

because of the inequality $Kx \le \frac{\pi}{2\Delta x}$. So finally,

if ΔZ is small enough compared to ΔX, the first order development of the exponential term will be accurate. In practice, one can choose ΔZ such that

$$\Delta Z \le \frac{\Delta X}{3}. \qquad (36)$$

Now our extrapolation relation defined in (25) can be written in the form

$$P(Kx,z + \Delta Z,w) = P^*(Kx,z + \Delta Z,W) * \qquad (37)$$

$$[1 - i(v - \bar{v}) \frac{Kx^2\Delta Z}{2w}] *EXP \{i (\frac{1}{v} - \frac{1}{\bar{v}}) w\Delta Z\}$$

An inverse transform to go back in the x-domain gives us our final extrapolation expression for the second order PSPC method.

$$p(x,z + \Delta Z,w) = [p^* (x,z + \Delta Z,w) + \qquad (38)$$

$$i(v - \bar{v}) \frac{\Delta Z}{2w} \frac{\partial^2 p^*}{\partial x^2} (x,z + \Delta Z,w)] *EXP \{i \frac{1}{v} - \frac{1}{\bar{v}}) w Z\}$$

The partial derivative term is computed by the Fourier method.

ACCURACY OF THE MIGRATION ALGORITHM

In this section we study the PSPC algorithm both from a theoretical and a practical point of view. From what was shown in the previous section, we know that the accuracy of the PSPC method mainly depends on two factors. The first of these factors is the order of the Taylor development used to approximate the dispersion relation defined by (Equation 17). The second important factor is the ratio $R = v/\bar{v}$.

If we refer to the wave extrapolation equations section, we see that in term of complexity, the first order PSPC approximation is comparable to the 5 degree method, the second order approximation to the 15 degree method, and the third order approximation to the 45 degree method. Figure 3 compares the dispersion curve of the 5 degree method and of the first order PSPC approximation for different values of R, to the exact dispersion relation.

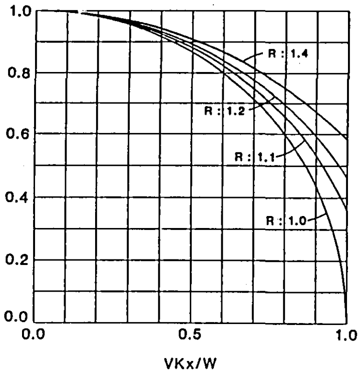


Figure 3. Dispersion Curve of the First Order PSPC Method.

Compared to the 5 degree method, dispersion curves of the first order PSPC method follow much better the exact dispersion curve. Figure 4 plots, for different values of R, the angle where the curve departs from the exact dispersion relation. For the values of R between 0.8 and 1.5 the first order PSPC method approaches the exact curve up to 9 degree dips or better.

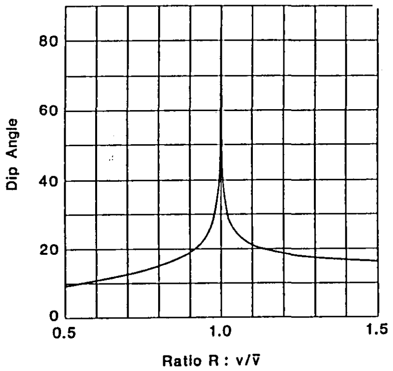


Figure 4. Theoretical Accuracy of the First Order PSPC Method.

Figure 5 compares the dispersion curve of the 15 degree method and of the second order PSPC approximation for different values of R, to the exact dispersion relation. Again, the behavior of the dispersion curves is quite good. Figure 6 plots, for different values of R, the angle where the curve departs from the exact dispersion relation. For the values of R between 0.8 and 1.5, the second order PSPC method approaches the exact curve up to 15 degree dips or better.

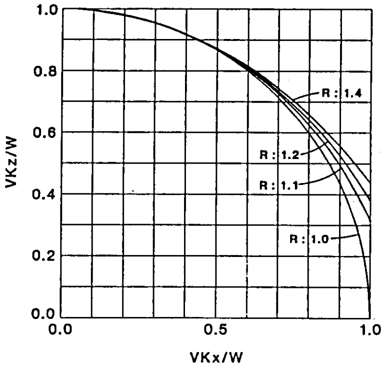


Figure 5. Dispersion Curve of the Second Order PSPC Method.

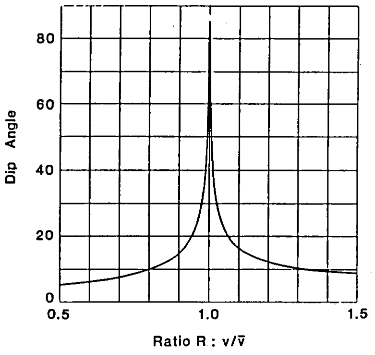


Figure 6. Theoretical Accuracy of the Second Order PSPC Method.

In order to test our algorithm, we have chosen to migrate a section that was collected in the acoustic modeling tank at the Seismic Acoustic Laboratory at the University of Houston. The same model was used by Kosloff and Baysal to test their migration algorithm with the full acoustic wave equation[7]. The model is a buried wedge structure, which is represented in Figure 1. The wedge was made of low-velocity rubber, whereas the base was made of high-velocity plexiglas. The whole model was immersed in water.

The dimensions of this model were chosen so that the data would represent field data. The scale factors used in the experiment are as follows:

Parameter	Model	Field
Time	$0.2\mu s$	1 msec
Length	2.5 cm	300 m
Velocity	V	2.4 V

In the following all parameters will be discussed in terms of field units. The zero offset line that was collected in the tank is made of 512 traces, with a shot spacing of 15 m. The first sample of each trace starts at T = 400 ms. This section is shown in Figure 7, for clarity reasons only every other trace (256 of the 512 traces) has been plotted.

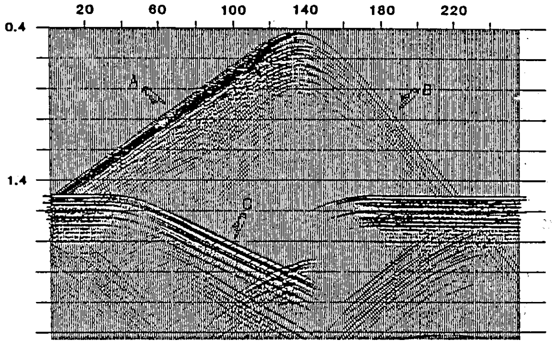


Figure 7. Time Section from a Zero-Offset Line Shot in the Physical Modeling Tank.

In a first test, this section has been migrated using the Phase Shift algorithm and a uniform velocity. The value used was 3600 m/s, which corresponds to the scale water velocity in the tank. The result of this migration is shown in Figure 8. The program was designed so that it is possible to skip the water in one single step, which explains why the depth section starts at Z = 600 m. Events A and B are correctly migrated, but event C, the plexiglas base underlying the wedge is undermigrated. The scale velocity for the rubber is equal to 2400 m/s, which is much slower than the velocity of 3600 m/s that was used in this first migration test.

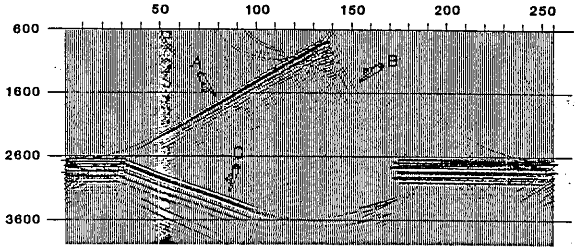


Figure 8. Migrated Section with a Uniform Velocity of 3600 m/s.

e velocity model to test our variable velocity migration programs is shown in Figure 1. The scale velocity for plexiglas was taken equal to 6000 m/s. Figure 9 shows the result of the migration using the first order PSPC method. The plexiglas base under the wedge is now in its proper position, although one portion is missing. According to Kosloff and Baysal[7] this comes from the fact that the energy propagating upward from this portion of the base encounters the steeply dipping side of the wedge at an angle beyond the critical angle for the wedge rubber and water. This type of migration is not accounted for by the exploding reflector model on which the migration is based. In this example the water velocity was used as the reference velocity, which means that although we used only the first order method, the 30 and 60 degrees dips of the wedge are correctly migrated because we are in a situation where R is equal to 1. On the other hand, if one chooses to use for the reference the wedge velocity, R is now equal to 1.5, which is the worst situation. The migrated section for this case is shown in Figure 10. Reflectors A and B are not properly positioned. The shape is preserved, but the dip angles are not recovered correctly.

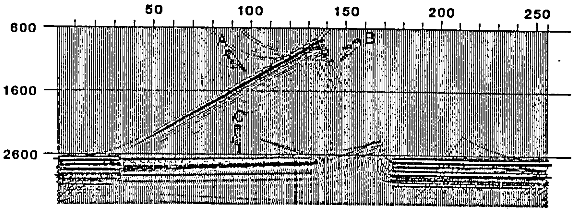


Figure 9. Migrated Section with a Variable Velocity. First Order PSPC with a Reference Velocity Equal to 3600 m/s.

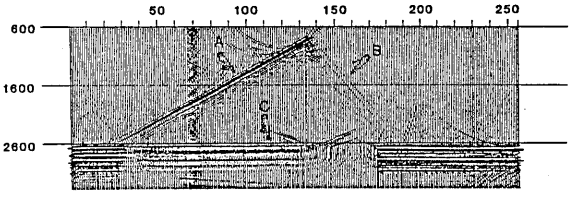


Figure 10. Migrated Section with a Variable Velocity. First Order PSPC with a Reference Velocity Equal to 2400 m/s.

Another interesting remark is that, contrarily to what reported Kosloff and Baysal for their migration algorithm with the full acoustic wave equation, our migration method is relatively insensitive to small changes in the values of the velocities. For example, a slight perturbation in the wedge velocity does not significantly modify the alignment of the plexiglass base under the wedge.

A 3D EXAMPLE: THE SALNOR7 MODEL

One of the main motivation for developing our Phase Shift Plus Correction migration method was to come up with an algorithm suitable for 3D processing. The purpose of this section is to report on 3D migration tests performed with this algorithm.

The SALNOR7 model, Figure 2, is used for the 3D test. As previously, all parameters will be discussed in terms of scaled units. A 3D zero-offset data set consisting of 240 traces on 240 lines has been collected in the tank. Each trace contains 3000 samples at 1 ms, the shot spacing in both directions is equal to 30 m. As our program works in the Fourier domain, it is more convenient to have dimensions that are power of 2 and so the model was padded with null traces to get 256 lines of 256 traces each. For the same reason data were truncated to keep only 2048 samples at 1 ms (all relevant information is at less than 2 s).

The size of the model for the migration is now equal to 256^*256^*2048, which represents a volume of 64 Mwords. Figure 11 shows a section (line 120) of this model. This section was first migrated using the Phase Shift algorithm, and a uniform velocity for each of the 128 depth steps. This first migration was completed in 142 s wall-clock time on a dedicated Cray X-MP 48 (using only one CPU). Figure 12 shows the result of this migration on line 120.

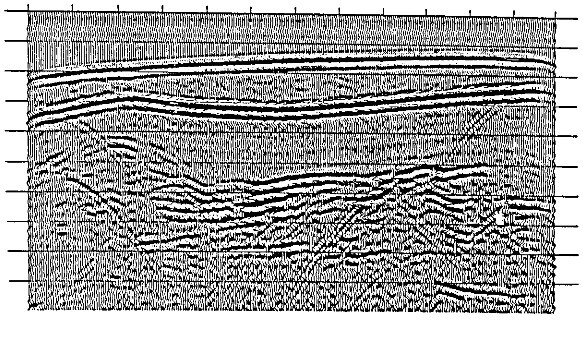


Figure 11. Line 120 of the SALNOR7 Model. Zero-Offset Time Section.

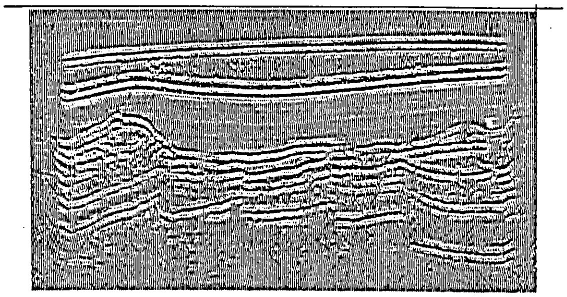


Figure 12. Line 120 of the SALNOR7 Model. 3D Migrated Depth Section Using The Phase Shift Method.

For our second test we ran the first order PSPC algorithm, with a laterally variable velocity field. Due to the fact that the SALNOR7 is a very complex model and that our tools are not appropriate, we used only a simplified 3D velocity structure. The migrated results thus do not show much difference over the previous case, however, this does not affect our benchmark results. Using the same parameters than in the previous case, plus the 3D velocity structure, the first order PSPC migration program requires 28 minutes wall-clock time on the Cray X-MP 48, using only one CPU. Table 1 gives a summary of these results, including some estimates on the performance that we can expect for the multitasked versions. All the results include both CPU and Input/Output waiting time.

Version	Size	Steps	Main Memory	Wall-Clock Time
Phase Shift 1-CPU	256 X 256 X 2048	128	1.8 Mwords	142 s
PSWC1 1-CPU	256 X 256 X 2048	128	2.1 Mwords	1719 s
PSWC1 4-CPU	256 X 256 X 2048	128	3.7 Mwords	470 s (estimated)
PSWC2 1-CPU	256 X 256 X 2048	128	2.3 Mwords	2300 s (estimated)

Table 1. 3D Migration Benchmark Results on a Dedicated Cray X-MP 48.

REFERENCES

(1) A.J. Berkhout, Wave field extrapolation techniques in seismic migration, a tutorial: Geophysics, 46, 1638-1656, 1981.

(2) A.J. Berkhout, Seismic migration imaging of acoustic energy by wave field extrapolation: A. Theoretical Aspects: Elsevier Science Publishers, B.V., 1985.

(3) J.F. Claerbout, Fundamentals of geophysical data processing, McGraw Hill, New York, 1976.

(4) J.F. Claerbout, Dispersion-related derivation of wave extrapolators: Stanford Exploration Project, 24, 278-286, 1980.

(5) J. Gazdag, Wave equation migration with the accurate space derivative method: Geophys. Prosp., 28, 60-70, 1980.

(6) J. Gazdag and P. Sguazzero, Migration of seismic data by phase shift plus interpolation: Geophysics, 49, 124-131, 1984.

(7) D. Kosloff and E. Baysal, Migration with the full acoustic wave equation: Geophysics, 48, 677-687, 1983.

(8) O. Lhemann, A 3D PSPI migration: Research Computation Laboratory, Annual Progress Review, University of Houston, 1, 86-108, 1985.

A TUTORIAL ON FINITE DIFFERENCE METHODS AND ORDERING OF MESH POINTS

David M. Young* - David R. Kincaid*

* The University of Texas at Austin, U. S. A.

ABSTRACT

This paper contains introductory material related to a tutorial on finite difference methods and ordering of mesh points.

We wish to illustrate a central mathematical idea which has grown in importance with the advent of the modern computer, namely, that one can approximate a differential equation by a system of linear algebraic equations and solve the resulting linear system on the computer to obtain an approximate solution to the original problem. The derivatives in the differential equation are approximated by difference quotients which are obtained from truncated Taylor series in many cases. This entire procedure goes by the name "finite difference methods."

In this discussion we will be primarily concerned with linear second-order partial differential equations of the form

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu = G \qquad (1)$$

where A, B, C, D, E, F, and G are given functions of x and y which are continuous in some region R in the (x,y) plane. A typical problem is the following: given a region R with boundary S, find a function u(x,y) which is twice differentiable and satisfies (1) in R, which is continuous in R + S and which satisfies prescribed conditions on S.

Let us consider the generalized Dirichlet problem involving a bounded and connected region R and a continuous function g(x,y) prescribed on S. The function u(x,y) is required to be continuous in R + S, to satisfy (1) in R and to satisfy u(x,y) = g(x,y) for $(x,y) \in S$. If (1) is an "elliptic equation" ($B^2 - AC < 0$ in R) and if $F \leq 0$ in R, then the generalized Dirichlet problem has a unique solution under fairly general conditions. A special case of the generalized Dirichlet problem is a classical problem in applied mathematics—the Dirichlet problem involving Poisson's

$$u_{xx} + u_{yy} = G\ .$$

This problem is used as a "model problem" in both theory and applications.

In principle, we can make a change of independent variables so that the coefficient B of the mixed derivative u_{xy} in (1) vanishes identically in R + S. (See Young and Gregory [1973].)

The method of finite differences involves superimposing a mesh over the region R and replacing the differential equation (1) by a difference equation at each mesh point. In this discussion we assume that a square mesh of horizontal and vertical lines with mesh spacing h in both the horizontal and vertical directions is used. At a mesh point (x,y) we replace the partial derivatives appearing in (1) by standard three-point central difference quotients

$$\begin{cases} u_x \cong [u(x+h,y) - u(x-h,y)]/(2h) \\ u_y \cong [u(x,y+h) - u(x,y-h)]/(2h) \\ u_{xx} \cong [u(x+h,y) + u(x-h,y) - 2u(x,y)]/h^2 \\ u_{yy} \cong [u(x,y+h) + u(x,y-h) - 2u(x,y)]/h^2 \end{cases}$$

Substituting in (1) with B = 0 and multiplying by $-h^2$, we obtain the difference equation

$$\begin{aligned} &a_0u(x,y) - a_1u(x+h,y) - a_2u(x,y+h) \\ &- a_3u(x-h,y) - a_4u(x,y-h) = t(x,y) \end{aligned} \qquad (2)$$

where

$$\begin{cases} a_1 = A(x,y) + \frac{h}{2}D(x,y)\ , & a_3 = A(x,y) - \frac{h}{2}D(x,y) \\ a_2 = C(x,y) + \frac{h}{2}E(x,y)\ , & a_4 = C(x,y) - \frac{h}{2}E(x,y) \\ a_0 = a_1 + a_2 + a_3 + a_4 - h^2F(x,y), & t(x,y) = -h^2G(x,y) \end{cases}$$

The difference equation (2) is valid if the point (x,y) and the four adjacent points (x ± h,y), (x,y ± h) are in R + S. Special formulas are available for more general situations including cases involving curved boundaries. (See, e.g., Forsythe and Wasow [1960], Young and Gregory [1973], or Birkhoff and Lynch [1984].)

CH2345-7/86/0000/0556$01.00 © 1986 IEEE

In the case where B = 0 and (1) is "self-adjoint", i.e., $D = A_x$, $E = C_y$, we can write (1) in the form

$$(Au_x)_x + (Cu_y)_y + Fu = G \tag{3}$$

For this equation, we can derive a difference equation which has the same accuracy as (2) and such that the resulting linear system is symmetric. This is done by replacing $(Au_x)_x$ and $(Cu_y)_y$ by the difference quotients

$$(Au_x)_x \cong \{A(x+\frac{h}{2},\ y)[u(x+h,y) - u(x,y)] - A(x-\frac{h}{2},\ y)[u(x,y) - u(x-h,y)]\}/h^2$$

and

$$(Cu_y)_y \cong \{C(x,y+\frac{h}{2})[u(x,y+h) - u(x,y)] - C(x,y-\frac{h}{2})[u(x,y) - u(x,y-h)]\}/h^2$$

Substituting in (3) and multiplying by $-h^2$

$$\begin{aligned} &a_0u(x,y) - a_1u(x+h,y) - a_2u(x,y+h) \\ &\quad - a_3u(x-h,y) - a_4u(x,y-h) = t(x,y) \end{aligned} \tag{4}$$

where

$$\begin{cases} a_1 = A(x+\frac{h}{2},\ y)\ , & a_3 = A(x-\frac{h}{2},\ y) \\ a_2 = C(x,y+\frac{h}{2})\ , & a_4 = C(x,y-\frac{h}{2}) \\ a_0 = a_1 + a_2 + a_3 + a_4 - h^2F(x,y), & t(x,y) = -h^2G(x,y) \end{cases}$$

If (1) is not self-adjoint and B = 0, it may be possible to obtain a self-adjoint equation by multiplying (1) by an "integrating factor" $\mu(x,y)$. In this case, (1) is "essentially self-adjoint" and the following condition must hold

$$\frac{\partial}{\partial y}\left(\frac{D - A_x}{A}\right) = \frac{\partial}{\partial x}\left(\frac{E - C_y}{C}\right)$$

Thus for example the equation $u_{xx} - x^{-1}u_x + u_{yy} = 0$ is essentially self-adjoint and can be made self-adjoint by multiplying by the integrating factor $\mu(x,y) = x^{-1}$ obtaining $(x^{-1}u_x)_x + (x^{-1}u_y)_y = 0$.

As an example, we consider the difference equation corresponding to the Poisson equation

$$u_{xx} + u_{yy} = G(x,y) \tag{5}$$

where G(x,y) is a given function. Evidently, we have A = C = 1, B = D = E = F = 0, and

$$\begin{cases} a_1 = a_2 = a_3 = a_4 = 1 \\ a_0 = 4, \quad t(x,y) = -h^2G(x,y) \end{cases}$$

If we wish to solve this problem on the unit square $0 \leqq x \leqq 1$, $0 \leqq y \leqq 1$ where the values on the boundary are given by $u(x,y) = g(x,y)$, then a uniform square mesh with spacing h can be established over the region. With this mesh subdivision, the finite difference approximation to Poisson's equation at each interior grid point (x_i,y_i) is an equation of the form

$$\begin{aligned} 4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} \\ = -h^2G(x_i,y_i) \end{aligned} \tag{6}$$

using either (2) or (4). Here $u_{ij} = u(x_i,y_j)$ and $x_i = x_0 + ih$, $y_i = y_0 + jh$ for $0 \leqq i \leqq m+1$, $0 \leqq j \leqq m+1$ and $h = 1/(m+1)$.

Let us now consider the case where $h = \frac{1}{4}$ which leads to nine linear equations—one for each interior grid point. We label the mesh points as indicated in Figure 1.

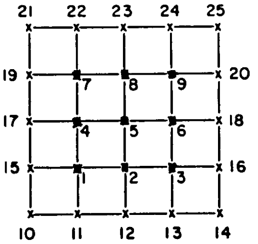


Figure 1. Natural ordering for point partitioning.

The interior points are numbered first in the "natural" order and then the boundary points are numbered. From (6) we obtain, after transferring known values to the right-hand side

$$\begin{bmatrix} 4 & -1 & 0 & -1 & & & & & \\ -1 & 4 & -1 & 0 & -1 & & & 0 & \\ 0 & -1 & 4 & 0 & 0 & -1 & & & \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & & \\ & -1 & 0 & -1 & 4 & -1 & 0 & -1 & \\ & & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ & & & -1 & 0 & 0 & 4 & -1 & 0 \\ & 0 & & & -1 & 0 & -1 & 4 & -1 \\ & & & & & -1 & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \end{bmatrix} = \begin{bmatrix} g_{11} + g_{15} - h^2G_1 \\ g_{12} - h^2G_2 \\ g_{13} + g_{16} - h^2G_3 \\ g_{17} - h^2G_4 \\ - h^2G_5 \\ g_{18} - h^2G_6 \\ g_{19} + g_{22} - h^2G_7 \\ g_{23} - h^2G_8 \\ g_{20} + g_{24} - h^2G_9 \end{bmatrix} \tag{7}$$

Here we have let the k^{th} component u_k be the unknown corresponding to the mesh point marked k. Similarly, g_k and G_k are the values of these functions at the labelled mesh points in Figure 1.

This system can be written in matrix form as

$$Au = b \tag{8}$$

Notice the simple pattern of 0's in the coefficient matrix (7) with this natural ordering. We can write the system as

$$\begin{bmatrix} A_{1,1} & A_{1,2} & 0 \\ A_{2,1} & A_{2,2} & A_{2,3} \\ 0 & A_{3,2} & A_{3,3} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix}, \tag{9}$$

where

$$A_{i,i} = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix}$$

and

$$A_{i,i+1} = A_{i+1,i} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} .$$

Using Figure 2, the unknowns for line k are denoted U_k and we have a grouping according to lines.

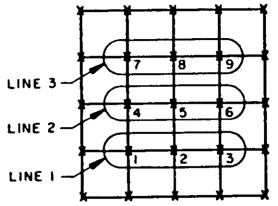


Figure 2. Natural ordering for line partitioning.

The submatrix A_{ij} gives the coupling of the unknowns from line i to those on line j. Since the coefficient matrix is symmetric, we have $A_{ij} = A_{ji}^T$.

Another ordering of the interior mesh points is the "red/black ordering" in which every other point is given either a red or black label as on a checkerboard. (See Figure 3.)

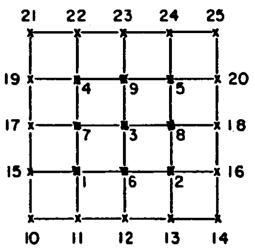


Figure 3. Red/Black ordering for point partitioning.

For this ordering of the unknowns, the difference equations (6) for each interior grid point can be expressed in the matrix form

$$\begin{bmatrix} 4 & & & & & -1 & -1 & 0 & 0 \\ & 4 & & 0 & & -1 & 0 & -1 & 0 \\ & & 4 & & & -1 & -1 & -1 & -1 \\ 0 & & & 4 & & 0 & -1 & 0 & -1 \\ & & & & 4 & 0 & 0 & -1 & -1 \\ -1 & -1 & -1 & 0 & 0 & 4 & & & \\ -1 & 0 & -1 & -1 & 0 & & 4 & 0 & \\ 0 & -1 & -1 & 0 & -1 & & 0 & 4 & \\ 0 & 0 & -1 & -1 & -1 & & & & 4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \end{bmatrix} = \begin{bmatrix} g_{11} + g_{15} - h^2G_1 \\ g_{13} + g_{16} - h^2G_2 \\ - h^2G_3 \\ g_{19} + g_{22} - h^2G_4 \\ g_{20} + g_{24} - h^2G_5 \\ g_{12} \quad - h^2G_6 \\ g_{17} \quad - h^2G_7 \\ g_{18} \quad - h^2G_8 \\ g_{23} \quad - h^2G_9 \end{bmatrix} . \tag{10}$$

The red unknowns u_1, u_2, u_3, u_4, u_5 can be grouped in a vector u_R and the black unknowns u_6, u_7, u_8, u_9 in a vector u_B. We can write such a system in a red/black partitioned form

$$\begin{bmatrix} D_R & H \\ K & D_B \end{bmatrix} \begin{bmatrix} u_R \\ u_B \end{bmatrix} = \begin{bmatrix} F_R \\ F_B \end{bmatrix} \tag{11}$$

where D_R and D_B are diagonal matrices and $K = H^T$.

A red/black ordering for the line partitioning corresponding to the numbering given in Figure 4 would have the red lines of unknowns in variables U_1 and U_2 while the black lines unknowns are in U_3. The resulting partitioned system is of the form

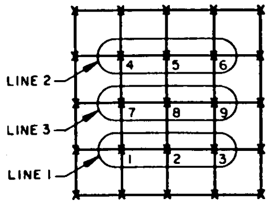


Figure 4. Red/Black ordering for line partitioning.

$$\begin{bmatrix} A_{1,1} & 0 & A_{1,3} \\ 0 & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix}. \tag{12}$$

Now if we partition by red and black lines by letting $u_R = [U_1, U_2]^T$ and $u_B = U_3$ then we obtain the red/black partitioned system (11) where

$$D_R = \begin{bmatrix} A_{11} & 0 \\ 0 & A_{22} \end{bmatrix} , \qquad D_B = A_{33} ,$$

$$H = \begin{bmatrix} A_{13} \\ A_{23} \end{bmatrix}, \quad K = [A_{31} \quad A_{32}]$$

In the point red/black ordering, D_R and D_B are diagonal matrices whereas in the line red/black ordering they are block diagonal.

The solution of a difference equation involves the solution of the system (8). To obtain an accurate solution of the differential equation it is often necessary to use a very small mesh size h. This results in a large system of equations. These can be solved numerically using direct methods or using iterative methods (see, e.g., Varga [1962], Wachspress [1966], Young [1971], Hageman and Young [1981], and Birkhoff and Lynch [1984]). In each case special procedures should be used to take advantage of the fact that the coefficient matrix of the linear system is "sparse", i.e., that most of the elements are zero. Frequently-used iterative methods include the successive overrelaxation method, the Jacobi method and the incomplete Cholesky method. The latter two methods are often combined with an acceleration procedure such as Chebyshev acceleration or conjugate gradient acceleration to speed up the convergence.

Iterative methods are ideal for such systems since they produce no "fill-in". Partitioning such as (9) and (11) can be used to advantage in special iterative methods for solving the linear system. Iterative methods which utilize the unique structure of these partitioning require that the subsystems

$$A_{ii}U_i = \hat{F}_i, \quad D_R u_R = \hat{F}_R, \quad D_B u_B = \hat{F}_B \tag{13}$$

can be "easily" solved since this must be done repeatedly for every iteration—the "inner-iteration" step.

If the coefficients appearing in the differential equation are sufficiently well-behaved and if the region R is sufficiently "regular" then the accuracy of the solution of the difference equations (2) or (4) is of order h^2. Greater accuracy can be achieved in some cases by the use of higher order difference equations. The use of a 9-point stencil $(x + \sigma h, y + \tau h)$ where $\sigma, \tau = -1, 0, 1$ can sometimes be used. It has the advantage that for many problems the same stencil can be used throughout the region. Other stencils can be used such as the 9-point stencil $(x + \sigma h, y + \tau h)$ where $\sigma = -2, -1, 0, 1, 2$ and $\tau = 0$ or $\sigma = 0$ and $\tau = -2, -1, 0, 1, 2$.

Finite difference methods may be used to solve problems involving more general boundary conditions such as

$$\alpha(x,y)u + \beta(x,y) \frac{\partial u}{\partial n} = \gamma(x,y)$$

where α, β, and γ are given functions defined on R. In the case of curved boundaries the procedure for representing the boundary conditions can be difficult. For points on a horizontal or vertical boundary, simpler procedures are available. A procedure for deriving difference equations at boundary points and at interior points, even when some of the coefficients of the differential equation are discontinuous, is given in Varga [1962].

We wish to thank Thomas C. Oppe and John R. Respess for reading this paper and making suggestions.

REFERENCES

[1] G. Birkhoff and R. E. Lynch, Numerical Solution of Elliptic Problems, SIAM, Philadelphia, PA., 1984.

[2] G. E. Forsythe and W. R. Wasow, Finite Difference Methods for Partial Differential Equations, Wiley, New York, N.Y., 1960.

[3] L. A. Hageman and D. M. Young, Applied Iterative Methods, Academic Press, New York, N.Y., 1981.

[4] R. S. Varga, Matrix Iterative Analysis, Prentice Hall, Englewood Cliffs, N.J., 1962.

[5] E. L. Wachspress, Iterative Solution of Elliptic Systems, Prentice Hall, Englewood Cliffs, N.J., 1966.

[6] D. M. Young, Iterative Solution of Large Linear Systems, Academic Press, New York, N.Y., 1971.

[7] D. M. Young and R. T. Gregory, A Survey of Numerical Mathematics, Volume II, Addison-Wesley, Reading, MA., 1973.

FINITE ELEMENT METHODS

J. Tinsley Oden

Texas Institute for Computational Mechanics
The University of Texas at Austin, Austin, TX 78712

Abstract. This note summarizes the features of finite element methods that are responsible for its great success as a tool for the numerical solution of partial differential equations. Arguments are also given as to why the method will continue to be dominant in the future development of numerical schemes for solutions of complex boundary and initial value problems.

Finite elements; perhaps no other family of approximation methods has had a greater impact on the theory and practice of numerical methods during the twentieth century. Finite element methods have now been used in virtually every conceivable area of engineering that can make use of models of nature characterized by partial differential equations.

A natural question that one may ask is: why have finite element methods been so popular in both the engineering and mathematical community? There is also the question, do finite element methods possess properties that will continue to make them attractive choices of methods to solve difficult problems in physics and engineering?

In answering these questions, one must first point to the fact that finite element methods are based on the weak or variational formulation of boundary and initial value problems. This is a critical property, not only because it provides a proper setting for the existence of very irregular solutions to differential equations (e.g. distributions), but also because the solution appears in the integral of a quantity of a domain. The simple fact that the integral of a bounded measurable function over an arbitrary domain can be broken up into the sum of integrals over an arbitrary collection of almost disjoint subdomains whose union is the original domain, is in fact a vital observation in finite element theory. Because of it, the analysis of a problem can literally be made locally, over a typical subdomain, and by making the subdomain sufficiently small one can argue that polynomial functions of various degrees are adequate for representing the local behavior of the solution. This summability of integrals is exploited in every finite element program. It allows the analyst to focus his attention on a typical finite element domain and to develop an approximation independent of the ultimate location of that element in the final mesh.

This simple property has important implications in physics and in most problems in continuum mechanics. Indeed, the classical balance laws of mechanics are global, in the sense that they are integral laws applying to a given mass of material, a fluid or solid. From the onset, only regularity of the primitive variables sufficient for these global conservation laws to make sense is needed. Moreover, since these laws are supposed to be fundamental axioms of physics, they must hold over every finite portion of the material: every finite element of the continuum. Thus once again, one is encouraged to think of approximate methods defined by integral formulations over typical pieces of a continuum to be studied.

These rather primitive properties of finite elements lead to some of their most important features:

1) Arbitrary geometries. The method is essentially geometry-free. In principle, finite element methods can be applied to domains of arbitrary shape and with quite arbitrary boundary conditions.

2) Unstructured meshes. While there is still much prejudice in the numerical analysis literature toward the use of coordination-dependent algorithms and mesh generators, there is nothing intrinsic in finite element methodology that requires such devices. Indeed, finite element methods by their nature lead to unstructured meshes. This means that, in principle, the analyst can place finite elements anywhere he pleases. He may thus model the most complex types of geometries in nature and physics, ranging from the complex cross sections of biological tissues to the exterior of aircraft to internal flows in turbo machinery, without strong use of a global fixed coordinate frame.

3) Robustness. It is well known that in finite element methods the contributions of local approximations over individual elements are assembled together in a systematic way to arrive at a global approximation of a solu-

CH2345-7/86/0000/0560$01.00 © 1986 IEEE

tion to a partial differential equation. Generally this leads to schemes which are stable in appropriate norms and, moreover, insensitive to singularities or distortions of the mesh. There are notable exceptions to this, of course, and these exceptions have been the subject of some of the most important works in finite element theory. But, by and large, the direct use of Galerkin or Petrov-Galerkin methods to derive finite element methods leads to conservative and stable algorithms for most classes of problems in mechanics and mathematical physics.

4) Mathematical foundation. Because the extensive work on the mathematical foundations done during the seventies and eighties, finite element methods now enjoy a rich and solid mathematical basis. The availability of methods to determine a-priori and a-posteriori estimates provide a vital part of the theory of finite elements, and make it possible to lift the analysis of important engineering and physical problems above the traditional empiricism prevalent in many numerical and experimental studies.

When one attempts to project into the future trends in numerical methods for partial differential equations, it makes sense to pose some fundamental questions in this subject. We feel that the issue of optimality of numerical methods is central, and this issue is embodied in two questions: how good are the answers one obtains from a numerical analysis of a model of a physical phenomenon?; how can one obtain the best possible answers for a fixed level of computational cost or effort? -

The answer to the first is mathematical in essence: determine a-posteriori error estimates. The answer to the second question is computational: knowing an estimate of the error, change the structure of the approximation to reduce it in an efficient way. The implementation of such an optimal numerical strategy thus requires several basic properties of the methods in use: a rich mathematical basis since sharp error estimates must be obtained, a structure independent of geometry and coordinates, since the mesh and mesh properties must be dynamically changed to reduce error; robustness, since the method must be stable under changes in structure; and it must admit to supercomputing strategies. We feel that finite element methods represent one of the very few - and perhaps the only - class of approaches that can fulfill these requirements.

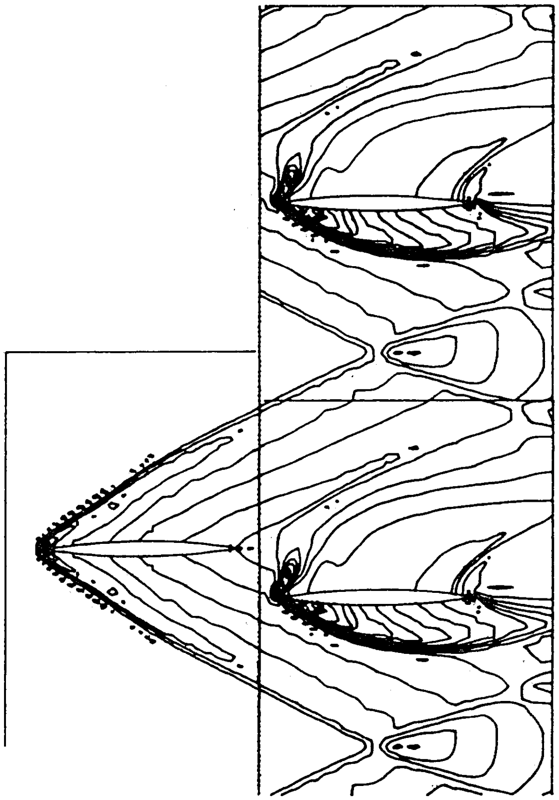


Fig. 1.

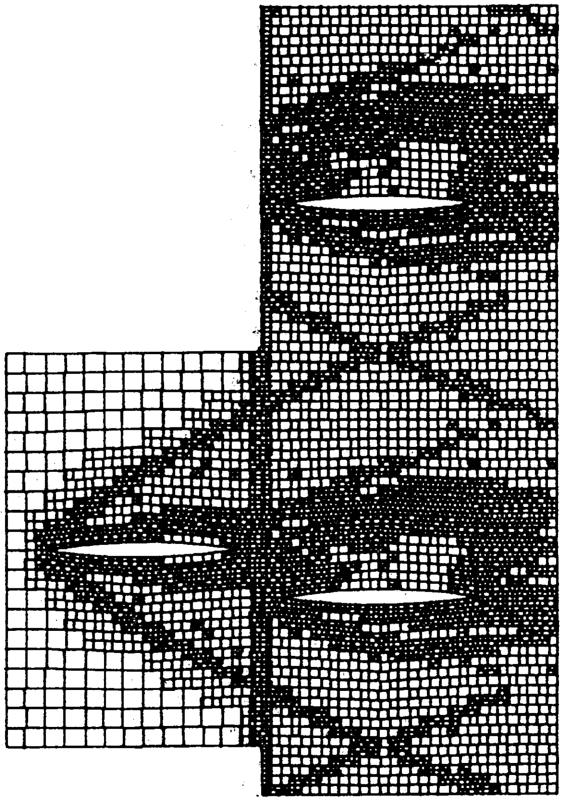


Fig. 2.

An example of the great adaptability of finite element methods is shown in Figs. 1 and 2. There one observes a calculation of supersonic flow of a compressible gas flowing through a set of spinning turbine blades. The finite element algorithm tests local errors and automatically refines the mesh to attempt to produce the best possible accuracy using the least number of elements; this being done dynamically as the phenomenon evolves in time. Larger elements are used where the error is small. Also shown are computed pressure contours which exhibit the evolution of shock and shock interaction.

For the above reasons, it is certain that finite element concepts will continue to occupy a dominant role in applications and in research on the numerical solution of partial differential equations.

BOUNDARY ELEMENT METHODS

S. R. KENNON*

* Graduate Research Assistant, Dept. of Aerospace Engineering and Engineering Mechanics, University of Texas at Austin, Austin, TX.

Abstract

The boundary element method (BEM) is surveyed and shown to be a powerful tool for the accurate and efficient solution of a certain class of field problems. We discuss the application of the method to an important model problem, discuss some computational aspects of the technique, and point out some advantages and disadvantages of the method. Finally, some applications of the BEM are discussed, with particular reference to inverse design problems. A short bibliography of relevant literature is included.

Introduction

The boundary element method (BEM)[1,2] is a powerful sub-class of finite element techniques for the accurate and efficient solution of a wide class of problems posessing a special structure. The method, where applicable, can give more accurate solutions to certain problems than finite difference or finite element methods. Moreover, the method can in most cases provide an order of magnitude increase in efficiency over finite difference or finite element methods. The gain in efficiency and ease of use is due to the fact that solution of the partial differential equation in the entire domain is reduced to the solution of an integral equation on the boundary of the domain. Therefore, only the boundary of the domain being studied need be discretized (this is especially important in complex, multiply connected domains). Thus in two-dimensional problems, only a one-dimensional domain (e.g. the boundary) need be discretized. In addition, since exact, analytic solutions of the governing partial differential equation are used in the approximate solution, the BEM is inherently more accurate. For these reasons, the method is particularly appealing in either time-dependent or inverse design problems since a field problem needs to be solved at either every time-step or every design iteration.

This paper gives the basic concepts underlying the method and shows how it is applied to an important model problem. Next we show some standard applications of the method, discuss the BEM for non-linear problems, and finally show how the BEM can be used for very efficient inverse design.

A Model Problem[1]

To present the power and ease of application of the BEM we will consider the solution of the Dirichlet/Neumann problem in a two-dimensional domain Ω (Fig. 1). The problem can be stated mathematically as

P1: Find the function $u(x,y)$ such that

$$\Delta u = 0 \quad \text{in } \Omega$$

$$u = \underline{u} \quad \text{on } \Gamma_1$$

$$q = \partial u/\partial n = \underline{q} \text{ on } \Gamma_2 \qquad (1)$$

where $\underline{u}$ and $\underline{q}$ are given functions, $\Gamma = \Gamma_1 + \Gamma_2 = \partial\Omega$ the boundary of Ω and Δ is the Laplacian operator ($\Delta u = \partial^2 u/\partial x^2 + \partial^2 u/\partial y^2$). This is the standard potential problem and is the governing equation for such physical problems as heat and/or electrical conduction in homogeneous media and ideal fluid flow.

The Weighted Residual Statement

In the BEM, we use a weighted residual statement of the problem. The weighted residual statement is achieved by multiplying the governing partial differential equation with a sufficiently smooth weighting function $u^*(x,y)$ and integrating the product over the domain. In addition, we weight the boundary conditions giving the following integral statement of the problem:

P2: Find $u(x,y)$ such that

$$\int_\Omega (\Delta u)u^* \, d\Omega = \int_{\Gamma_2} (q - \underline{q})\, u^* \, d\Gamma \; - \int_{\Gamma_1} (u - \underline{u}) q^* \, d\Gamma \qquad (2)$$

for all admissible u^* and q^*. Problem P2 is equivalent to problem P1. Now we can integrate the left-hand-side of eq. 2 by parts to obtain the following equivalent equation

$$-\int_\Omega \Sigma_k (\partial u/\partial x_k \; \partial u^*/\partial x_k)\, d\Omega \;=\; -\int_{\Gamma_2} \underline{q}\, u^* \, d\Gamma$$
$$-\int_{\Gamma_2} q u^* \, d\Gamma \; - \int_{\Gamma_1} u q^* \, d\Gamma + \int_{\Gamma_1} \underline{u} q^* \, d\Gamma \qquad (3)$$

We can integrate by parts once more and obtain the fundamental equation of the BEM applied to the potential problem

$$-\int_\Omega (\Delta u^*) u \, d\Omega = -\int_{\Gamma_2} \underline{q}\, u^* \, d\Gamma - \int_{\Gamma_2} q u^* \, d\Gamma$$
$$-\int_{\Gamma_1} u q^* \, d\Gamma + \int_{\Gamma_1} \underline{u} q^* \, d\Gamma \qquad (4)$$

In the above, we have made use of the Gauss divergence theorem to relate integrals over the domain Ω to equivalent integrals over the boundary $\partial\Omega$, which is the key to the BEM (the Gauss Divergence formula states that for sufficiently

CH2345-7/86/0000/0563$01.00 © 1986 IEEE

smooth functions u(x,y), $\int_{\Omega} \Delta u = \int_{\partial\Omega} \partial u/\partial n \, d\Gamma$ where n is the direction normal to the boundary $\partial\Omega$).

The next step is to introduce the fundamental solution to the Laplace equation. Assume that a point charge (or source) is acting at point $\mathbf{x}_i = (x_i, y_i)$ in Ω. Then the fundamental solution u* is required to be a solution of the problem

$$\text{P3:} \quad \Delta u^* + \delta(x - x_i) = 0 \ \text{ in } \Omega \tag{5}$$

where $\delta(x)$ is the Dirac delta distribution. For two dimensions, the fundamental solution is

$$u^* = (1/2\pi) \ln |x - x_i| \tag{6}$$

This solution has the property that

$$\int_{\Omega} (\Delta u^* + \delta(x - x_i))u \, d\Omega = \int_{\Omega} (\Delta u^*)u \, d\Omega + u(x_i) \tag{7}$$

If u* satisfies eq. 5 then

$$\int_{\Omega} (\Delta u^*)u \, d\Omega = -u(\mathbf{x}_i) \tag{8}$$

and the fundamental equation 4 becomes

$$u(\mathbf{x}_i) + \int_{\Gamma_1} uq^* \, d\Gamma + \int_{\Gamma_1} \underline{u}q^* \, d\Gamma = \int_{\Gamma_2} \underline{q}u^* \, d\Gamma + \int_{\Gamma_2} qu^* \, d\Gamma \tag{9}$$

Next, we let $\mathbf{x}_i$ approach the boundary and through a limiting procedure obtain the final boundary integral statement of P2:

$$c_i \, u(x_i) + \int_{\Gamma} uq^* \, d\Gamma = \int_{\Gamma} qu^* \, d\Gamma \quad (x_i \text{ is on } \partial\Omega) \tag{10}$$

where $\Gamma = \Gamma_1 + \Gamma_2 = \partial\Omega$ and we assumed that $u = \underline{u}$ on Γ_1 and $\partial u/\partial n = q = \underline{q}$ on Γ_2. Here c_i is a constant that depends on the smoothness of the boundary (if Γ is smooth then $c_i = 1/2$). Equation 10 is the basic equation to be approximated by the BEM.

Discretization

Notice that eq. 10 only involves quantities on the boundary of the domain in question. Thus we can introduce a discretization of the boundary as shown in figure 2. Here we show several possibilities in which the solution is assumed to be piecewise constant (2a) piecewise linear (2b) and piecewise quadratic (2c) on the boundary. The unknown solution u(s) and q(s) will be approximated at the nodal points s_j where s is a point on the boundary Γ. One can introduce linearly independent basis functions $\varphi_j(s)$ on the boundary Γ that are piecewise constant, linear, quadratic or higher order. Then the approximate solution u and the weighting functions u* and q* can be represented as $u = \Sigma_j u_j \varphi_j$, $u^* = \Sigma_j u_j^* \varphi_j$, and $q^* = \Sigma_j q_j^* \varphi_j$, respectively where $u_j = u(s_j)$. The interpolants φ_j have the property that $\varphi_i(s_j) = \delta_{ij}$, where δ_{ij} is the Kronecker delta. Introducing these approximations into eq. 10 yields (for piecewise constant elements)

$$c_i \, u(x_i) + \Sigma_j u_j \int_{\Gamma_j} q^* \, d\Gamma = \Sigma_j q_j \int_{\Gamma_j} u^* \, d\Gamma \tag{11}$$

The integrals appearing in eq. 11 can be approximated using numerical integration formulas (numerical quadrature). Now, this equation can be written as a system of linear equations

$$\mathbf{Hu} = \mathbf{Gq} \tag{12}$$

where **H** and **G** are n x n matrices (n is the number of nodal points discretizing Γ) and **u** and **q** are n-vectors. Assuming that Γ_1 is discretized by n_1 elements and Γ_2 is discretized by n_2 elements where $n = n_1 + n_2$ we therefore know n_1 values of u_j and n_2 values of q_j from the specified boundary conditions. Thus, eq. 11 can be re-written in terms of the vector **y** containing the unknowns u_j and q_j as

$$\mathbf{Ay} = \mathbf{b} \tag{13}$$

where **A** is an n x n matrix and **b** is an n-vector. Once **y** is solved for, the value of u and its derivatives can be calculated at any point in the domain Ω using eq. 11. The power of the boundary element method is now clear. We began with a second order partial differetial equation with fairly complicated boundary conditions, and reduced it to the problem of solving a linear system of equations in a few unknowns that represent the solution values on the boundary of the domain. We must now note some disadvantages of the method. The matrix of the system eq. 13is full and can become ill-conditioned. For problems with many unknowns (e.g. for three-dimensional or complex two-dimensional problems) eq. 13 becomes difficult to solve using direct methods and thus it might be necessary to use iterative techniques to solve it. When solving a non-homogeneous problem (e.g. a Poisson equation), the entire domain needs to be discretized with sub-regions akin to finite elements to integrate the non-homogeneous term. This does not introduce any new unknowns into the problem but does make the computational complexity of the problem higher since the entire domain Ω needs to be discretized. Finally, the BEM is only applicable to linear problems in which an appropriate Green's identity can be used to reduce the problem to a boundary integral statement. Non-linear problems can be treated with special techniques discussed below. Nevertheless, the method can be claimed to be among the best methods for the class of problems that it is applicable.

Examples of Problems Solvable by the BEM

For reference, we list some other types of problems and their associated partial differential equations that can be solved by the BEM[2]:

1) Linear elasticity (Navier's equation)
2) Helmholtz equation
3) Darcy flow equation (flow through a porous medium)
4) Wave equation

Non-Linear Problems

In cases where the problem to be solved can be written in the form

$$Lu = Nu \text{ in } \Omega \text{ (plus boundary conditions)} \qquad (14)$$

where L is a linear operator that can be treated with the BEM and N is a non-linear operator, then the BEM can be applied in an iterative fashion. Essentially, the linear part of the equation is discretized as usual (a weighted integral statement is derived, basis functions are introduced and a linear system of equations is formed). The non-linear part of the equation is treated as a non-homogeneous term that must be evaluated and integrated in the interior of the domain. The procedure results in a discrete system of the following form

$$\mathbf{Ay} = \mathbf{b} + \mathrm{N}(\mathbf{y}) \qquad (15)$$

where N(y) is the non-linear analog of the non-linear part of eq. 14. This equation can be solved, for example, by the following algorithm

```
Algorithm:
y := y^0  an initial guess for the solution
k := 0
while || δ || > ε do
    begin
        k := k + 1
        Solve  Aδ^k = − Ay^k +  b + N(y^k) for δ^k
        y^{k+1} := y^k + δ^k
    end                                                    (16)
```

An example of this type of problem is the full potential equation[3] governing isentropic irrotational flow of a perfect gas:

$$\Delta\phi = N(\phi) \text{ (plus appropriate boundary conditions)} \qquad (17)$$

where $N(\phi)$ is given by

$$[\ \phi_x{}^2\phi_{xx}\ - 2\ \phi_x\phi_y\phi_{xy} + \phi_y{}^2\phi_{yy}\ ]\ /\ [\ c_1 + c_2(\ \phi_x{}^2 + \phi_y{}^2\ )\ ]$$

and c_1 and c_2 are constants. Fig. 3 shows a plot of the surface pressure distribution on an airfoil calculated using the non-linear BEM[3]. The results compare well with other methods.

Inverse Design Applications

As mentioned above, the BEM provides a very efficient method for accurately solving heat conduction problems. We now consider a special application of the BEM for analysis and design of turbine blade coolant flow passage shapes. Turbine blades in gas turbine engines are subjected to extremely high temperatures from the combustor portion of the engine. Jet fuel is burned in the combustor and then the by-products are allowed to expand and accelerate through the turbine section. To keep the blades from melting, cooler air from the compressor portion of the engine is passed through holes in the interior of the blade. The turbine blade designer faces the problem of how many holes to drill in the blade and where to place them.

Ref. 4 presents an application of the BEM to the analysis of heat conduction in turbine blades with holes. Special attention is paid to mixed type boundary conditions and the use of the BEM to determine interior solutions given experimentally determined data on the exterior of the blade.

The problem of blade design can be attacked with the use of the BEM in conjunction with an optimization procedure to iteratively adjust the position and size of the holes so that an optimal design is achieved. In Refs. 5 and 6 this method was developed and tested on some practical turbine design problems. Fig. 4 shows a result of an optimization sequence. This figure shows the initial and subsequent evolution of the position and size and of the holes in a turbine blade. The BEM is used at each iteration to solve the heat conduction problem in the solid portions of the hollow blade. The BEM solution is coupled with an optimization routine to iteratively adjust both the size and position of the holes. The function being minimized in this example is the difference between a specified and calculated heat flux on the exterior surface of the blade. This procedure allows the designer to specify a low enough temperature on the surface of each hole, so that the blade retains its structural integrity and does not melt when subject to hot combustor gases.

Conclusions

We have discussed the BEM and its applications to a model problem. In addition, special applications of the method were detailed. The BEM was shown to be a powerful alternative to finite element or finite difference methods for the solution of boundary value problems in engineering science.

References

[1] Brebbia, C. A. The Boundary Element Method for Engineers, John Wiley & Sons, New York, 1978.

[2] Carey, G. F. and J. T. Oden. Finite Elements: A Second Course, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.

[3] Ravichandran, K. S., N. L. Arora, and R. Singh, "Transonic Full Potenetial Solutions by an Integral Equation Method," AIAA J., 22, No. 7, July, 1984, pp. 882-888.

[4] Nakata, Y. and T. Araki, "Application of Boundary Element Method to Heat Transfer Coefficient Measurements Around a Gas Turbine Blade," presented at ASME Winter Annual Meeting, New Orleans, December 8-13, 1984.

[5] Kennon, S. R. and G. S. Dulikravich, "The Inverse Design of Coolant Flow Passage Shapes with Partially Fixed Internal Geometries," ASME paper no. 85-GT-118, March, 1985.

[6] Chiang, T. L., and G. S. Dulikravich, "Inverse Design of Composite Turbine Blade Circular Coolant Flow Passages," ASME paper no. 86-GT-190, June, 1986.

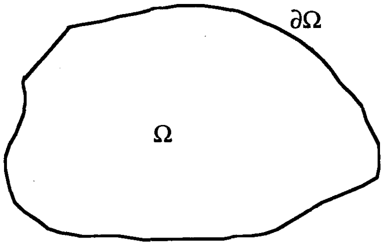


Fig. 1 A two-dimensional domain Ω

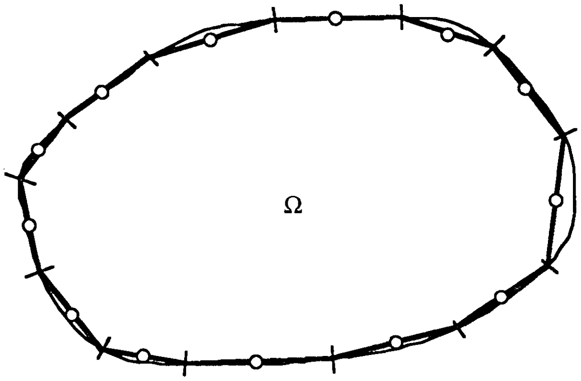


Fig 2a Constant boundary elements showing mid-node locations

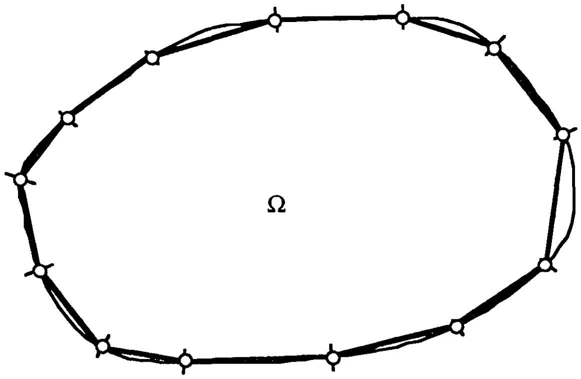


Fig. 2b Linear boundary elements showing end-point locations

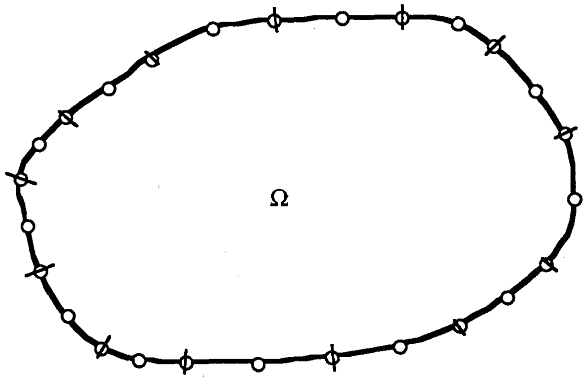


Fig. 2c Quadratic boundary elements showing mid-side nodes

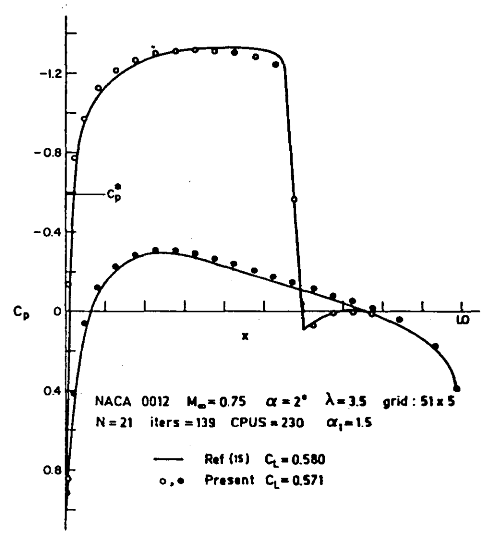


Fig. 3 Coefficient of pressure distribution around a NACA 0012 airfoil calculated using the non-linear BEM.

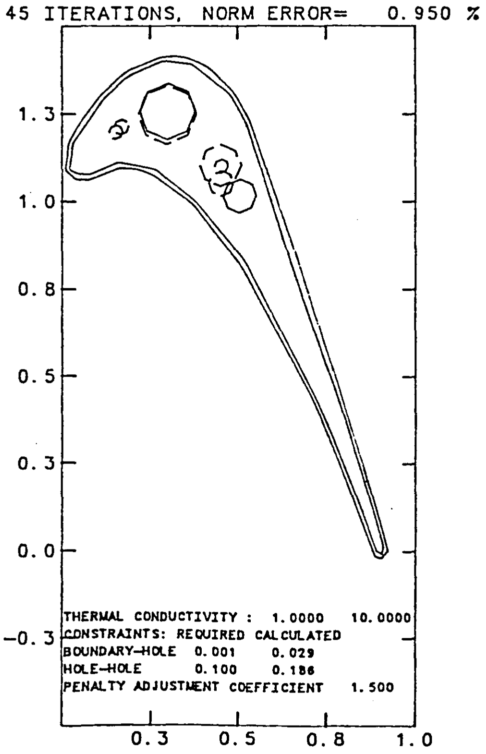

45 ITERATIONS, NORM ERROR= 0.950 %
1.3
1.0
0.8
0.5
0.3
0.0
-0.3
0.3
0.5
0.8
1.0
THERMAL CONDUCTIVITY : 1.0000 10.0000
CONSTRAINTS: REQUIRED CALCULATED
BOUNDARY-HOLE 0.001 0.029
HOLE-HOLE 0.100 0.186
PENALTY ADJUSTMENT COEFFICIENT 1.500

Fig. 4 Example of application of BEM to inverse design: evolution of the design of turbine blade coolant flow passage shapes.

A COMPARISON OF GRID GENERATION TECHNIQUES

S. R. KENNON* - G. S. DULIKRAVICH**

* Graduate Research Assistant, Dept. of Aerospace Engineering and Engineering Mechanics, University of Texas at Austin, Austin, TX.
** Associate Professor, Dept. of Aerospace Engineering, Pennsylvania State University, University Park, PA.

Abstract

A survey of techniques for the numerical generation of computational grids is presented. These methods are shown to be a powerful addition to the numerical analyst's tools for simulation of complex physical processes. Computational grids are classified into their various forms and methods for generating each type of grid are discussed. In addition to the mathematical and numerical details of each method, examples of the application of each method are shown. A short bibliography of these methods is also given to assist the interested reader in finding more information about this important part of computational mechanics.

Introduction

This paper presents a survey of methods for the numerical generation of computational grids used in the simulation of physical problems governed by partial differential equations. Computational grids are required to represent a given region of space as a discrete collection of sub-regions. Within these sub-regions, the solution of the governing partial differential equation is assumed to be of a certain form representable on a computer. The discrete system can then be solved on a computer and the results analyzed. The problem of generating good quality grids is very difficult for complex regions and has been a pacing item in computational mechanics. As grid generation techniques have become more powerful, they have allowed analysts to simulate more and more complex and realistic physical problems. Moreover, with the advent of supercomputers, grid generation methods are now required for discretizing arbitrary three-dimensional regions such as the region surrounding an aircraft in flight, prior to solution of three-dimensional field equations in these domains. Researchers in a wide range of fields have developed some very sophisticated methods for computational grid generation. We begin this survey by first classifying different types of grids as either structured, quasi-structured or unstructured and discuss the different topologies of each. Next we discuss various popular methods for generating grids and give some discussion of their relative merits. Finally we note some special topics including grid improvement methods and adaptive grid generation.

Classification of Computational Grids

We define a computational grid in a domain Ω (subset of R^n) as a collection of M sub-domains Ω_i such that the intersection of Ω_i and Ω_j is empty for all i and j, $i \neq j$, i,j = 1,2,...M and the union of the Ω_i is "close" to Ω. Therefore, we exclude the possibility that any of the Ω_i overlap each other and we ensure that there are no 'gaps' in the grid (except possibly right at the boundary of Ω). Moreover, we do not allow the volume (area in 2-d or length in 1-d) of any of the Ω_i to vanish (see fig. 1). We will be primarily concerned with Ω_i that have boundaries $\partial\Omega_i$ that are representable by polynomials. For example, in a two-dimensional domain we can have straight sided quadrilaterals representable by linear functions of the nodal points. We define nodal points as special points on the boundary of Ω_i that are used to construct the polynomial representing the boundary of Ω_i (fig. 1b) and are normally the points at which the approximate numerical solution is evaluated.

Computational grids can be classified into three main groups: 1) structured, 2) quasi-structured, and 3) unstructured. Structured grids are those that can be logically associated with (mapped, or transformed to) another base grid that has a regular repeatable structure. The base (also called the canonical or computational) domain is constructed entirely of a repetition of the same sub-domain througout the domain. Thus if Ω^* is the base domain, then the base grid is the collection of Ω_i^* where each Ω_i^* has the same size and shape. Examples of different types of base grids are shown in figures 2 a) and b). One can see that a base grid can be defined very easily as a array of the nodal points and a simple rule that determines which nodal points belong to which sub-domains and which elements are connected to each other. Moreover, the discretization of a partial differential equation on the base grid assumes a very simple (essentially canonical) form. For this reason, structured grids have been the most popular grids to date, especially in the finite difference literature. One important aspect of quadrilateral structured grids is that the sides of the elements can be associated with generalized coordinate directions. Thus to generate a quadrilateral structured grid is tantamount to specifying a transformation function that maps the computational space with coordinates $\xi = (\xi_1, \xi_2, ..., \xi_n)$ into the physical domain Ω with coordinates $\mathbf{x} = (x_1, x_2, ..., x_n)$. This transformation can be stated as $\xi = \xi(\mathbf{x})$. This fact is extensively used in finite difference methods where the governing equation in the physical coordinates $\mathbf{x}$ is transformed into the computational coordinates using the map $\xi(\mathbf{x})$ and then the equation is solved in the regular base domain Ω^*. This has the advantage that the finite difference equations are essentially the same at each nodal point and

CH2345-7/86/0000/0568$01.00 © 1986 IEEE

therefore are easy to code in a computer program. Finite element methods generalize this concept with the idea of a master element. The master element is the computational domain and consists of only one sub-domain that can be mapped to every element in the physical domain. The discretized governing equation assumes a canonical form on the master element and therefore, the solution procedure becomes easy to program on a computer.

We define a quasi-structured grid as a grid consisting of a union of structured grids. The union is required to discretize the domain as we require for sub-domains. These grids are used in cases where there does not exist a simple transformation that defines a structured grid (for example in multiply connected domains). Thus the domain to be discretized is broken into a few sub-domains that each form a structured grid. These grids are sometimes termed composite or zonal grids and are used extensively in finite difference methods.

Finally, we define unstructured grids as those that are neither structured nor quasi-structured. These types of grids have been used extensively in finite element methods, although the grids were mostly generated by hand. Only fairly recently have automatic unstructured grid generation methods become common.

Transformations and Computational Coordinates

Structured grid generation methods are based on the existence of a transformation from the physical domain Ω to the computational domain Ω^*. We now show how derivatives of quantities in the physical space transform into their respective derivatives in the computational space (we show this for a two-dimensional domain only). Assume we want to determine the derivatives $\partial/\partial x$ and $\partial/\partial y$ of some function u(x,y) in terms of derivatives $\partial/\partial\xi$ and $\partial/\partial\eta$ in the computational coordinates (ξ,η). Using the chain rule of partial differentiation we find

$$\begin{aligned}\partial u/\partial x &= \partial\xi/\partial x\, \partial u/\partial\xi + \partial\eta/\partial x\, \partial u/\partial\eta \\ \partial u/\partial y &= \partial\xi/\partial y\, \partial u/\partial\xi + \partial\eta/\partial y\, \partial u/\partial\eta\end{aligned} \tag{1}$$

This can be written as a matrix equation

$$\begin{bmatrix}\partial u/\partial x \\ \partial u/\partial y\end{bmatrix} = \begin{bmatrix}\partial\xi/\partial x & \partial\eta/\partial x \\ \partial\xi/\partial y & \partial\eta/\partial y\end{bmatrix}\begin{bmatrix}\partial u/\partial\xi \\ \partial u/\partial\eta\end{bmatrix} \tag{2}$$

We can similarly use the chain rule on the derivatives $\partial u/\partial\xi$ and $\partial u/\partial\eta$ to obtain the relations

$$\begin{bmatrix}\partial u/\partial\xi \\ \partial u/\partial\eta\end{bmatrix} = \begin{bmatrix}\partial x/\partial\xi & \partial y/\partial\xi \\ \partial x/\partial\eta & \partial y/\partial\eta\end{bmatrix}\begin{bmatrix}\partial u/\partial x \\ \partial u/\partial y\end{bmatrix} \tag{3}$$

The matrix can be inverted to obtain the relation

$$\begin{bmatrix}\partial u/\partial x \\ \partial u/\partial y\end{bmatrix} = \frac{1}{J}\begin{bmatrix}\partial y/\partial\eta & -\partial y/\partial\xi \\ -\partial x/\partial\eta & \partial x/\partial\xi\end{bmatrix}\begin{bmatrix}\partial u/\partial\xi \\ \partial u/\partial\eta\end{bmatrix} \tag{4}$$

where J is the Jacobian and is given by $J(\xi,\eta) = \partial x/\partial\xi\, \partial y/\partial\eta - \partial y/\partial\xi\, \partial x/\partial\eta$. Equating coefficients of the two matrices in eq. 2 and 4 gives the following relationships:

$$\begin{aligned}\partial\xi/\partial x &= (\partial y/\partial\eta)/J & \partial\eta/\partial x &= (-\partial y/\partial\xi)/J \\ \partial\xi/\partial y &= (-\partial x/\partial\eta)/J & \partial\eta/\partial y &= (\partial x/\partial\xi)/J\end{aligned} \tag{5}$$

These are the required relations that determine how derivatives of a function in physical space transform into derivatives in the computational space. We now discuss some techniques for the generation of structured grids.

Conformal Mapping Methods

Conformal mapping methods are based on the use of complex analytic conformal transformations. That is, the map from computational to physical space is accomplished with the use of complex valued analytic functions. If we let $z = x + iy$ $(i = \sqrt{-1})$ be the physical complex-valued coordinate and $\zeta = \xi + i\eta$ be the computational complex-valued coordinate then grids are determined by specifying analytic transformations from ζ to z:

$$z = F(\zeta) \tag{6}$$

Since all analytic transformations are conformal (except at singular points) then these transformations will produce conformal maps. A conformal transformation is one that preserves both the magnitude and the sense of angles. Thus, if the base grid consists of mutually orthogonal lines, then the physical grid will also be orthogonal. This fact is an advantage for conformal mapping techniques because if the physical grid is orthogonal, the resulting discretization of the partial differential equation becomes greatly simplified, and truncation errors are reduced. However, it is most usual to use conformal mappings to map the physical space to a simpler non-orthogonal domain and then use simple algebraic transformations to map the intermediate domain to the computational grid. This makes the resulting physical grid non-orthogonal. An example of a nearly orthogonal grid generated using a conformal mapping is shown in fig. 3.

Conformal mapping methods (like the algebraic methods discussed below) are extremely efficient and thus were widely used in the past[1,2,3].

Algebraic Methods[1]

The algebraic grid generation method is based on interpolation of known values of the nodal values of the physical grid on the boundary $\partial\Omega$ throughout the domain. We begin with a short discussion of one-dimensional interpolation. Assume that we have a function $r(\xi)$ that we know the values of at some points ξ_i. We wish to interpolate this function with a simple interpolant that posesses certain desirable properties. We construct interpolants $\varphi_i(\xi)$ that have the property that $\varphi_i(\xi_j) = \delta_{ij}$ (where δ_{ij} is the Kronecker delta). Thus, we can write

$$r(\xi) = \Sigma_i\, r(\xi_i)\ \varphi_i(\xi) \qquad (7)$$

where we assume that ξ varies from 0 to 1. The functions $\varphi_i(\xi)$ can take many forms, but it is most usual to use polynomial interpolants such as the Lagrange or Hermite polynomials. It is also possible to use cubic splines, B-splines or other types of spline functions for the uni-directional interpolation. Now assuming that we know the values of the grid point coordinates on the boundary of the transformed space, that is we know

$$\begin{aligned} &\mathbf{r}(\xi,\eta)\ \text{ for } \eta = 0 \text{ or } 1\\ &\mathbf{r}(\xi,\eta)\ \text{ for } \xi = 0 \text{ or } 1 \end{aligned} \qquad (8)$$

then we can construct two interpolants based on interpolating **r** in the ξ and η directions separately:

$$\begin{aligned} \mathbf{r}(\xi,\eta) &= \Sigma_{i=1,2}\ \mathbf{r}(\xi_i,\eta)\ \varphi_i(\xi)\\ \mathbf{r}(\xi,\eta) &= \Sigma_{j=1,2}\ \mathbf{r}(\xi,\eta_j)\ \psi_j(\eta) \end{aligned} \qquad (9)$$

These two functions exactly interpolate the values of **r** on the boundaries $\xi = 0$ or 1 and $\eta = 0$ or 1 respectively. We can add the two interpolants together, but the resulting function does not interpolate **r** on the boundaries. However, we can interpolate the discrepancies on the boundaries, and subtract off this error from the total interpolant. This results in the transfinite interpolant:

$$\begin{aligned} \mathbf{r}(\xi,\eta) = \Sigma_{i=1,2}\ \mathbf{r}(\xi_i,\eta)\ \varphi_i(\xi) + \Sigma_{j=1,2}\ \mathbf{r}(\xi,\eta_j)\ \psi_j(\eta)\\ \Sigma_{i=1,2}\ \Sigma_{j=1,2}\ \varphi_i(\xi)\ \psi_j(\eta)\ \mathbf{r}(\xi_i,\eta_j) \end{aligned} \qquad (10)$$

This function exactly interpolates the values of **r** on all four boundaries of the rectangular computational space. By uniformly dividing the computational space, and applying the interpolating function we can generate the grid in the physical space, a very efficient process. The algebraic method can be easily generalized to three-dimensional grids, and to generate grids posessing any desired order of smoothness by an appropriate choice of the basis functions. However, the user of this method does not have complete control over such grid quantities as orthogonality and clustering.

The Maximum Principle

Solutions of the Dirichlet problem

$$\begin{aligned} &\Delta u = 0 \ \text{ in } \Omega\\ &u = \underline{u} \ \text{ on } \partial\Omega\\ &(\,\Delta = \partial^2/\partial x^2 + \partial^2/\partial y^2\,) \end{aligned} \qquad (11)$$

exhibit a very special and useful property known as the maximum principle. This principle states that the solution achieves its maximum and minimum values on the boundary of the domain. This fact can be used to generate grids that are guaranteed not to be overlapped and in addition are very smooth. The smoothness of the grid results from the inherent smoothness of solutions to the Dirichlet problem. Methods that are based on the maximum principle are popular and include the elliptic grid generation methods of Winslow (ref) and Thompson *et. al.*[1] and the variational method of Brackbill and Saltzman[4].

Elliptic Grid Generation

The elliptic grid generation method[1] is based on the numerical solution of the following equations

$$\begin{aligned} &\Delta\xi = 0\\ &\Delta\eta = 0 \end{aligned} \qquad (12)$$

Solutions to these equations will satisfy the maximum principle. Thus, the maximum and minimum values of ξ and η will be reached on the boundary of the physical space and thus there is no possibility that grid overlap can occur in the domain. To generate the grid, these equations are transformed using relations (5) above to relate the derivatives in the computational space to derivatives in physical space. The result is the following non-linear elliptic system of partial differential equations to be solved for the physical grid

$$\begin{aligned} &\alpha\, x_{\xi\xi} - 2\,\beta\, x_{\xi\eta} + \gamma\, x_{\eta\eta} = 0\\ &\alpha\, y_{\xi\xi} - 2\,\beta\, y_{\xi\eta} + \gamma\, y_{\eta\eta} = 0 \end{aligned} \qquad (13)$$

where

$$\alpha = x_\eta^2 + y_\eta^2$$
$$\beta = x_\xi x_\eta + y_\xi y_\eta$$
$$\gamma = x_\xi^2 + y_\xi^2$$

These equations are discretized using finite differences and then solved using iterative techniques. The method can be modified so that interior grid point clustering can be controlled by the addition of non-homogeneous terms to the right-hand-side of eq. 12. This results in a coupled system of non-linear Poisson equations to be solved. An Example of the type of grid generated by this method is shown in fig. 4.

The Variational Grid Generation Method

The variational grid generation method[4] is based on some heuristic principles concerning what a good quality grid consists of. These principles are 1) the grid should be as smooth as possible, 2) orthogonal as possible, and 3) should cluster in specified regions of the physical domain. These principles can be stated mathematically by minimizing appropriate measures of the smoothness, orthogonality and clustering functions. The smoothness of the grid is measured by the following functional

$$I_s = \int_\Omega (\nabla\xi)^2 + (\nabla\eta)^2 \text{ dx dy} \qquad (14)$$

The orthogonality of the grid is measured by the functional

$$I_o = \int_\Omega \nabla\xi \cdot \nabla\eta \text{ dx dy} \qquad (15)$$

Finally, the grid clustering is measured by the functional

$$I_w = \int_\Omega w(\xi,\eta)\ J(\xi,\eta) \text{ dx dy} \qquad (16)$$

Now we can form one functional that is a weighted average of all three quantities we wish to minimize:

$$I(\xi,\eta) = \lambda_s I_s + \lambda_o I_o + \lambda_w I_w \qquad (17)$$

where λ_s, λ_o, and λ_w are scalars that give different weighting to each functional. Minimizing $I(\xi,\eta)$ will produce an optimal grid with respect to the chosen grid quality measures. The $\boldsymbol{\xi}$ coordinates are transformed to the physical **x** coordinates using eq. 5 and then the Euler-Lagrange equations are formed. The resulting system is similar to the system of eq. 13 and is solved using finite difference discretization and iteration. One can see that where the weighting function is large, the grid cell size will be small, achieving the required clustering. Therefore, the method is particularly suited to generation of adaptive grids. The usual procedure is to choose the weight function as a measure of the rate of change of the solution to the equation we are trying to solve on the grid. One example of this adaption is shown in fig. 5. Note how the grid adapts to the steep gradients in the vicinity of the shock waves in the solution.

It is possible to directly discretize and minimize $I(\xi,\eta)$ before applying the Euler-Lagrange equations. This method was used in refs. 5 and 6 to generate structured and quasi-structured grids. Functionals analogous to I_s, I_o and I_w were developed to measure the grid quality. The grid points are solved for by minimizing the grid quality measures using a conjugate gradient optimization algorithm. An example grid generated by this procedure is shown in fig. 6. This method is also applicable to unstructured grids and is further discussed below.

Quasi-Structured Grids

Quasi-structured (zonal, composite or patched) grids have been used by many[1,2] to take advantage of the simplicity of finite difference methods while allowing the discretization of more general domains. The physical domain is first subdivided into a small number of regions, each of which can be separately associated with a rectangular computational domain. The grids are generated in each sub-domain and then patched together along common boundaries. These interfaces between the domains can be fixed or allowed to float along with the nodal points in the grid generation procedure. A method for generating composite grids based on the method of ref. 5 was developed in ref. 6. An example of using this procedure is shown in fig. 7. This is a composite grid consisting of two structured grids, one an O-type grid about a gas turbine engine blade and the second a C-type grid surrounding the O-type grid. The quasi-structured grid approach is a powerful tool; however, it is not as general as the unstructured grid methods.

Unstructured Grids

Unstructured grids afford the most geometric generality. In fact, one can discretize an arbitrary domain using an appropriate unstructured grid. For this reason, these types of grids have been receiving much attention recently, as solutions of field equations are required on more and more complex geometries. Unstructured grids can be generated by hand (or with the use of a digitizing tablet), but this method is tedious, time-consuming and has a large human error factor. Thus, we are led to automatic methods for unstructured grid generation. The most general automatic unstructured grid generation methods are those based on the n-simplex (triangle in 2-d and tetrahedra in 3-d). The procedure is to first specify the positions of the N nodal points which in this case are just the vertices of the triangles or tetrahedra. (The positions of the nodal points can, for example, be generated pseudo-randomly). Next, a scheme is used to connect the N points to form the triangular elements forming the grid. The points x_j, j = 1,2,...N, can be joined using purely heuristic or semi-optimal rules[7]. The optimality of the grid can be measured by how similar each element is to an equilateral triangle (similarly in 3-d). It is striking that there exists a very simple rule for joining the vertices that ensures that the grid is optimal--the Delaunay criterion[8,9]. First we define the Voronoi polygon surrounding a point $\mathbf{x}_i$ as the set of points closer to $\mathbf{x}_i$ than any other point

$$V(\mathbf{x}_i) = \{ \mathbf{x} \mid d(\mathbf{x},\mathbf{x}_i) \leq d(\mathbf{x},\mathbf{x}_j) \quad \text{for all } j=1,2,\ldots,N\}$$

where $d(\cdot,\cdot)$ is the Euclidean metric ($d(\mathbf{x},\mathbf{y})$ = distance between $\mathbf{x}$ and $\mathbf{y}$). Two points $\mathbf{x}_i$ and $\mathbf{x}_j$ are termed Voronoi neighbors if their respecive Voronoi polygons share a common edge. Thus, the Delaunay criterion states that node $\mathbf{x}_i$ is to be connected to node $\mathbf{x}_j$ if they are Voronoi neighbors. This procedure produces grids such as the one shown in fig. 8 for two airfoils in a wind tunnel. Note that the procedure has to be modified for non-convex domains such as this. In effect, the connections are checked to see if they cross any of the required boundary edges, and if so, one of the nodes is deleted from the search for Voronoi neighbors. This triangulation procedure can be made to take on the order of NlogN operations, and thus is very efficient.

Grid Improvement and Adaption Methods

In general, a given computational grid is not the optimal one for solving a problem. For example, it is usually required to have the grid clustered in some regions of the domain in which the solution to the problem is varying rapidly. Since the solution is not known *a priori*, we are led to methods that dynamically adapt the grid to the solution. This adaption can take either of two forms: 1) the main grid is held fixed while individual sub-domains are refined into smaller sub-domains (this is called adaptive refinement) or 2) the basic logical structure of the grid is held fixed and the grid points are allowed to move such that they become clustered in required regions (this is just called an adaptive grid). The effect of either of these types of adaptions is to make the grid spacing smaller in the required regions, and therefore increasing the accuracy of the numerical solution. An example of adaptive grid refinement[10] is shown in fig. 9. Note how the grid is refined in certain regions requiring resolution. An example of an adaptive grid is shown in fig 5. The grid points have moved and clustered in the required regions. The method of ref. 5 can be modified so that it is applicable to unstructured grids. Thus, unstructured adaptive grids could be generated analogously to structured adaptive grids. In conclusion, adaptive methods are one of the most promising techniques for improving the accuracy of numerical solutions to partial differential equations.

Conclusions

Numerical methods for the generation of computational grids have been presented. These methods have been shown to be very powerful tools for use in the simulation of complex physical processes governed by partial differential equations.

References

[1] Thompson, J. F., Z. U. A. Warsi, and C. W. Mastin. Numerical Grid Generation. Elsevier Science Publishing Co. New York. 1985.

[2] Ghia, U. and K. Ghia, eds. Advances in Grid Generation. American Society of Mechanical Engineers. 1983.

[3] Dulikravich, G. S., "CAS22-FORTRAN Program for Fast Design and Analysis of Shock-Free Airfoil Cascades Using Fictitious-Gas Concept," NASA CP-3507, January, 1982.

[4] Brackbill, J. U. and J. S. Saltzman, "Adaptive Zoning for Singular Problems in Two Dimensions," Journal of Computational Physics, **46**, 1982, pp. 342-368.

[5] Kennon, S. R. and G. S. Dulikravich, "A Posteriori Optimization of Computational Grids," AIAA paper no. 85-0483, Reno, Nevada, January, 1985.

[6] Kennon, S. R. and G. S. Dulikravich, "Composite Computational Grid Generation Using Optimization," First International Conference on Numerical Grid Generation in Computational Fluid Dynamics, Landshut, W. Germany, July 14-17, 1986.

[7] Nguyen, V. Ph., "Review of Techniques for Efficient Network Generation for Finite Element Analysis," VDI-Forschungsheft No. 2, 1982.

[8] Watson, D. F., "Computing the n-dimensional Delaunay Tessellation with Application to Voronoi Polytopes," The Computer Journal, **24**, no. 2, 1981, pp 167-172.

[9] Bowyer, A., "Computing Dirichlet Tessellations," The Computer Journal, **24**, no. 2, 1981, pp. 162-166.

[10] Löhner, R., K. Morgan and O. C. Zienkiewicz, "An Adaptive Finite Element Procedure for Compressible High Speed Flows," CMAME, **51**, 1985, pp. 441-465.

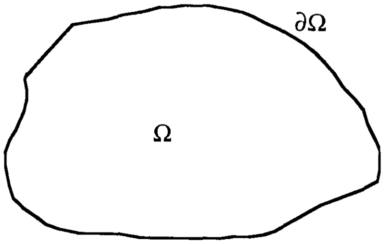


Fig. 1a A two-dimensional domain Ω

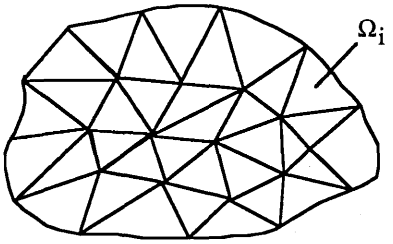


Fig. 1b The domain Ω showing its discretization into sub-domains Ω_i

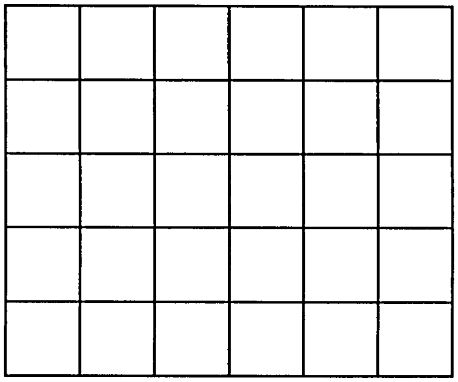


Fig. 2a A typical base domain showing discretization using a repetition of squares

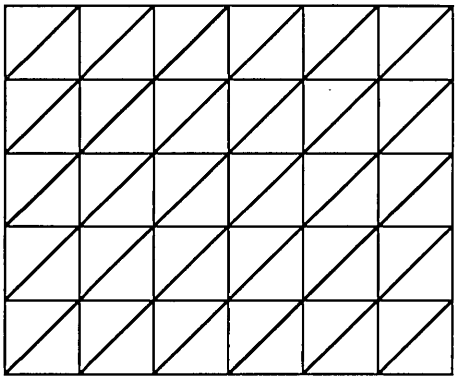


Fig. 2b A base domain consisting of a repetition of regular triangular sub-domains

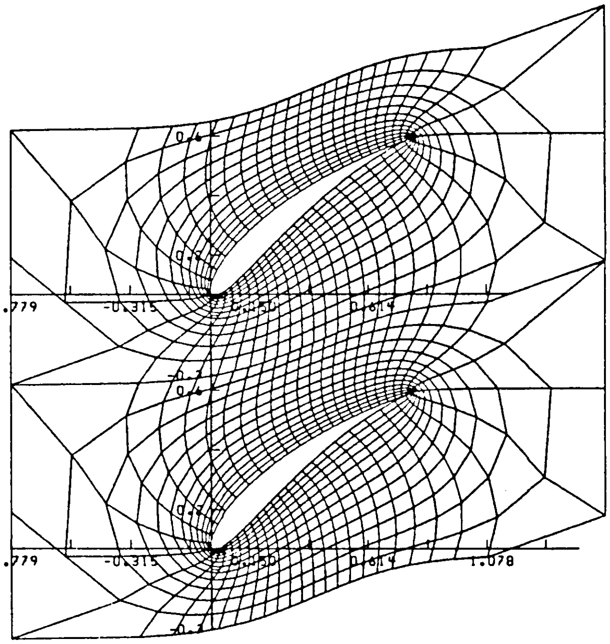


Fig. 3 Grid about a two-dimensional cascade generated using conformal mapping[3].

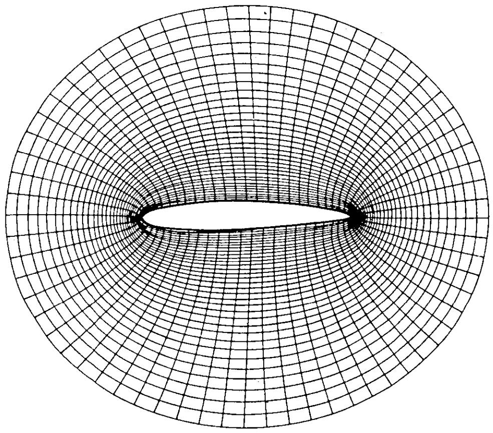


Fig. 4 Airfoil grid generated using the elliptic method[1].

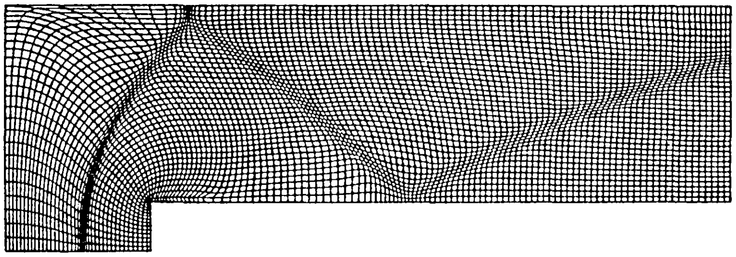
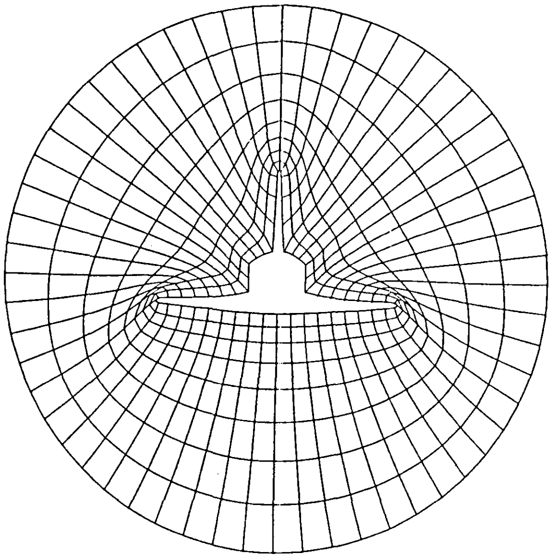


Fig. 5 Adaptive grid for a two-dimensional compressible flow problem generated using the variational method[4].



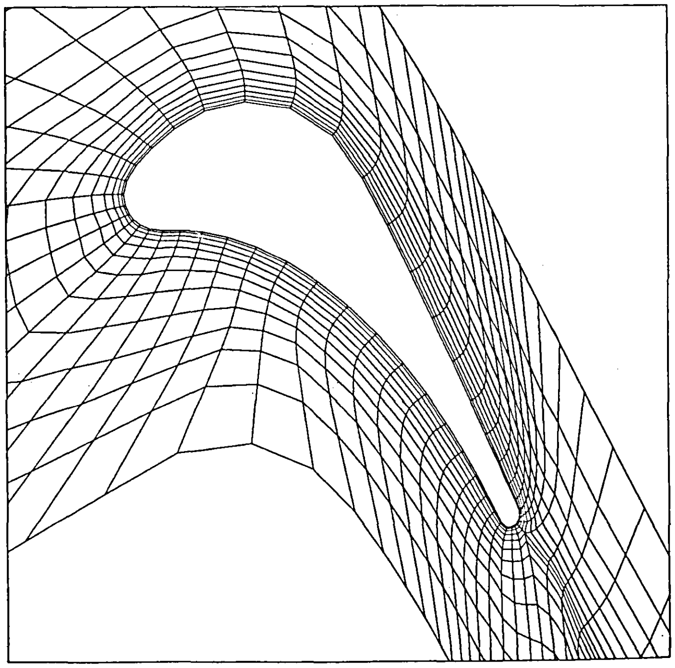


Fig. 6 Optimized grid for a space shuttle cross-section[5].

Fig. 7 Composite grid for a turbine blade[6].

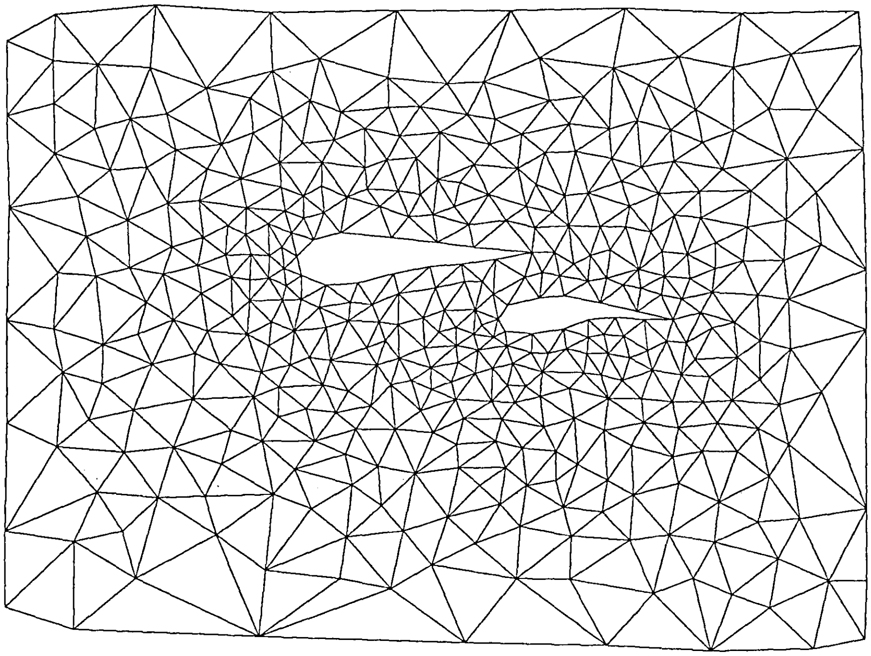


Fig. 8 Delaunay triangulation for two airfoils in a wind tunnel.

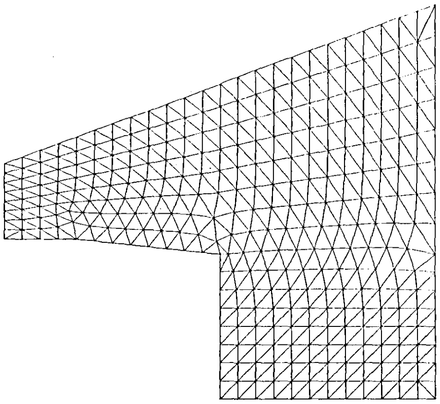


Fig. 9a Initial un-refined grid

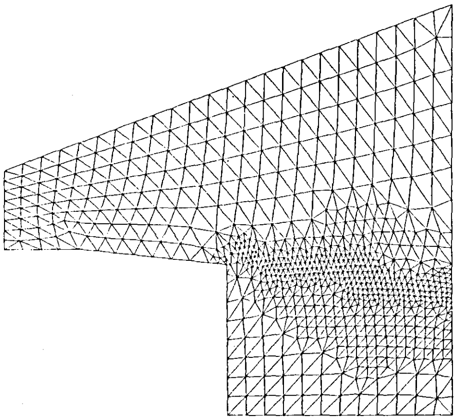


Fig. 9b Adaptively refined grid

Intelligent Backtracking Using Symmetry

Cynthia A. Brown and Larry Finkelstein

College of Computer Science
Northeastern University
360 Huntington Ave.
Boston, Mass. 02115

Paul Walton Purdom, Jr.

Computer Science Dept.
Indiana University
101 Lindley Hall
Bloomington, In. 47405

Abstract. Symmetries occur naturally in many types of problems. (An example is the eight queens problem under rotation or flip of the chessboard). The performance of search algorithms in solving such problems can be improved by recognizing as early as possible that two partial solutions are equivalent under symmetry. However, most uses of symmetry in searching are problem specific and may not take full advantage of the potential improvements. In this paper we describe a general method for incorporating symmetries in search methods. The input to our procedure is a problem together with an abstract representation of its underlying symmetries (known as a *group*). The method automatically calculates the symmetry tests to be performed at each stage in searching, making use of efficient algorithms for computing with groups. In this way the full benefit of the symmetries is obtained without the need for *ad hoc* techniques.

1. Introduction

Searching is an important method for solving difficult problems. A great deal of effort has been directed into making search methods more "intelligent", in order to improve their efficiency. There are two common approaches. One method is to guide the search by using known properties of the problem under investigation. A second approach is to develop general methods which do not depend on the nature of the problem. A common characteristic of such general methods is the ability to reject large numbers of possible solutions simultaneously, rather than having to consider each one individually. The most popular intelligent search method of this type is known as *backtracking.* The basic idea of backtracking is to build a solution to a problem one piece at a time. If it is possible to identify a partial solution as one that cannot be extended to a full solution, then we eliminate from the search space all possible extensions of this partial solution.

To do well with backtracking, it is necessary to develop tests that can eliminate partial solutions as early as possible. Most efforts to improve backtracking programs for individual problems concentrate on developing better criteria for eliminating partial solutions. While important, such improvements tend to be problem-specific, so there is little or no carry-over to other problems. There are also general methods, such as search rearrangement [4,18], that allow the criteria to be applied earlier in the search process. Search rearrangement can lead to a very significant improvement in the behavior of a backtrack program. For some problems there is another possibility: the existence of symmetry in the potential solutions to the problem may allow certain sections of the search space to be omitted altogether.

Symmetries occur naturally in many types of backtrack problems. For example, consider the problem of embedding a graph $\mathcal{G}_1$ in another graph $\mathcal{G}_2$. If the automorphism group G of $\mathcal{G}_2$ (the group of one-to-one edge-preserving maps of $\mathcal{G}_2$ onto itself) is known, then a partial embedding can be tested by seeing if it is equivalent under some automorphism to a partial embedding that has already been examined. If such an equivalence is found, then the current partial solution can be discarded.

In many problems, there are variables that play interchangable roles. This is another manifestation of symmetry. As a simple example, consider the missionaries and cannibals problem. Three missionaries and three cannibals must cross a river using a two-man boat. Either one or two people can take the boat across the river. The problem is to devise a plan for crossing in which the cannibals never outnumber the missionaries on either side of the river. This problem is typically given to beginning classes in artificial intelligence to be solved using backtracking. The missionaries are clearly interchangable amongst themselves, as are the cannibals. Thus, it is only necessary to consider solutions where the missionaries cross the river in some particular order, and the same for the cannibals. Any other solution can be obtained from such a solution by permuting the names of the missionaries, and of the cannibals, amongst themselves.

In general, it is not necessary to try all combinations of values for interchangable variables. For example, if there are n interchangable Boolean variables, then we need only test $n+1$ sets of values (no variables equal to zero, one variable equal to zero, ..., n variables equal to zero) instead of the 2^n different combinations that would otherwise be needed.

CH2345-7/86/0000/0576$01.00 © 1986 IEEE

Symmetry can also arise from the underlying structure of the problem. As an example, consider the eight queens problem: place eight queens on a chessboard so that none of them is attacking any of the others. Any solution to the problem can be converted to another (possibly different) solution by rotating the board through 90, 180, or 270 degrees, or by flipping it to a mirror image position. How can this fact be used to reduce the search space? Let us refer to the queens as $Q_1, \ldots, Q_8$. Since no two queens can be in the same row, we assume that Q_i goes in row i. Then we need only consider placing Q_1 in columns 1 to 4, since any solution with Q_1 in column $4+i$ can be obtained from a symmetrical one with Q_1 in column $5-i$ by flipping to the mirror image. This observation eliminates half of the search space.

It is less obvious how to apply the rotational symmetries. A little thought shows, for example, that if we can rule out any solution with Q_1 in column 1, we do not need to try putting a queen in any corner of the board. But what we really need is a systematic way of applying knowledge about symmetries to reducing the search space.

The systematic study of symmetries is the province of group theory. The obvious symmetries of the eight queens problem form a well-known group: the dihedral group of order eight. In recent years there has been a great deal of activity in the field of computational group theory, inspired in part by the classification of all finite simple groups [3]. Several systems for performing basic group theory operations are currently available [8].

It is very common for a problem to display one or more of the types of symmetry described above. By taking advantage of such symmetries to reduce the search space, knowledge about the problem that is not explicit in the predicates can be exploited. We are developing a systematic procedure for applying group-theoretic methods to exploit the existence of symmetries in backtracking problems. This procedure includes methods for representing the symmetries and for automatically calculating the tests to be applied at each stage in the backtracking process in order to take maximum advantage of them. We are also working on developing and improving the group-theoretic algorithms required for these calculations.

Some care is needed if the symmetry testing is to speed up a search program, not just reduce the number of nodes. A naive approach would use time proportional to the size of the symmetry group at each node. Since the maximum reduction in the number of nodes is also proportional to the size of the symmetry group, little or no savings would result. There are two strategies for improving on the naive approach. The first is to use efficient algorithms for the symmetry testing. We make use of algorithms from computational group theory to carry out the symmetry tests [3,6,15,16,19]. The general test we apply at each node in the search space is essentially equivalent to computing a certain color automorphism group. We have two fairly distinct methodologies for computing this test. The first one is based on a general algorithmic method for computing with permutation groups developed by Sims [19] and refined by Leon [15] and Butler [6]. This method usually does the symmetry testing rapidly, but it can take exponential time. The second only applies where the group is a p-group [11], but it is fast because it uses Luks' color automorphism algorithm [16].

A second approach to reducing the time spent on symmetry testing is to discontinue the testing on portions of the search space where it can have no effect. We have a criterion which establishes certain conditions under which symmetry testing cannot yield any further savings. Since symmetry testing usually has most of its effect near the root of the search tree, this often permits us to save a significant amount of time.

The methods for intelligent searching described in this paper have characteristics similar to other such methods. In practice they may speed up searching greatly, but there is no guarantee of a speed-up on any particular problem, and there is no indication that the method improves the worst-case time for searching.

2. Backtracking and Symmetry

Backtracking is a widely used method for reducing the size of a search space. To apply backtracking to a problem it must be expressed in a suitable form. Many problems may be expressed as predicates on a set of variables. A solution to such a problem is an assignment of values to the variables that makes the predicate true. To do backtracking, we must have a set of *intermediate predicates* available. An intermediate predicate is a predicate over a subset of the variables which is false for a given assignment of values to its variables (which we call a *partial solution*) only if that assignment cannot be extended to a solution to the full problem. In other words, an intermediate predicate is able to identify some partial solutions as dead ends, and therefore cause the program to back up. In the eight queens problem, for example, the intermediate predicate is the check that the queens placed on the board so far do not attack each other.

We call the problem predicate P, and the problem variables $x_1, \ldots, x_n$. Let P_i be the intermediate predicate for $x_1, \ldots, x_i$. We simplify the discussion by assuming that each variable has values in the range $0, \ldots, v$. In addition, we use a special value, u, to represent an unset variable.

Each partial solution X may be thought of as a sequence $X = x_1 \ldots x_n$ of n values. A partial solution is said to be *complete* if all of its variables are set, and *incomplete* otherwise. We will assume that the variables of X are set from left to right. If X is an incomplete partial solution and x_k is the first unset variable of X, then all variables x_j of X with $j \geq k$ would normally be unset. However, as we shall see, our use of symmetry will often allow us to set certain variables $x_j, j \geq k$, to v. Accordingly, we assume that all variables $x_j, j \geq k$, are either unset or set to v. We need a partial order $\prec$ on the set of all partial solutions with the property that if $X \prec Y$ and if X and Y are incomplete, then all possible extensions of X to a complete solution will be examined before any attempt to extend Y. The partial order we use is defined by $X \prec Y$ if there is a prefix $y_1 \dots y_k$ of Y consisting of set variables which satisfy the following properties:

1. For $j < k$, either $y_i = x_i$, or $y_i = v$ and $x_i = u$.
2. x_k is set and $y_k > x_k$.

In the case where X and Y are complete, $\prec$ is the usual lexicographic ordering.

Let S_n be the symmetric group of degree n. The group S_n may be described as the set of all permutations of the set $\Omega = \{1, 2, \dots, n\}$, with the group action being ordinary composition of functions. We denote an application of $g \in S_n$ to $i \in \Omega$ by the exponential notation i^g. The group S_n acts naturally on the set of all sequences of length n by permuting the sequence values according to the permutation of their indices. To describe the action formally, let g be an element of S_n and $X = x_1 x_2 \dots x_n$ be a sequence of values. Then $Y = X^g$ is the sequence defined by setting $y_j = x_i$, where $j = i^g$. Equivalently, one may write Y as the sequence $x_{1_{g^{-1}}} x_{2_{g^{-1}}} \dots x_{n_{g^{-1}}}$. It is easy to check that according to this definition $X^{gh} = (X^g)^h$ for all sequences X and all elements $g, h \in G$. In many instances it is more convenient to consider $X^{g^{-1}}$ instead of X^g because $X^{g^{-1}} = x_{1^g} x_{2^g} \dots x_{n^g}$.

We are particularly interested in the set G of elements of S_n which leave invariant the set of solution sequences to P. (That is, the elements of G carry solutions into solutions.) It is easy to show that the product of two elements of G preserves all solutions, and so G is a subgroup of S_n. As a consequence, G also preserves the set of sequences which are not solutions to P; either property defines G. For example, the dihedral group of order eight may be used for G in the eight queens problem.

We now describe our method for exploiting the existence of symmetries. Let X be a partial solution for which the values $x_1, \dots, x_{k-1}$, $k > 1$, have already been set, and assume that x_k is the variable under consideration. (The current value of x_k must either be u or some number less than v.) If $x_k = u$, then we initialize the value of x_k to 0. Otherwise, x_k is incremented by 1. Assume that $P_k(x_1, \dots, x_n)$ is true so that, in the normal course of events, the backtracking procedure will continue by attempting to extend the partial solution X.

At this point we call the function *SymTest*, which contains the heart of our application of symmetry to backtracking. SymTest has an argument list of the form $(\mathcal{G}, X, k, \mathcal{H}, g)$, where $\mathcal{G}$ is a list of generators for G, X is the current partial solution, and k is the index of the last variable of concern. SymTest returns a Boolean value which is *true* if there is an element $g \in G$ (such that $X^{g^{-1}} \prec X$) and *false* otherwise. If such an element g exists, it is returned in the last parameter of Symtest; otherwise, variable $\mathcal{H}$ will contain generators for a subgroup H of G which will be defined shortly. In addition, SymTest has the side effect of possibly setting certain variables $x_j, j > k$, to the value v and of restricting the possible values of other variables.

If SymTest returns *false*, then $\mathcal{H}$ will contain a list of generators for the subgroup H of G which preserves a certain coloring $\mathcal{C}$ of the elements Ω induced by X. The colors in $\mathcal{C}$ are drawn from the set $\{0, \dots, v\} \cup \{u\}$. Elements $i, j \in \Omega$ are said to have the same color, denoted $i \sim j$, if and only if either $x_i = x_j$ or one has value v and the other u. (For reasons which will become clear later, we consider u and v to be the same color.) We then define $H = \mathcal{C}_\Omega(G)$ to be the set of all elements $g \in G$ which preserve the coloring $\mathcal{C}$; i.e. $g \in H$ iff $i^g \sim i$ for all $i \in \Omega$. It is straightforward to show that H is closed under composition, and hence forms a subgroup of G.

In our implementation of SymTest, we essentially attempt to compute the color automorphism group H with modifications made to suit our backtracking environment. At this time, it is not known if there is a polynomial time algorithm to solve the color automorphism problem. However, there do exist special cases (which depend on the structure of G) for which polynomial time algorithms are known [16]. We have two different methodologies for computing H. The first method is based on a general approach for computing with permutation groups develped by Sims [19] and the second method utilizes an especially efficient algorithm [11] based on methods of Luks [16] for computing H in the case where G is a $p-$group, i.e. G has order a power of a prime p. It seems likely that the computations needed for our application will be relatively inexpensive in either case. We sketch the Sims' type method in Section 5; see [5] for a complete description of both methods.

Although our two methodologies for implementing SymTest are rather different, we present a somewhat simplified view of what both are attempting to accomplish. The basic idea is to incrementally compute permutations g that preserve $\mathcal{C}$. As g is extended to the next variable, we check whether it gives us a reason to back up. Thus, SymTest is designed to successively examine the values of x_{j^g}, $1 \leq j \leq k$, and to take appropriate action according to which of the the following cases occurs. Initially, $\mathcal{H}$ is the empty list, and at any given time $H = \langle \mathcal{H} \rangle$ is the group generated by H (the empty list generates the identity subgroup). We start by choosing an arbitrary element g of

G and setting j to 1. The more realistic versions of the algorithm differ from this naive version by not actually examining each g separately.

1. j = k+1. If we eventually arive at this case, then we know that $x_{i^g} = x_i$ for $1 \leq i \leq k$. Since x_j, $j > k$, is either set to v or is unset, it follows that g preserves $\mathcal{C}$. We then add g to $\mathcal{H}$ and choose an element not yet considered in $G - H$. If no such element exists, then we return false and $\mathcal{H}$ will then contain a set of generators for $\mathcal{C}_\Omega(G)$.

2. $x_j = x_{j^g}$. In this case, we increment j and continue to examine g.

3. $x_{j^g} < x_j$. In this case, $X^{g^{-1}} \prec X$ and SymTest returns true with its last parameter set to g . At this point in the backtracking procedure we need not consider X further, since X is equivalent under symmetry to a smaller partial solution.

4. $x_j < x_{j^g}$, or $x_j < v$ and $x_{j^g} = u$. Two subcases arise in this case. If x_{j^g} is set, then it is unnecessary to consider g further, since $X^{g^{-1}} \succ X$. If $x_j < v$ and $x_{j^g} = u$, then if x_{j^g} were later set to a value less than x_j while we were in the current branch, then we would have $X^{g^{-1}} \prec X$. Thus we should keep track of this information and not set x_{j^g} to any value less than x_j. This information represents a constraint on the future values of x_{j^g}, so this part of the method is reminiscent of constraint satisfaction methods. In either case, we choose a new element to consider from $G - H$.

5. $x_j = v$ and $x_{j^g} = u$. In this case, $x_j = v$ and $x_{j^g} = u$. If, at some later time, we were to set $x_{j^g} = r$ with $r < v$, then we would have $X^{g^{-1}} \prec X$. Therefore, we may avoid additional work by setting $x_{j^g} = v$ now and continuing to extend the definition of g.

In cases 1 and 4, if we have exhausted all elements of $G - H$, then we return false with $\mathcal{H}$ a set of generators for $\mathcal{C}_\Omega(G)$. It should be noted that each time we add a generator to $\mathcal{H}$ we at least double the size of H.

Much of the useful work done by SymTest occurs during the attempt to extend an element $g \in G$ to all indices $\{1, \ldots, k\}$. Each time a variable is set or its domain of definition is restricted, as in case 4, we remove a large portion of the search space.

SymTest returns either an element $g \in G$ with the property that $X^{g^{-1}} \prec X$, if one exists, or else return a set of $O(n)$ generators for the subgroup H. In the latter case, it may still be possible to set some additional variables. To see how this is done, suppose $g \in H$. This leads to the partial solution given by the sequence $X^{g^{-1}} = x_{1^g} \ldots x_{n^g}$ (it is more convenient to consider the image of X under g^{-1} than g). By definition of H, for $i \leq k$, we must have $x_i = x_{i^g}$ whenever $x_i < v$; however, we may have $x_i = v$ and $x_{i^g} = u$. As noted before, if at some later time, we were to set $x_{i^g} = r$ with $r < v$, then the symmetry g^{-1} would have the property that $X^{g^{-1}} \prec X$. Therefore, we may set all variables x_{i^g} currently set to u to have value v. Applying g to $X^{g^{-1}}$ then results in the original partial solution X with possibly one or more unset variables now set to v. Each such setting represents a decrease by one in the number of unset variables of X, and thus effectively decreases by one variable the size of the remaining backtrack problem on the branch being investigated.

A more efficient method for accomplishing the above procedure is to simply compute the orbit j^H for each position $j \leq k$ that contains a v, and to make sure that the variables corresponding to the positions in each of these orbits is set to v. In other words, for each such j compute

$$j^H = \{\, r : r = j^g,\ g \in H \}$$

and set $x_r = v$ for each r in the orbit. The advantage in using this method is that the cost of computing an orbit of H is proportional to the product of n and the number of generators of H. In general, H can be compactly described by a set of at most $n-1$ *strong generators* [19] (the notion of a strong generator will be discussed in Section 4). Thus the cost of applying the symmetry information is linear in the number of generators times n, whereas a naive method would take time proportional to the size of H times n.

We now formulate a backtracking algorithm to incorporate these ideas. There are many possible data stuctures which may be used for representing groups. For our purposes, we will represent a group by a list of generating elements.

Backtracking with Symmetry Test. Input: Variables $\{x_1, \ldots, x_n\}$ and intermediate predicates P_i. In addition, a set $\mathcal{G}$ of generators for a group G of symmetries for the problem. Output: all sets of values for the variables such that P is true. Variables used: a stack S_X to store the partial solution X. The primitive stack operations are *Push, Pop* and *TopOf.* The algorithm calls the procedure *SymTest* with argument list $(\mathcal{G}, X, k, \mathcal{H}, g)$. SymTest returns a Boolean variable which is true if an element $g \in G$ is found such that $X^{g^{-1}} \prec X$. In this case g is returned in the last parameter. Otherwise, SymTest returns false and a list of generators for $\mathcal{C}_\Omega(G)$ is returned in $\mathcal{H}$.

```
1. [Initialize]
   $k \leftarrow 0$; $S_X \leftarrow \Lambda$;
   for $i \leftarrow 1$ to $n$ do $x_i \leftarrow u$ endfor
2. [Solution?]
   if $k \neq n$ then goto Step 3
   else print($X$); goto Step 4 endif
3. [Search deeper]
   $k \leftarrow k + 1$;
   if $x_k = v$ then goto Step 6 {$x_k$ previously set }
```

```
4. [Try next value]
   if $x_k = v$ then goto Step 7
   elseif $x_k = u$ then
      $S_X \leftarrow Push(S_X, X)$; $x_k \leftarrow 0$
   elseif $x_k < v - 1$ then
      $X \leftarrow TopOf(S_X)$; $x_k \leftarrow x_k + 1$
   else $\{x_k = v - 1\}$;
      $X \leftarrow Pop(S_X)$; $x_k \leftarrow v$ endif;
   if $P_k(X)$ is false then goto Step 4 endif
5. [Apply Symmetry Check]
   if $SymTest(\mathcal{G}, X, H, k, g)$ then goto Step 4;
   $\mathcal{O} \leftarrow \{i \in \Omega : i = j^g, x_j = v, g \in H, j \leq i\}$;
   foreach $i \in \mathcal{O}$ do $x_i \leftarrow v$
6. [Check predicate]
   if $P_k(X)$ then goto Step 2 endif
7. [Backtrack]
   $k \leftarrow k - 1$;
   if $k = 0$ then stop
   elseif $x_k = 0$ then goto Step 4
   else goto Step 7 endif.
```

In Step 4, x_k is initally set to u. Once we set $x_k = 0$ and call SymTest, we may well alter some of the variables $x_j, j > k$. Therefore, it is necessary to stack the previous values of X. If we backtrack to variable x_k, then we have to restore the previous environment. In the case where we enter Step 4 with x_k set to $v - 1$, it is unnecessary to save the environment stored on the stack, since it will never be used again. Consequently, we may pop the stack and store the result in variable X, before setting $x_k = v$.

3. The Four Queens

Let us consider the effect of applying symmetry considerations to the problem of placing queens on a chessboard, as described in the introduction. In order to make the size of the problem tractable for illustrations, we consider placing four queens on a four by four board. The four queens problem is normally formulated in terms of four variables, $Q_1, \ldots, Q_4$, where the value of Q_i is the number of the column that the queen in row i has been placed in. This formulation takes advantage of the fact that no two queens can be in the same row to reduce the number of variables from sixteen binary variables (representing the sixteen squares on the board) to four variables, each of which can take on four possible values.

The four queens problem has two (symmetrical) solutions; the ordinary backtrack tree, which has 61 nodes, is shown in Fig. 1. (The black squares are those occupied by queens.) Most programmers notice and take advantage of the obvious column symmetry, which means that Q_1 need be tested only for values 1 and 2; any solution with Q_1 equal to 3 or 4 can be obtained by rotating the board on its vertical axis. This observation gives a tree with 31 nodes.

Formulating the problem in terms of row variables, the only symmetry that we can express as a subgroup of the permutation group of the variables is the row symmetry which interchanges rows 1 and 4 and rows 2 and 3: in cycle notation, (14)(23). In order to make any use of the symmetry, we must set the two variables in one orbit before the two variables in the other, giving search order Q_1, Q_4, Q_2, Q_3 or Q_2, Q_3, Q_1, Q_4. In this problem, variables in adjacent rows are more likely to eliminate each other than more widely-separated variables, so the second order is preferable. The search tree for the second search order is shown in Figure 2. It has 35 nodes altogether. Eliminating the right side of the tree gives a tree with 30 nodes, one less than before. (The tree for the first search order has 55 nodes; 42 remain when the right subtree is removed.)

We would like to retain the convenience and efficiency of the row-variable formulation of the problem, while taking advantage of the rotational symmetries as well as the row and column symmetries. This can be done by finding a mapping of the row variables onto the variables of the underlying problem (*square variables*) which preserves the search order in the row-variable problem. A partial solution in the row-variable problem is also a partial solution in the square-variable problem. Any partial solution in the row-variable problem which survives the predicate has only one queen per column; hence, it maps onto another partial solution to the row-variable problem under rotational and column symmetry, as well as under row symmetry.

We number the squares of the board with their row and column indices; this gives a set of variables $R_{i,j}, 1 \leq i, j \leq 4$. A search order consistent with the normal order for the row variables is $R_{1,4}$, $R_{1,3}$, $R_{1,2}$, $R_{1,1}$, $R_{2,4}$, $R_{2,3}$, $R_{2,2}$, $R_{2,1}$, $R_{3,4}$, $R_{3,3}$, $R_{3,2}$,$R_{3,1}$, $R_{4,4}$, $R_{4,3}$, $R_{4,2}$, $R_{4,1}$. Using this search order, the partial solutions for the square-variables problem that are also partial solutions for the row-variables problem occur in the same order as in the row-variables problem. The existence of this ordering justifies the use of the symmetries for the square-variables problem on partial solutions for the row-variables problem. Similar orderings exist for other search orders on the row variables.

This justifies the use of the rotational symmetries to prune the search tree for the row-variables problem. Using the normal search order, we obtain a tree with 35 nodes. The results are again better if we choose the search order Q_2, Q_3, Q_1, Q_4. This gives a tree of 28 nodes; if we count in the same way as for the other cases, eliminating the right subtree altogether, then we have 26 nodes. This tree is shown in Figure 3.

On such a small problem, it is very difficult to obtain much improvement, since the predicates are quite effective by themselves and the search tree is already small. The column symmetries (removing the right half of the tree) are usually recognized and used by experienced programmers. We were able to remove one additional node from the left side of the search tree by using row symmetries alone; using both row and rotational symmetries, we eliminated five additional nodes. These results indicate that the application of symmetries, combined with an intelligent search order, will yield very significant improvements on large backtrack problems where symmetry is found.

4. Group Theory Algorithms

We briefly review some fundamental concepts which underlie both the computer implementation and theoretical analyses of algorithms for permutation groups. Let $Sym(\Omega)$ be the group consisting of all permutations of some arbitrary finite set Ω with n elements. A subgroup G of $Sym(\Omega)$, denoted $G \leq Sym(\Omega)$, will always be identified with a list of generators.

A base Γ for G is a subset of Ω with the property that only the identity of G fixes each element of Γ pointwise. The notion of a base was invented by Sims [19] as a means of testing equality of two elements of G. If $g, h \in G$ and g and h agree on Γ, then $g = h$. This idea is useful because many subgroups of $Sym(\Omega)$ have bases which are substantially smaller than n.

Let $\Gamma = \{b_1, b_2, \ldots, b_m\}$. Define the chain of subgroups

$$\{1\} = G_{m+1} \subseteq G_m \subseteq \ldots \subseteq G_1 = G$$

where G_i is the subgroup of G which fixes all the points of $\{b_1, b_2, \ldots, b_{i-1}\}$. A *strong generating* set for G is a set S of generators for G such that if

$$S_i = \{s \in S : s \in G_i, i \geq 1\}$$

then S_i is a set of generators of G_i.

A base and strong generating set is the basic data structure used to encode information about permutation groups. Perhaps the most fundamental result in computational group theory is that if G is specified by a small set of generators then a base and strong generating set can be computed in polynomial time. The original algorithm was given by Sims [19] and will be referred to as Sims' algorithm. An alternative version together with a proof that the algorithm runs in polynomial time was later given by Furst, Hopcroft and Luks [10]. Sims' algorithm can be used to show that the following problems have polynomial time solutions ([10],[19]):

1. (Order) *Compute the order of G.*
2. (Membership) *Given $x \in Sym(\Omega)$, determine if $x \in G$.*
3. (Pointwise stabilizer) *Generators for the subgroup $G_{a_1,a_2,\ldots,a_r}$ of G which fixes pointwise the subset $\{a_1, a_2, \ldots, a_r\}$ of Ω.*

5. SymTest Using Sims' Method

In describing our implementations of SymTest, we shall assume that $\Omega = \{1, 2, \ldots, n\}$ and denote $Sym(\Omega)$ by S_n. Recall that SymTest has argument list $(\mathcal{G}, X, k, \mathcal{H}, g)$ where $\mathcal{G}$ is a list of generators for G, X is the partial solution, x_k is the variable under consideration and $\mathcal{H}$ and g are the output parameters. SymTest returns *true* if there is an element $g \in G$ such that $X^{g^{-1}} \prec X$; otherwise, SymTest returns *false* and $\mathcal{H}$ contains a set of generators for the color automorphism group $H = \mathcal{C}_\Omega(G)$.

The version of SymTest described in this paper is based on a general method for computing with permutation groups developed by Sims [19] and later refined by Butler [6] and Leon [15]. Our goal is to simply sketch the highlights of this method and refer the reader to our forthcoming paper [5] for full details. Sims' method has a wide range of applications and forms the basis of many of the group theory algorithms implemented in the CAYLEY system [8].

We will assume that the first m elements $\{1, \ldots, m\}$ of Ω form a base Γ for G. A sequence $T = (t_1, \ldots, t_j)$ of distinct points of Ω is called a *partial image*. If $j = m$, then T is called a *complete image*. For any partial image T and subgroup $H \subseteq G$,

$$H(T) = \{g \in H : (1, \ldots, j)^g = T\}$$

and if T is incomplete then

$$X_H(T) = \{y \in \Omega : H(T \cup y) \neq \emptyset\}.$$

Think of $X_H(T)$ as the possible ways of extending a valid partial image (i.e. $X_H(T) \neq \emptyset$) to another valid partial image.

The following result describes $X_G(T)$ precisely (see [6], Proposition 1).

Proposition 1

Let $T = (t_1, \ldots, t_{j-1})$ be a partial image and let $g \in G(T)$. Then

$$X_G(T) = (j^{G_j})^g.$$

Suppose now that P is a certain property defined on G and we are trying to find an element of G, or possibly all elements of G, which have this property. Let

$$P(T) = \{g \in G(T) : g \text{ has property P}\}$$

and let

$$X_P(T) = \{y \in \Omega : P(T \cup y) \neq \emptyset\}.$$

For our purposes, g has property P if g preserves the coloring C. Equivalently, g has property P if $x_i = x_{i^g}$ for all $j, 1 \le j \le k$. In general, it is difficult to compute $X_P(T)$. Instead, we use an easily computed approximation $\bar{X}_P(T)$. Given $T = (t_1, \dots, t_{j-1})$, with $1 \le j \le k$, and $g \in G(T)$, if $x_j < v$, define

$$\bar{X}_P(T) = \{t \in (j^{G_j})^g : x_t \le x_j\}.$$

If $x_j = v$, let

$$\bar{X}_P(T) = \{t \in (j^{G_j})^g : x_t \le x_j \text{ or } x_t = u\}.$$

Note that by Proposition 1,

$$X_P(T) \subseteq \bar{X}_P(T) \subseteq X_G(T).$$

Let us briefly describe how some of these notions will be utilized in SymTest. Suppose we are attempting to extend T in such a way that $g \in G(T)$ will lead to a partial solution $X^{g^{-1}}$ which is smaller than X. This will happen, for example, if we can find a t in $\bar{X}_P(T)$ such that x_t is set and $x_t < x_j$. For then, by Proposition 1, there exists an element $h \in G_r$ such that $hg \in G(T \cup t)$ and so $X^{(hg)^{-1}} \prec X$. Otherwise, assuming that $X_P(T) \ne \emptyset$, we choose the smallest element $t \in \bar{X}_P(T)$. Either $x_j < v$ and $x_t = x_j$, or $x_j = v$ and $x_t \in \{v, u\}$. Again, by Proposition 1, there exists an $h \in G_r$ such that $r^h = t$. If $x_j = v$ and $x_t = u$, we set $x_t = v$. We then make the assignments $g \leftarrow hg, T \leftarrow (T \cup t)$, in which case $g \in G(T)$, and continue the process of extending T. Eventually, we arrive at an image T such that one of the following occurs:

(i) $\bar{X}_P(T) = \emptyset$. We then set $T \leftarrow (t_1, \dots, t_{r-1})$ and attempt a further extension of the new T, or

(ii) There exists a $g \in G(T)$ such that $X^{g^{-1}} \prec X$, or

(iii) T is a complete image and $x_i = x_{i^g}$ for $1 \le i \le m$.

In case(iii), there exists a unique element $g \in G(T)$ (by the definition of a base) and g has the property that $x_i = x_{i^g}$ for each $i \in \Gamma$. If $k \le m$, then $g \in C_\Omega(G)$; otherwise we need to check g on the remaining values $\{m+1, \dots, k\}$ to see if $g \in C_\Omega(G)$. The net effect is that either we find a smaller partial solution than X, we find an element of $C_\Omega(G)$ or we backtrack on T.

As might be expected from this discussion, SymTest is itself constucted as a backtracking algorithm and systematically searches through all possible partial images T. Furthermore the algorithm is organized in such a way that not only do we output a set of strong generators for H, but we also use the subgroups of H that are built up enroute in order to prune the sets $\bar{X}_P(T)$. A careful analysis is required, however, to ensure that we don't at the same time throw out any partial image which corresponds to an element $g \in G$ such that $X^{g^{-1}} \prec X$.

Section 6. A Criterion for Turning SymTest Off

In Section 2 we described the useful work performed by SymTest as we successively examine the values $x_{j^g}, 1 \le j \le k$, for $g \in G$. As variables are set, it becomes more difficult to find an element $g \in G$ which sets new variables or has the property that $X^{g^{-1}} \prec X$. We present a simple criterion for deciding when further use of SymTest will yield no further productive work until we backup past this point. Once this criterion has been met, we may then switch from symmetry checking to another intelligent search method such as search rearrangement.

Suppose that we have set the variables $x_1, \dots, x_k$ and that C is the current coloring of Ω. We say that *Condition U* is *true at k* if for each $j \le k$ and $g \in G$ such that x_i and x_{i^g} have the same color for $1 \le i \le j-1$, $x_{j^g} \ne u$. In the case where $x_j = v$ and $x_{j^\sigma} = u$, we immediately set $x_{j^\sigma} = v$ in our backtracking algorithm, so that this case won't invalidate *Condition U.*

Turnoff SymTest Criterion: *Condition U* is true and $C_\Omega(G)$ is the identity subgroup.

If this criterion is true, then no further information will be gained by applying SymTest until the next time we examine variable x_k. To see why this criterion works, suppose that in the course of searching deeper in this branch, and after setting x_j for some $j > k$, we find an element $g \in G$ such that $X^{g^{-1}} \prec X$. This means there exists an $r \le j$, such that $x_i = x_{i^g}$ for $1 \le i \le r-1$ and $x_r < x_{r^g}$.

We present a proof by contradiction, that no such g exists. This proof depends on the following observations.

Observation I. If any variable x_i is set to v in C. for $i > k$, then x_i is set to v for the coloring C' which arises from setting the additional variables $x_{k+1}, \dots, x_j$.

Observation II. If i satisfies $1 \le i \le \min(k, r-1)$ then i and i^g have the same color in C.

Proof. By the definition of g, i and i^g have the same color in C' for $1 \le i \le r-1$. Furthermore C and C' agree on $\{1, \dots, k\}$. Let $1 \le i \le min(k, r-1)$. If $i^g \le k$, then both i and i^g have the same color in C. Otherwise, choose i to be minimal such that $i^g > k$. Then i^g is colored u or v in C. If i^g is colored u in C, then we can use g to obtain a contradiction to our assumption that *Condition U* is true. Thus i^g is colored v in C. But then i^g is colored v in C' by Observation I and so i is colored v in C', and therefore in C as well. The observation then follows by an easy induction argument.

We now derive a contradiction to the existence of g. If $r > k$, then g preserves C by Observation II. But then g must be the identity element by our Criterion. Thus we may assume that $r \le k$. By Observation II, we know that i^g has the same color as i in C for $1 \le i \le r-1$. If $r^g \le k$, then r^g has the same color in C as in C'. Since, $x_{r^g} < x_r$, it then follows that $X^{g^{-1}} \prec X$ in C. The existence of such a g

would have caused our backtracking program to abandon the present branch rather than attempting to extend C.

Thus we may assume that $r^g > k$. This implies that r^g is colored u or v in C. The case where $x_{r^g} = u$ is a contradiction to our Criterion that *Condition U* holds at k. On the other hand, if r^g is colored v in C, then by Observation II, r^g is colored v in C' as well. This contradicts $x_{r^g} < x_r$ in C' and completes the proof.

Acknowledgement. Our thanks to Hugh Brown for making the diagrams.

References

[1] A. Aho,J. Hopcroft and J. Ullman, *The design and analysis of algorithms*, Addison Wesley, Reading, 1974

[2] M. D. Atkinson (editor), *Computational Group Theory*, Academic Press, New York, 1984.

[3] L. Babai, W. M. Kantor, E. M. Luks, *Computational complexity and the classification of finite simple groups* , Proc. **24**th IEEE FOCS, (1984), 162-171.

[4] J. Bitner and E. Reingold, *Backtrack programming techniques*, CACM **18** (1975), 121-136.

[5] C. Brown, L. Finkelstein, and P. Purdom, *Symmetry and Searching*, Northeastern University Technical Report (to appear).

[6] G. Butler, *Computing in permutation and matrix groups II: backtrack algorithm* Math. Comp., **39** (1982), 671-680.

[7] G. Butler and J. J. Cannon, *Computing in permutation and matrix groups I: normal closure, commutator subgroups, series* Math. Comp., **39** (1982), 663-670.

[8] J. J. Cannon, *An introduction to the group theory language, Cayley*, in *Computational Group Theory*, edited by M.D. Atkinson, Academic Press, 1984, 145-184.

[9] E. Freuder, *Utilizing Subgraph Isomorphism in Constraint Graphs*, University of New Hampshire Technical Report 84-13.

[10] M. Furst, J. E. Hopcroft, E. M. Luks, *Polynomial-time algorithms for permutation groups* , Proc. **21**th IEEE FOCS, (1980),36-41.

[11] Z. Galil, C. M. Hoffman, E. M. Luks, C. P. Schnorr, A. Weber, *An $O(n^3logn)$ deterministic and an $O(n^3)$ probabilistic isomorphism test for trivalent graphs*, Proc. **23**rd IEEE FOCS, (1982), 118-125.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W.H. Freeman, San Franciso (1979).

[13] C. M. Hoffman, *On the complexity of intersecting permutation groups and its relationship to graph isomorphism*, manuscript.

[14] C. M. Hoffman, *Group-theoretic algorithms and graph isomorphism*, Lecture Notes in Computer Science,**136**, Springer-Verlag, Berlin, 1982.

[15] J. Leon, *Computing automorphism groups of combinatorial objects*, in *Computational Group Theory*, edited by M.D. Atkinson, Academic Press, 1984, 321-337.

[16] E. M. Luks, *Isomorphisms of graphs of bounded valence can be tested in polynomial time*, J. Comp. Sys. Sci. **25** (1982), 42-65.

[18] P. W. Purdom, E. L. Robertson, and C. A. Brown, *Backtracking with multiple level search rearrangement*, Acta Informatica (1981).

[19] C. C. Sims, *Computation with permutation groups* in *Proc. Second Symposium on Symbolic and Algebraic Manipulation*, edited by S. R. Petrick, ACM, New York, 1971.

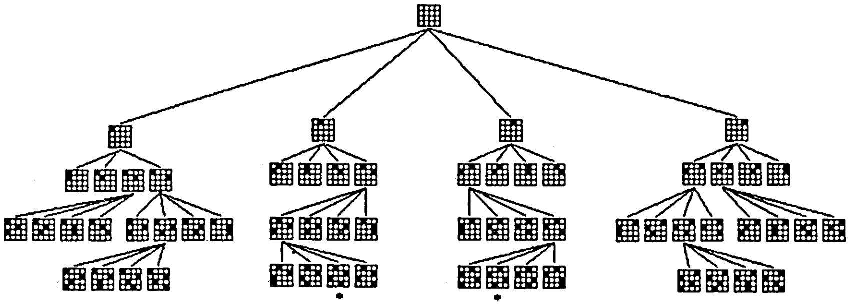


Fig. 1. The ordinary backtrack tree for the four queens problem. Solutions are marked with an asterisk.

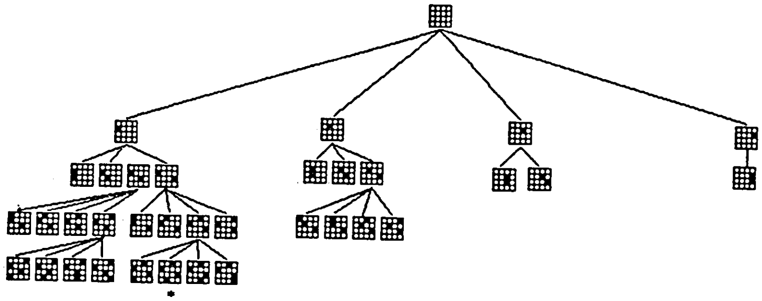


Fig. 2. The backtrack tree for the four queens problem using row symmetries and an alternate search order.

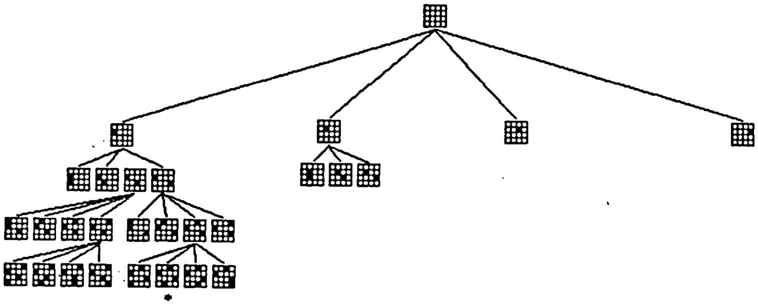


Fig. 3. The backtrack tree for the four queens problem using the full symmetry group and the alternate search order.

TIME-SPACE TRADEOFFS FOR TREE SEARCH AND TRAVERSAL

David A. Carlson

Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003

Abstract

Tree search is a fundamental computation that is performed frequently in the application areas of combinatorial optimization and artificial intelligence. There, algorithms that search the state-space tree representation of a problem instance for a solution of minimum cost often use prohibitively large amounts of space. In this paper, we develop new algorithms for tree search that use reduced amounts of space. The algorithms are modifications of depth-first, breadth-first, and heuristic tree search algorithms. They achieve their space reductions by recomputing certain nodes in the tree from scratch. Thus, they require more time than the algorithms they are derived from. However, the time-space tradeoff is a favorable one, since we are able to show that significant savings in space can be achieved at the expense of only a small (constant factor) increase in run time.

1. Introduction

Searching for an item in a set of items is a problem that arises often in computer applications. Since it is a fundamental problem that must be solved frequently, researchers have attempted in the past to develop algorithms with good performance, either in terms of time or space. These are the major resources consumed by a computer algorithm, and developing algorithms having better performance with respect to them leads to a reduction in the overall cost of computing.

Here, we will consider the problem of tree search, which occurs when the set of items to be searched is organized as a tree, as shown in Figure 1. We assume that each item in the tree has associated with it a nonnegative integer cost, and that the problem is to determine the minimum cost item in the tree. Little or no information about the minimum is known in advance, so that it may be necessary to search the entire tree structure in order to guarantee that the minimum has been found. In this situation, tree search can be accomplished by an orderly traversal technique that visits all nodes in the tree.

The problem of searching for a minimum in a tree arises often in a number of important application areas including combinatorial optimization and artificial intelligence. In combinatorial optimization, problems often have associated with them a solution space that is organized as a tree, and the tree must be searched in order to find a solution that optimizes a given criteria. Artificial intelligence problems often take on the same flavor, where a tree structure must be searched in order to find a goal state that minimizes a given objective function. An important property of the tree structures arising in the above applications is that the costs of nodes along a path from the root to a leaf are nondecreasing. Heuristic search algorithms take advantage of this property by maintaining a current minimum and eliminating from the search process subtrees whose nodes exceed in cost the value of the current minimum. Even though a fairly large subset of the tree may not be searched, heuristic search terminates having found a node of guaranteed minimum cost. It shares this property with other algorithms that do a full search of the tree.

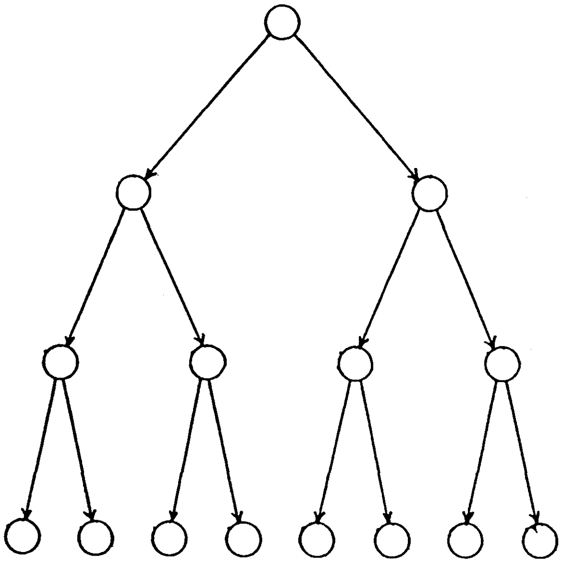


Figure 1: A complete binary tree

CH2345-7/86/0000/0585$01.00 © 1986 IEEE

In this paper, we will explore the possible tradeoffs between time and space that exist for different algorithmic implementations of full and heuristic tree search. We will develop algorithms that use reduced amounts of space to search a tree for its node of minimum cost, and analyze their time performance. While an increase in time is required in order to achieve a decrease in space, we are able to show that the tradeoff is a favorable one. That is, only a small, constant factor increase in time is associated with algorithms that achieve significant reductions in space. Our results for heuristic search are particularly interesting, since standard heuristic search algorithms often use prohibitively large amounts of space to find a minimum cost item in the tree.

The algorithms that we propose here are variations of full search (either depth-first or breadth-first) and heuristic search applied to tree structures. For depth-first search, instead of saving pointers into the tree on a stack, we recompute them from scratch when required. We further show that breadth-first and heuristic search can be realized using multiple depth-first searches. Thus, the results we obtain for reduced space depth-first search also hold true for reduced space breadth-first and heuristic search.

Previous research on tree search has been fairly limited due to the widespread acceptance of the standard depth-first and breadth-first algorithms. These methods and the potential they offer in solving combinatorial optimization problems are surveyed in most intermediate level textbooks on computer algorithms, including [1,2,4,8]. Knuth [4] goes further to explain methods that eliminate the need for the separate stack used by depth-first search. These methods usually are employed when the tree has grown to exhaust all available space, as often happens in list processing systems that employ garbage collection. Instead of a stack, they use a small amount of extra space in each node of the tree.

Recently, Korf [3], in the context of AI applications, has considered multiple depth-first search to successive levels in a tree and its viability as an alternative to breadth-first search. He also generalizes these ideas to obtain reduced space heuristic search algorithms that can be applied to a wide variety of AI problems. On tree structures whose number of nodes is an exponential function of their depth, Korf argues that a multiple depth-first search strategy is much better in terms of space performance than breadth-first and heuristic search and only increases the number of node expansions slightly.

Another reference where work appears that is related to the results presented here is Pippenger [6]. There, a tradeoff between time and space is proven for a particular problem using a theoretical framework referred to as pebbling. The problem considered by Pippenger is a special case of full tree search where the leaves of the tree must be computed in a certain order.

Here, we will generalize the results of Pippenger to tree structures where the leaves can be computed in any desired order. We will use the theoretical framework of pebbling as a tool to explain our methods in an abstract setting, and to analyze their performance with respect to time and space. The end result will be algorithms similar to those presented in Korf [3] for both full and heuristic tree search that combine significant reductions in the usage of space with small increases in running time.

This paper is organized as follows. Section 2 explains the concept of pebbling and other preliminaries needed in the remainder of the paper. In Section 3, we present reduced space tree search methods based on combining depth-first search on subtrees with the recomputation of certain nodes in the tree. Sections 4 and 5 show how these ideas can be extended to obtain reduced space emulations of breadth-first and heuristic search. Conclusions and directions for further research are given in Section 6.

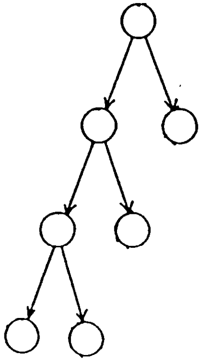
2. Preliminaries

This section explains the basic notion of a tree and the concept of pebbling, which we will use to "compute" the nodes of a tree in an algorithmic manner to be described later in the paper.

Throughout this paper, we will restrict our attention to complete binary trees that have edges directed out from a distinguished root node into the roots of two complete subtrees. In general, such a tree has a branching factor of 2 at each node, and if it has depth d, then it contains 2^d leaf (terminal) nodes. Depth refers to the length, i.e. number of edges, of the longest path from an input (the root) to an output (a leaf). Figure 1 illustrates a complete binary tree of depth 3 (8 leaves).

Our reason for concentrating only on complete binary trees is based on the following observations. First of all, such trees preserve an exponential relationship between the depth of the tree and the number of leaves in the tree, i.e., #leaves=exp{depth}. Other tree structures, such as a serial, chain-like tree (shown in Figure 2), do not have this property. Search trees having this exponential relationship often arise in the modeling of combinatorial optimization problems and AI problems. Usually, the depth of the tree structure corresponds to the number of inputs to the problem (one level for each input), and the difficulty in solving the problem stems from the fact that an exponential number of solutions (settings of the inputs) must be checked in order to find the one which optimizes a given objective function. In the search tree, a node represents a partial setting of values to inputs, and each node has a branching factor b>1; i.e. there are a multitude of values that can be assigned to the next input to obtain a new, more complete partial setting.

Figure 2: A serial, chain-like binary tree



While there are many tree structures that are not complete, but for which #leaves=exp{depth}, such trees are reasonably "close" to a complete binary tree, so that significant information regarding them can be obtained by analyzing only the special case of a complete tree. This will be our approach in this paper: to consider in detail the case of a complete binary tree, which leads to meaningful insights concerning other similar tree structures. The same can be said regarding the branching factor of the tree. Here, we consider only the branching factor b=2, but the results we derive can also be shown to hold true for other branching factors b>1.

At this point, we outline the concept of pebbling, which forms a useful abstraction of how the search of a tree is performed. The "pebble game" has been used mainly in theoretical computer science as a paradigm for analyzing the simultaneous time and space requirements of a computation. First, a graph representation of a computation is formed that models the data dependencies inherent in the computation. On this graph representation, pebbles are then moved from inputs to outputs according to certain rules (hence the term "pebble game"). These rules reflect how a uniprocessor would perform a straight-line computation, the main one being that a non-input node can be pebbled only when all its predecessors hold pebbles. In other words, the uniprocessor can only compute an intermediate result when all the values it depends on are held in temporary storage locations.

An algorithmic solution to the problem being modeled in a pebble game analysis corresponds to a pebbling strategy that places pebbles on each output node at certain points in time. The space requirement of the algorithm is simply the maximum number of pebbles used during this process, and time is measured by the number of movements of pebbles made on nodes of the graph. When the number of pebbles available is reduced, certain nodes in the graph may have to be repebbled in order to pebble all the outputs. This often leads to tradeoffs between time and space for a given computational problem. A good survey of the vast amount of theoretical work that has been done on this subject appears in Pippenger [7]. Here, we will apply the pebble game analysis technique to the problem of searching for the minimum cost item in a complete tree.

As a final point in this section, we will use the big-O notation for expressing the asymptotic behavior of a function. The relevant symbols are:

O $f(n)=O(g(n)$ iff there are constants c and N such that for all $n \geq N$, $f(n) \leq g(n)$

Ω $f(n)=\Omega(g(n))$ iff there are constants c and N such that for all $n \geq N$, $f(n) \geq g(n)$

Θ $f(n)=\Theta(g(n))$ iff $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$

3. Depth-First Search with Reduced Space

Depth-first search is a well known technique for visiting all the nodes in a graph. When the graph is a tree, it operates by expanding the most recently encountered node until a leaf is reached, and then backtracking to nodes previously encountered having descendants not already expanded. In practice, a stack is used to guide the backtracking process, with a node being placed on the stack when the search procedure moves to one of its descendants. A node is taken off the stack once a leaf is reached. For a complete binary tree of depth d, a stack of size d is required, and the search procedure uses time proportional to the number of nodes in the tree.

Standard depth-first search can easily be viewed as a pebbling strategy for a complete binary tree. We now outline such a strategy to give the reader a feel for the concept of pebbling. The strategy works by placing pebbles on nodes of the tree starting at the root and working towards the leaves. A pebble is held on a node until the point in time when all leaves in the subtree rooted at that node have been pebbled. Further placements of pebbles on the tree are made from the node deepest in the tree that holds a pebble. If its left child has not been pebbled, then a pebble is placed there, otherwise a pebble is placed on its right child. Figure 3 illustrates a configuration of pebbles in this strategy on a binary tree with 16 leaves.

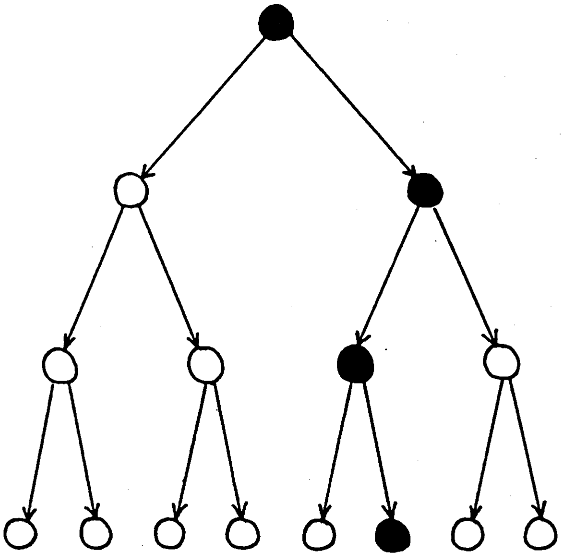


Figure 3: A configuration of pebbles in the depth-first strategy

The pebbling strategy described above emulates depth-first search by making further advances into the tree from the deepest of the nodes that currently hold pebbles. The nodes holding pebbles can be thought of as the contents of the stack in depth-first search, with the node most recently pebbled (i.e. most recently placed in the stack) being selected as the one to be further expanded.

The time and space requirements of our depth-first pebbling strategy are easily analyzed. If a complete binary tree of depth d and 2^d leaves is pebbled using this strategy, the at most d+1 pebbles are used at any point in the strategy, which is the number of nodes in a path from the root to any leaf in the tree. The number of pebble placements or moves made is simply the number of nodes in the tree (each is pebbled exactly once), which can be expressed by the formula

$$T=1+2+2^2+\ldots+2^d=2^{d+1}-1=O(2^d)$$

We note that the strategy is optimal in terms of its time performance, since any strategy must make at least one pebble placement on each node of the tree.

Complementary to the notion of a pebbling strategy that runs in optimal time is one that uses an optimal amount of space. We now outline such a strategy. First, we claim that any leaf in the tree can be pebbled using only one pebble, by simply advancing a single pebble down the unique path that connects the root to the leaf (we are allowing "sliding" in the pebble game, i.e. the movement of a pebble on a predecessor directly onto the node being pebbled). Once the single pebble being used is placed on a leaf, it is not of use in the pebbling of any other leaf, so that to pebble the next leaf the same process must be repeated in its entirety. Doing this for 2^d leaves in the tree, it is easily seen that $T=(d+1)2^d$ pebble placements are made by this strategy.

In comparing the two pebbling strategies outlined above, an intuitive tradeoff between time and space can be observed. A decrease in space from S=d+1 to S=1 requires an increase in time from $T=O(2^d)$ to $T=\Omega(d2^d)$. If we normalize time and space to be functions of the number of nodes in the tree $N=2^d$, then the standard depth-first strategy has $T=O(N)$, $S=\Theta(\log N)$ and the minimum space strategy has $T=\Omega(N\log N)$, S=1.

At this point, we note that if the minimum space strategy is to be implemented in practice, then a means for "guiding" a single pebble down the tree multiple times must be provided. This is easily done using d bits to encode the 2^d unique paths from the root to the leaves. A specific setting of these bits determines a path to move a pebble down, so that once the pebble reaches a leaf, the setting of the d bits must be updated to give a new path to a leaf that has not yet been pebbled. This additional d bit field needed to guide the search is not a lot of overhead, especially when one considers that a pebble must consist of at least d bits in order to properly distinguish between all the different nodes in the tree. Taking this into account, the minimum space strategy requires O(d) bits to implement, whereas standard depth-first search requires $O(d^2)$ bits (O(d) pebbles each requiring O(d) bits).

From a theoretical standpoint, the minimum space strategy outlined above is interesting since it provides an alternative to standard depth-first search. However, in practice, if time $O(2^d)$ can be afforded, then d is relatively small, and the achievable space savings are not that large. For now, the strategy should be viewed more as the exposition of a methodology that reduces space by allowing recomputation. Applying the methodology to other well-known search algorithms has the potential of yielding more significant space savings, as will be seen later in this paper.

We continue our discussion of reduced space depth-first search algorithms by describing methods that fall in between the standard and the minimum space strategies outlined above. As with the minimum space strategy, they are interesting mainly from a theoretical perspective, so the more practically oriented reader may wish to skip ahead to Section 4.

While the minimum space strategy requires time that is non-linear in the size of the tree, it may be that there are other strategies which reduce space yet maintain time $T=O(2^d)$. We claim that such strategies exist, and they use significantly less space than the standard depth-first strategy. We are able to construct them by combining the minimum space strategy with the standard depth-first strategy on appropriate subtrees in the tree structure.

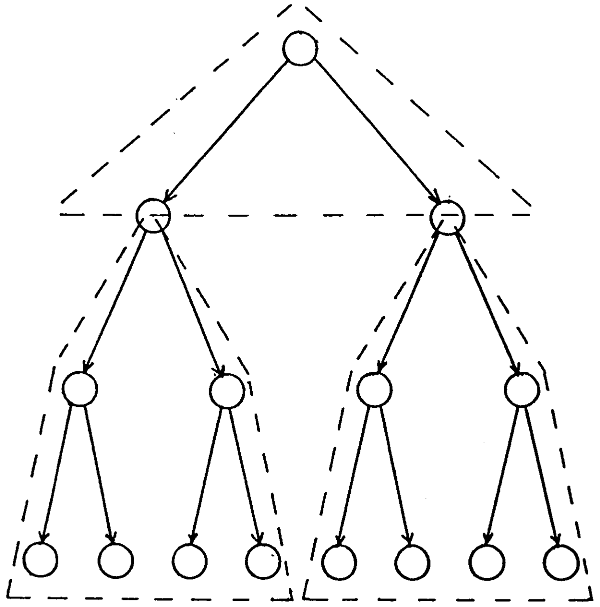
To aid in viewing these new pebbling strategies, we decompose a tree of depth d into a top subtree of depth k, $0\leq k\leq d$, whose leaves are the roots of 2^k disjoint subtrees of depth d-k, as shown in Figure 4. On the top subtree, we will employ the minimum space strategy. Upon reaching one of the top subtree's leaves, we will hold a pebble there while employing the standard depth-first strategy on the bottom subtree. This is repeated for each of the 2^k bottom subtrees rooted at a leaf of the top subtree.

The time and space requirements of this new pebbling strategy can be analyzed as follows. First, d-k+1 pebbles are used, a single one on the top subtree and d-k additional ones on each bottom subtree. Second, the number of pebble placements made can be expressed as follows

$$
\begin{aligned}
T&=(k+1)2^k && \text{(total for top subtree)}\\
&\quad+2^k(2^{d-k}-1) && (2^{d-k}-1 \text{ for each bottom subtree})\\
&=O(k2^k+2^d)
\end{aligned}
$$

The extremes of this new pebbling strategy are k=0, which is the standard depth-first strategy, and k=d, which is the minimum space strategy.

Figure 4: The decomposition of a tree into a top subtree and bottom subtrees



However, it is interesting to observe the time requirements of the strategy for different values of k between 0 and d. For example, if k=d/2 then half the space of standard depth-first search is used, and time remains $O(2^d)$. We can further reduce space to S=logd and still maintain "linear" time. This follows by choosing k=d-logd, i.e. we pebble bottom subtrees of depth logd using the standard depth-first strategy. Time remains proportional to 2^d, since

$$T=O(k2^k+2^d)=O((d-\log d)2^{d-\log d}+2^d)=O(2^d)$$

If space is reduced slightly below S=logd in the above strategy, say to S=(logd)/2, then we obtain $T=O(d^{1/2}2^d)$, which is non-linear in the size of the tree. Thus, we can conclude that further reductions in space below S=logd cannot be made in this pebbling strategy if time $T=O(2^d)$ is to be achieved.

In order to achieve further reductions in space, we modify the way that bottom subtrees are pebbled in our new pebbling strategy. Instead of using a standard depth-first strategy, we apply our new pebbling strategy to each bottom subtree. This decomposes each bottom subtree in a manner similar to the way the entire tree was initially decomposed. For a bottom subtree of depth logd, we break it into a top subsubtree of depth logd-loglogd, below which are multiple bottom subsubtrees each of depth loglogd. We employ a minimum space strategy on the top subsubtree and a standard depth-first strategy on each bottom subsubtree to pebble the entire subtree in time $T=O(2^{\log d})$. It easily follows that the strategy for the entire tree uses space S=loglogd+2 and runs in time $T=O(2^d)$. Further reductions in space to S=logloglogd, etc. for which time remains $T=O(2^d)$ are straightforward and are left to the reader.

The existence of the pebbling strategies outlined above allows us to state the following theorem.

Theorem 1: A complete binary tree of depth d can be pebbled (traversed) in time $T=O(2^d)$ using space S=d, S=logd, S=loglogd, etc.

Theorem 1 shows that significant reductions in space can be made to the standard depth-first pebbling strategy. From it, the question can be asked regarding whether a pebbling strategy exists that is asymptotically optimal with respect to both time and space. In other words, can constant space ($S=O(1)$) and linear time ($T=O(2^d)$) be achieved simultaneously. In an attempt to answer this question, we now investigate pebbling strategies that use a fixed, constant number of pebbles.

We have already considered the case when S=1, and have shown that $T=\Theta(d2^d)$. The case S=2 can be handled in a manner similar to that of the new pebbling strategy presented earlier. We separate the entire tree structure of depth d into a top subtree of depth d-logd and multiple bottom subtrees of depth logd. Our pebbling strategy uses one pebble on the top subtree, which is held on a leaf of the top subtree while the bottom subtree is pebbled using the second available pebble. Analyzing the time requirements of this strategy, we obtain

$$T=(d-\log d)\ 2^{d-\log d}+2^{d-\log d}[(\log d)2^{\log d}]$$
$$=O((\log d)2^d)$$

While this is an asymptotic improvement over the time required when S=1 pebbles are used, it still is not linear in the size of the tree.

It is straightforward to extend the above strategy to S=3, S=4, ... For S=3, two cutoff points are defined so that the tree is separated into a top subtree and bottom subtrees. Each bottom subtree is separated into a top subsubtree and bottom subsubtrees. Time performance $T=O((\log\log d)2^d)$ is obtained when the cutoff points are chosen correctly. For S=4, we obtain $T=O((\log\log\log d)2^d)$, and so forth for higher values of space.

Thus, for constant values of space, we can exhibit pebbling strategies that come close to achieving "linear" time $T=O(2^d)$. The higher the constant, i.e. the more space available, the closer we can come to linear time. However, we can also prove that we cannot truly achieve linear time using constant space by showing that the pebbling strategies presented above cannot be improved upon.

Theorem 2: A complete binary tree of depth d requires pebbling time $T=\Theta((\log^{(c)}d)2^{d})$ when $S=c+1$ pebbles are used to pebble its leaves, where $c\geq 0$ is a constant value with respect to the size of the tree. Here, $\log^{(c)}d$ stands for the c^{th} iterate of the logarithm function, i.e. $\log^{(c)}d=\log(\log^{(c-1)}d)$ and $\log^{(0)}d=d$.

Proof: The upper bound $T=O((\log^{(c)}d)2^{d})$ when $S=c+1$ pebbles are used follows from the above discussion on pebbling strategies that use a fixed, constant number of pebbles. We must prove a matching lower bound of the form $T=\Omega((\log^{(c)}d)2^{d})$ to obtain the result of the theorem.

At this point, we derive a lower bound for the case $S=2$; arguments for other constant values of space are similar. Consider a subtree of depth k in a tree of depth d and the unique path from the root of the tree to the root of the subtree, as shown in Figure 5. If we choose $k=\log d-\log\log d-1$, then we can make the following statement about an optimal pebbling strategy for the tree. While the leaves of the subtree are being pebbled, an optimal strategy holds a pebble at or above the root of the subtree. To verify the claim, consider the opposite, i.e. a pebble is not held at or above the root of the subtree. Then, the path from the root of the tree to the root of the subtree must be pebbled at least twice, for a total of $2(d-\log d+\log\log d+1)$ moves. However by holding a pebble at or above the root, a number of moves $\leq (d-\log d+\log\log d+1)+(\log d-\log\log d-1)2^{\log d-\log\log d-1} \leq(3/2)d$ can be achieved. Thus, the strategy that does not hold a pebble at or above the root of the subtree clearly is not optimal.

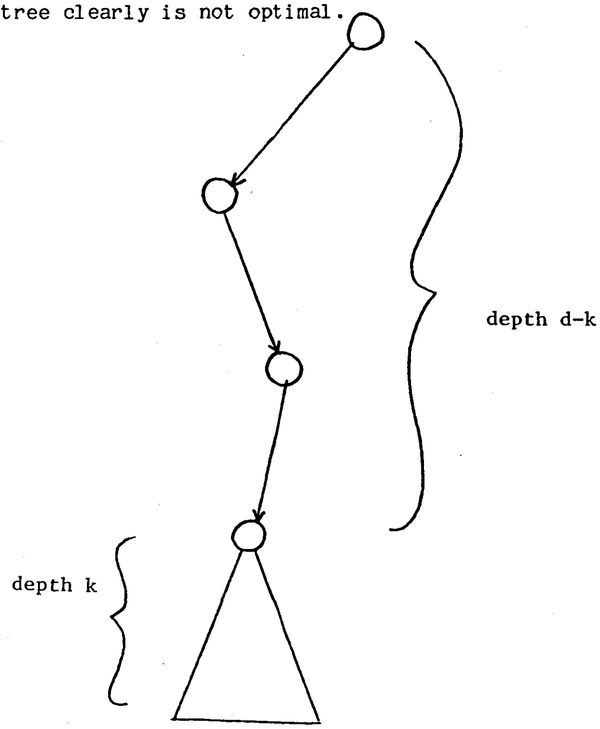


Figure 5: A subtree and its unique path to the root

From the statement above, it follows that each subtree of depth $\log d-\log\log d-1$ must be pebbled with $S=1$ in an optimal strategy. Totalling the number of moves made on these subtrees, we obtain $T=\Omega((\log d)2^{d})$, the desired lower bound. Q.E.D.

The pebbling strategies outlined above, one leading to time $O(2^{d})$ and the other leading to space $O(1)$, can both be viewed in a common light as follows. First, c cutoff points are established at levels logd, loglogd, $\log^{(c)}d$ from the bottom of the tree. Both strategies dedicate one pebble to use between each pair of different levels, and differ only how the bottom set of subtrees (below the last level) are pebbled. Using standard depth-first search on the bottom subtrees gives $T=O(2^{d})$, $S=O(\log^{(c)}d)$ while using minimum space search gives $T=O((\log^{(c)}d)2^{d})$, $S=O(1)$. Another strategy of note operates by establishing the first level at height logd from the bottom, dedicating one pebble for between the root and the first level, and pebbling subtrees below the first level recursively. Analysis similar to that conducted above reveals that this pebbling strategy requires time $T=O(2^{d}\log^{*}d)$ and space $S=O(\log^{*}d)$, where $\log^{*}d$ is the number of applications of the logarithm function required to produce a value ≤ 2. The function $\log^{*}d$ increases at a very slow rate, so that for all practical purposes this strategy can be thought of as achieving linear time and constant space simultaneously.

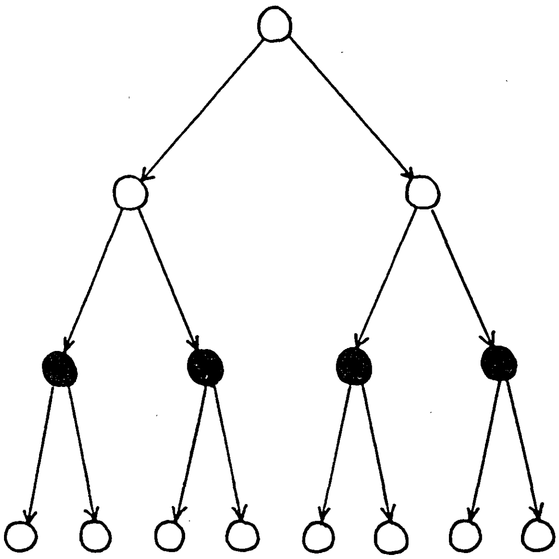
To summarize this section, we have identified time-space tradeoffs for tree search and traversal using the pebble game technique. Our methods show that reductions in space, almost down to a constant value, can be made while still preserving time that is asymptotically optimal, i.e. proportional to the size of the tree. However, there is indeed a time-space tradeoff inherent in the tree search problem, as our results also show that optimal time and optimal space (a constant with respect to tree size) cannot be achieved simultaneously.

4. Breadth-First Search with Reduced Space

In this section, we generalize the ideas developed in Section 3 to breadth-first search of a complete binary tree. We are able to show that significant reductions in space can be made while maintaining time proportional to the size of the tree.

Breadth-first search complements the notion of depth-first search in that it operates by expanding the least recently encountered node in the tree instead of the most recently encountered one. This is done until all children of the node being expanded are encountered, at which point control moves to the next least recently encountered node. In practice, nodes are put on a queue as they are encountered (during the expansion of their parent), and are taken off the queue when they are later chosen for expansion.

Figure 6: A configuration of pebbles in the breadth-first strategy



Breadth-first search can be viewed as a pebbling strategy for a complete binary tree as follows. Starting at the root of the tree, each successive level is pebbled in its entirety before any pebbles are placed on the next level. The expansion of a node corresponds to placing pebbles on both of its two children. Once a node is expanded, it no longer holds a pebble. Figure 6 shows a "snapshot" in the middle of such a strategy where pebbles have been moved from one intermediate level to another.

The time and space requirements of a breadth-first pebbling strategy for a complete tree of depth d are easily seen to be $T=O(2^d)$ and $S=O(2^d)$. Time equal to the number of nodes in the tree is used, which cannot be improved upon, however an enormous amount of space is also used by this strategy. It would be desirable to design a new search strategy that expands the nodes in the tree in the same order as the breadth-first strategy, but which uses significantly less space.

Korf [3] discusses a method for emulating breadth-first search which he refers to as depth-first iterative deepening. The basic idea is to perform multiple depth-first searches to each successive level in the tree. Each depth-first search starts fresh in the sense that it begins at the root and proceeds to visit all the nodes at a particular level in the tree. Thus, a certain amount of recomputation is required, but the strategy does perform a valid emulation of breadth-first search.

It is straightforward to view the multiple depth-first search technique described above as a pebbling strategy, and we now analyze its time and space requirements. For a tree of depth d, it has a space requirement the same as the deepest depth-first search performed, which is a depth-first search of the entire tree. Thus, space can be expressed as $S=O(d)$. The time used by the strategy is simply the sum of the time used in each depth-first search. For $k=0,1,\ldots,d$ a depth-first search to level k is performed, which requires $T\leq 2^{k+1}$ moves. Thus, the total time can be upper bounded by

$$T\leq \sum_{0\leq k\leq d} 2^{k+1}\leq 2^{d+2}=O(2^d)$$

It follows that the pebbling strategy emulates breadth-first search in an amount of time proportional to the size of the tree, while achieving a very significant reduction in space (from linear in the size of the tree to logarithmic in the size of the tree). Korf [3] concludes that the strategy is optimal with respect to space (along with time), since it uses space proportional to the logarithm of the size of the tree. However, this conclusion is incorrect, because he counts only the number of storage locations used by the strategy and ignores the bit level complexity of each storage location. In fact, we now discuss how further reductions in space can be made in pebbling strategies that emulate breadth-first search.

Since the pebbling strategy above is composed of multiple depth-first pebbling strategies, we can apply the ideas of Section 3 to even further reduce the space consumption of an emulation of breadth-first search. For example, we can replace a standard depth-first search to level k in the tree with a pebbling strategy that uses time $T=O(2^k)$ and space $O(\log k)$. Doing this for each successive depth-first search, we obtain a new emulation of breadth-first search that uses time $T=\sum_{0\leq k\leq d} O(2^k)$ $=O(2^d)$ and space $S=O(\log d)$. Generalizing this, we can state analogies to Theorems 1 and 2 for breadth-first search.

Theorem 3: The breadth-first pebbling strategy on a complete binary tree of depth d can be emulated in time $T=O(2^d)$ and space $S=d$, $S=\log d$, $S=\log\log d$, etc. using multiple space-efficient depth-first searches.

Theorem 4: The breadth-first pebbling strategy on a complete binary tree of depth d cannot be emulated in time $T=O(2^d)$ using space $S=O(1)$, i.e. space that is a constant with respect to the size of the tree.

Theorems 3 and 4 show that essentially the same results hold true for both breadth-first and depth-first search of a complete binary tree. Significant reductions in the space consumption of these methods can be made, but linear time and constant space cannot be achieved simultaneously.

5. Heuristic Search with Reduced Space

The space used by standard breadth-first search when it is applied to a complete binary tree makes it an illogical choice for actual implementation. Standard depth-first search uses much less space and does not increase the time required to search the entire tree structure. However, breadth-first search can be generalized to yield heuristic search, which has been applied to many problems having a solution space organized as a tree. Heuristic search has the potential of decreasing the time required to search the tree by eliminating parts of it from consideration. The gains in time that it offers make it desirable for use despite the space inefficiency that it shares with breadth-first search. Pearl [5] discusses and analyzes heuristic search and its application to a variety of different AI problems.

In heuristic search, it is assumed that the cost of nodes along any path from the root to a leaf are nondecreasing. The search procedure operates by assigning a cost to each node in the tree, and choosing as the next node to expand the node with the minimum cost among all those encountered so far. The nodes that have been encountered are maintained in a priority queue, from which the next node to be expanded is selected based on its cost. Expansion of a node causes its children to be inserted into the priority queue.

Heuristic search will always find the leaf in the tree (an entire solution) of minimum cost, and has the ability to eliminate the expansion of large numbers of high cost nodes in the tree. For, when heuristic search terminates, no effort will have been expended in searching subtrees whose roots, and thus all contained nodes, have a higher cost than the minimum found (this is an invariant throughout the search; no expansions are made in subtrees whose roots have a cost higher than the current minimum). This is the major advantage of heuristic search; that it has the potential to significantly reduce the time required to find a minimal cost solution. The amount of time that is saved is very dependent on the function used to determine the cost of an internal node, and the actual instance of the problem being solved.

As with the other tree search methods discussed earlier, heuristic search can be viewed as a pebbling strategy on a complete tree (a tree whose number of leaves is exponential as a function of its depth). At any point in time, the set of nodes holding pebbles forms a cut (possibly ragged) across the entire width of the tree, i.e. the tree is separated into a top subtree that has been completely explored and bottom subtrees that have not been explored at all except for their roots. The pebbling strategy operates by selecting the node in the cut of minimum cost and moving pebbles onto its two children. We can associate with a cut the value of the current minimum, so that nodes in the top subtree all have a cost less than or equal to the value of the cut and the roots of the bottom subtrees (nodes along the cut) are all candidates for selection as the value of the next cut (the next current minimum). When the heuristic search procedure selects a new current minimum, it is easily seen that the number of nodes in the new cut and in the new top subtree both increase by one, since one node is moved from the cut into the top subtree and its two children are added to the cut. Figure 7 illustrates an example of a cut arising in heuristic search. A cut like this also appears in breadth-first search, except there it is confined to at most two successive levels of the tree (see Figure 6).

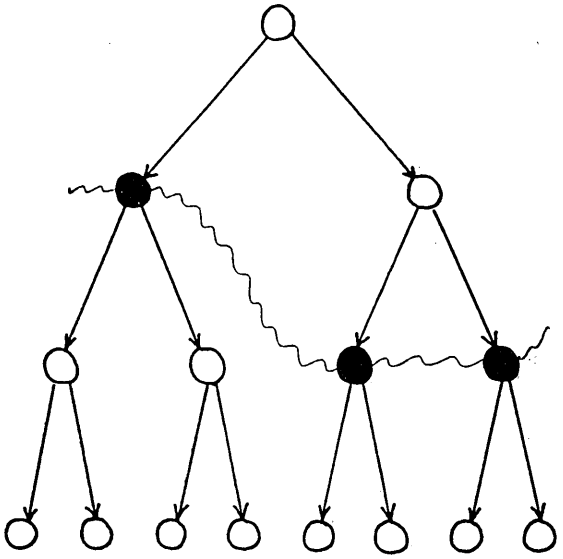


Figure 7: A cut in heuristic search

While heuristic search can decrease the time required to find a minimal cost solution, its space consumption can be large, since it uses separate storage locations organized as a priority queue to keep track of the current cut. In general, a cut across a complete tree can require a number of nodes proportional to the size of the tree, which explains the likelihood of requiring a large amount of space. Thus, we are motivated to reduce the space consumption of heuristic search without significantly increasing the number of nodes expanded. To achieve this, we propose techniques similar to those given earlier for breadth-first search that are based on performing multiple depth-first searches into the tree structure.

We start by considering the top subtree that lies above the current cut in a standard heuristic search procedure. Instead of maintaining the cut with separate storage locations for each node it contains, the top subtree can easily be searched in its entirety using a depth-first procedure that does not expand beyond nodes whose values exceed the current minimum. By keeping track of the lowest cost node whose value exceeds the current minimum (i.e. it is on the "fringe" of the top subtree) while performing such a depth-first search, the value of the next minimum that standard heuristic search would find is obtained. Executing multiple depth-first searches in this manner yields a valid emulation of heuristic search. In essence, what we are doing is a depth-first search of the top subtree associated with each cut that arises in standard heuristic search.

The strategy explained just above for emulating heuristic search is also presented in Korf [3]. There, it is claimed that the multiple depth-first emulation is optimal in time performance, i.e. it is within a constant factor of the run time of standard heuristic search. This claim is incorrect, as we will now demonstrate. Assume that standard heuristic search makes ℓ node expansions in searching a tree structure. Each time a node expansion is made, both the top subtree and the cut associated with a new current minimum grow by one node. Thus, multiple depth-first searches are performed on top subtrees of size (number of nodes) $1,2,\ldots,\ell$. Each depth-first search requires time proportional to the size of the current top subtree, resulting in total time $T=1+2+\cdots+\ell=O(\ell^2)$. Other strategies must be devised if time $O(\ell)$ is to be achieved.

The emulations of breadth-first search presented in Section 4 were based on moving from one level to the next in the tree structure. Time was increased by just a constant factor due to the fact that an entire top subtree located above a certain level only has twice as many nodes as its bottom level. Rather than maintain the bottom level using a large amount of space, the entire top subtree could be depth-first searched in proportionately the same amount of time.

With the above observations in mind, we propose the following strategy for emulating standard heuristic search that operates by moving from level to level in the tree structure. For each successive level, the node of minimum cost at that level is found. This is done by first estimating the minimum cost of the nodes at a particular level. Such an estimate is easily determined by examining children of the minimum cost node found at the previous level in the tree. Then, a depth-first search to the desired level is performed. During the depth-first search, a current minimum for the level is maintained (it initially is the estimate), and nodes of cost greater than the current minimum are not expanded.

<u>Algorithm for Heuristic Search with Reduced Space</u>

```
** initialize current minimum **
currentmin:=root;
** perform multiple depth-first searches to each
   level **
for k:=2 to d do
  ** calculate initial estimate of minimum cost
     node for level k **
  if cost(left(currentmin))<cost(right(currentmin))
     then currentmin:=left(currentmin)
     else currentmin:=right(currentmin);
  ** depth-first search to level k **
  call depthfirst(root,k);
```

```
** Recursive procedure for depth-first search
   called above. Cuts off the search when at
   bottom level (j=0) and when the cost of a node
   encountered is greater than current minimum **
procedure depthfirst(node,j)
   if j=0 and cost(node)<cost(currentmin)
      then currentmin:=node;
   if j>0 then
      if left(node) not visited and
         cost(left(node))<cost(currentmin)
         then depthfirst(left(node),j-1)
          else if right(node) not visited and
                 cost(right(node))<cost(currentmin)
                 then depthfirst(right(node),j-1)
```

In standard heuristic search, when the minimum cost node at a certain level is first found, a cut exists whose associated top subtree contains that node but no nodes at any deeper level. Our new strategy for performing heuristic search is most easily understood as trying to use depth-first search with a current minimum to efficiently explore this top subtree. The true minimum for the level cannot be known in advance and thus an initial estimate must be generated from the previous iteration of the algorithm.

We now discuss the time performance of our new emulation of heuristic search. First, we note that because the algorithm is based on moving from level to level, it has the potential for achieving run time only a constant factor above that of standard heuristic search. There are two factors that affect the run time of the new algorithm and make it difficult to analyze exactly. One is the fact that an estimate of the true minimum at a level must be used; the other is that the shape of the top subtree can vary away from being complete.

Since we are using estimates of the true minimum during a depth-first search to define something close to the actual cut generated in standard heuristic search, the boundaries of the subtree corresponding to the actual cut may be exceeded when it is explored. But if the estimates converge quickly to a good estimate (i.e. one close to the actual lowest cost), then only a small amount of extra expansions will be made. Also, if the top subtree remains roughly exponential in size as a function of its depth, then space is reduced from being proportional to the size of the subtree to the logarithm of its size, and the total time required is only a constant factor increase over that used by standard heuristic search. Thus, our new heuristic search method holds the promise of exhibiting a very favorable tradeoff of time for space, i.e. significant reductions in space consumption are possible at the expense of only a small increase in time.

Extensions to our new algorithm for heuristic search can be made in an effort to address the two factors mentioned above. To decrease the effect of estimating the true minimum, additional expansions can be made from nodes of cost close to the minimum found at the previous level. The possible noncompleteness of the top subtree can be addressed by performing fewer depth-first searches with each depth-first search extending out to a level more than one unit away from the previous level. Descriptions of these extensions similar to the algorithm presented above are left to the interested reader.

We conclude our discussion of heuristic search by noting again that it is difficult to obtain exact expressions for the time and space requirements of our new algorithm.. Empirical studies should be done to quantify the time-space tradeoffs that exist in it, and probabalistic methods offer a possible way of analyzing the variety of different tree structures that can arise when the algorithm is executed.

6. Conclusions

In this paper, we have analyzed the time and space requirements of tree search, a fundamental computational problem. We found that significant reductions in the space used by standard depth-first, breadth-first, and heuristic search can be made with the expense being only a small constant factor increase in the time required by the search strategy. However, there is a time-space tradeoff inherent in this problem, since we were able to show that strategies using minimum space require time greater than linear in the size of the tree.

The methods we have proposed seem to hold the most promise for heuristic search, since standard implementations of heuristic search reduce the time required by exhaustive search but are often hampered by large space requirements. Future research should be directed towards implementing these reduced space heuristic search methods in practice, so as to judge their actual time performance in comparison to that of standard heuristic search.

References

1. A. Aho, J. Hopcroft, and J. Ullman, Data Structures and Algorithms, Addison-Wesley, Reading, MA, 1983.

2. E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Rockville, MD, 1978.

3. R.E. Korf, "Depth-First Iterative Deepening: An Optimal Admissible Tree Search, Artificial Intelligence, Vol. 27, pp. 97-109, 1985.

4. D.E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, Reading, MA, 1973.

5. J. Pearl, Heuristics, Addison-Wesley, Reading, MA, 1984.

6. N. Pippenger, "A Time-Space Tradeoff," J. Assoc. Comp. Mach., Vol. 25, pp. 509-515, 1978.

7. N. Pippenger, "Pebbling," IBM-Japan Symp. Math. Found. Comp. Sci., 5, 1980.

8. R. Sedgewick, Algorithms, Addison-Wesley, Reading, MA, 1983.

A FAST PROBABILISTIC ALGORITHM FOR FOUR-COLORING LARGE PLANAR GRAPHS

Raymond A. Archuleta
Henry D. Shapiro

Department of Computer Science
University of New Mexico
Albuquerque, New Mexico 87131

The problem of four-coloring planar graphs — assigning colors to vertices so that no two adjacent vertices have the same color — is widely known. Backtracking algorithms guaranteed to four-color a graph, even those using sophisticated ordering heuristics, have very large running times on graphs of even moderate size (300–400 vertices). Numerous approximation algorithms have appeared in the literature, though they invariably fail to four-color large planar graphs[8]. In recent work, Morgenstern and Shapiro[5], give an approximation algorithm that experimentally exhibits no failures on graphs of up to 5000 vertices, but which is observed to have non-linear running time. In this paper we present a probabilistic algorithm for rapidly four-coloring large planar graphs. Extensive experimental tests indicate no failures on graphs of up to 15,000 vertices and a roughly linear running time over a wide range of sizes. For graphs of comparable size the algorithm is 100 times faster than any reported in the literature.

INTRODUCTION

Though the long standing four-color conjecture was proved by Appel and Haken in 1976, their proof does not provide a computationally feasible approach to four-coloring large planar graphs. Because of the large running times of exact algorithms based on backtracking, most of the attention of researchers has been focused on algorithms that are rapid, but do not necessarily produce a four-coloring. To put our new approach in perspective and because we use some of the terminology and ideas, we briefly review the strategies of others. One class of approximation algorithms is based on the "greedy" method — the vertices are processed in order, with each vertex assigned to the lowest numbered color class that does not place it in conflict with its previously colored neighbors. Because the vertices adjacent to the current vertex might use all four colors, a fifth, or even sixth, seventh, etc. color may be necessary. When the greedy method needs to create a new color class the vertex is said to be at *impasse*. In an attempt to minimize the total number of color classes required, different algorithms within the general class order the vertices using various heuristics. Some well known ordering schemes are:

Largest-First: The vertices are pre-ordered from highest degree to lowest degree. Because of the large number of ties inherent in the degrees of vertices in planar graphs, which must have an average degree of less than six, a secondary ordering is used to break ties. The size of the second neighborhood, i.e., the number of vertices two steps away, is a natural choice. More sophisticated tie breaking strategies based on the dominant eigenvector of the adjacency matrix have been considered[9].

Smallest-Last: This approach is motivated by the following observation: a vertex of degree three is trivial to color, since its three neighbors can only have used at most three of the four colors. The vertex of lowest degree is placed last in the ordering list, it is then removed from the graph (reducing the degrees of its neighbors), and the procedure is applied recursively to the graph that remains. There are two ways of handling a vertex, v, of degree four so that a conflict cannot arise. One is to perform a *contraction*. Two non-adjacent vertices connected to v (there is always such a pair) are contracted into a single vertex after v is removed. That is, their adjacency lists are merged. The resulting graph is then colored, the contracted pair split apart and v reinserted. Since the two vertices that were contracted have the same color when split apart, the neighborhood of v uses at most three colors. Contraction preserves maximal planarity. A planar graph always has a vertex of degree five; if contraction is not performed, using this ordering scheme guarantees that the greedy algorithm never uses more than six colors. If we use contraction a five-coloring is guaranteed. The other method of handling vertices of degree four is based on Kempe chains and will be taken up shortly.

Saturation: Unlike the previous two schemes that statically order the vertices, saturation dynamically orders the vertices as the algorithm proceeds. The *saturation degree* of a vertex, v, is defined by Brélez[2] as the number of different colors used by vertices adjacent to v. The algorithm always

CH2345-7/86/0000/0595$01.00 © 1986 IEEE

chooses the vertex of highest saturation degree to color next, following the intuition that it has the fewest degrees of freedom. Ties are broken using a secondary ordering scheme, usually largest first.

The greedy method can be refined as follows: when a vertex is at impasse, before enlarging the number of color classes, apply a transformation to the current coloration. If the vertex is not at impasse in the transformed coloration a new color class is not needed. Such transformations are usually based on Kempe chain interchanges, first described in Kempe's false proof of the four-color theorem[7]. The i-j Kempe chain containing vertex u, of color i, is the connected component of the graph consisting of all vertices reachable from u by traversing edges that connect vertices of colors i and j. If the colors of the vertices on an i-j Kempe chain are interchanged no conflicts are created. This can be used to resolve an impasse at a vertex of degree four. Figure 1 shows how this is done. If the a-b Kempe chain containing vertex u_1 does not contain vertex u_3 then an a-b interchange frees color a for use at v. If the a-b Kempe chain contains both u_1 and u_3 then the c-d Kempe chain containing u_2 cannot reach u_4. When a vertex of degree five is at impasse, application of Kempe's method often succeeds in resolving the difficulty, though it is not guaranteed to do so. This technique was coupled with smallest last ordering by Matula[4].

Morgenstern[6] has determined experimentally that when using saturation ordering on maximally planar graphs containing no vertex of degree less than five (MPG5 graphs), both the number of vertices at impasse and the size of the Kempe chains searched during impasse resolution grow linearly with the size of the graph, though the data on the sizes of the Kempe chains is quite erratic, as the large standard deviation indicates. The Kempe chains are sometimes very short, one or two vertices, and at other times they include half the vertices of the graph. We have observed similar behavior (Table 1). This makes the overall average running time of the approximation algorithm using interchange $O(N^2)$, while the simpler, but less effective, methods are $O(N)$ (because distribution sort can be used). The method in Morgenstern[5] is an enhanced version of this Kempe interchange strategy based on the work of I. Kittell[3]. They consider sequences of Kempe interchanges should a single interchange fail to resolve the impasse. A number of heuristics to guide the search and detect repeated colorations are employed to keep down the running time. They report no failures to four-color MPG5 graphs with up to 5000 vertices. Their running times on a VAX 11/780 for problems of this size is roughly 15 minutes. For smaller graphs, with 500–1000 vertices, their running times are about one minute.

IMPASSE RESOLUTION WITH "WANDERING FIFTH COLOR"

The probabilistic coloring algorithm, which we have named *wandering fifth color*, is a compromise between the simple greedy strategies and the complex interchange methods. Like the greedy algorithms, the algorithm attempts to color the next vertex, which can be selected using any of the above ordering schemes. The idea behind impasse resolution can be seen in Figure 2. It may be the case that the color assigned to a neighbor, u, of v, the vertex at impasse, could be assigned to v if the color were not already being used to color u. The pigeonhole principle guarantees this possibility always exists if the vertex has degree seven or less. We assign to v the color of u, uncolor u and see if u can now be recolored without resorting to a fifth color. If u can be colored we have resolved the impasse. If it cannot be colored, then u is now at impasse. We have "wandered" the problem to a new location, where we repeat the process.

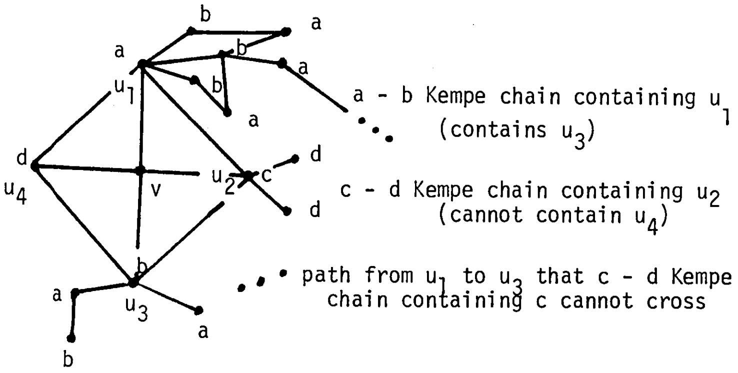

b
a
a
b
a
u_1
b
a
d
d
u_4
v
u_2
c
d
b
a
u_3
a
b
a - b Kempe chain containing u_1
(contains u_3)
c - d Kempe chain containing u_2
(cannot contain u_4)
path from u_1 to u_3 that c - d Kempe chain containing c cannot cross

Figure 1. Impasse Resolution with Kempe Chain Interchange

Several things can go wrong during this process. First we can wander into a vertex of degree eight or higher in which every color is used twice in the first neighborhood, and so reach a dead end. Even if we prohibit the vertex at impasse from immediately returning to its previous location, there is the possibility of getting stuck in an infinite cycle. We therefore terminate the wandering process if the vertex at impasse attempts to cross the path it has taken during the wandering process. This does eliminate some possibilities, since returning to a vertex does not imply returning to an isomorphic coloration. Nonetheless, this is a reasonable compromise. Of course, it is quite likely that for one of these two reasons the wandering process will fail. We therefore wander the vertex at impasse through all possibilities in a recursive (depth first) search rooted at the original impasse location. The exact algorithm, in high level pseudo-code, is given in Figure 3.

The wandering process can fail to resolve an impasse. When this occurs we abandon the entire coloring process and start over again. We include an element of randomness so that when we restart the algorithm we do not wind up in the same failed configuration. There are several places for randomness to be inserted into the algorithm:

Table 1. Experimentally Determined Properties of MPG5 Graphs

Approximate Size	Number of Impasse Vertices	*a*	Average *b*	Std. Dev. *b*	Average *c*
15000	388.50	0.0257			5.99
12500	326.40	0.0256			6.15
10000	255.90	0.0249			6.20
7500	196.00	0.0254			6.10
5000	132.34	0.0260	0.09	0.14	6.24
2500	67.69	0.0261	0.18	0.16	6.26
1000	25.94	0.0255	0.14	0.18	6.38

a = Number of Impasse Vertices / Number of Vertices

b = Size of Kempe Chain / Number of Colored Vertices at the Time Kempe Chain was Explored. Note: Average *b* is an average of ratios

c = Number of calls to ***recursivewander*** to resolve impasse or admit failure

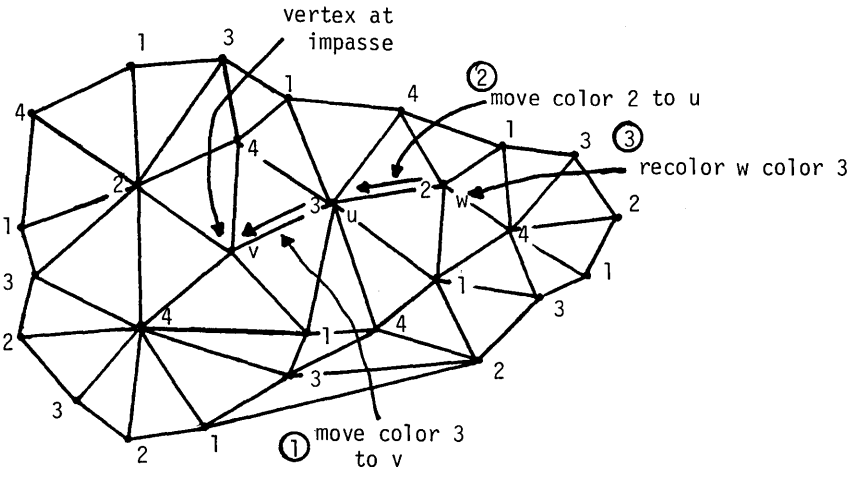


Figure 2. The Idea Behind Wandering Fifth Color

```
procedure wander(w: vertex, var success: boolean);
 procedure recursivewander(v: vertex);
  begin
   (* Try to resolve the impasse in the first neighborhood of v *)
   for each neighbor, u, of v whose color occurs only once in the first
        neighborhood of v do
     if coloring v the color of u and uncoloring u allows u to be recolored
          without creating a conflict
       then color v the color of u, recolor u, set success to true and
            return from wander;

   for each neighbor, u, of v whose color occurs only once in the first
        neighborhood and which has not been involved in the wandering
        process in the current call to wander do
     begin
       color v the color of u and uncolor u;
       recursivewander(u)
     end

   (* We are either trapped in a vertex of degree eight or higher in which
      every color is used twice in the first neighborhood, or have reached
      dead ends by attempting to cross the wandering path, or we have
      explored all the children and failed to resolve the impasse. In
      any event we backtrack.
   *)
  end; (* recursivewander *)
 begin
  recursivewander(w);
  (* If we return to this point we have failed to resolve the impasse *)
  success := false
 end; (* wander *)
```

Figure 3. Impasse Resolution by Wandering

- The vertices can be ordered randomly, or within an ordering scheme, randomness can be used to break ties, instead of relying on a secondary ordering strategy.
- Instead of assigning to a vertex the first available color of lowest index, we can assign it a random color from the colors that do not cause conflicts.

An exact analysis of the running time of the algorithm is impossible, because there are too many secondary interactions in the coloring process. It is possible to provide some experimental data and to make some simplifying assumptions that allow us to predict the running time. The results presented in Table 1 allow us to conclude, at least for graphs of size less than 15,000, that

- The number of vertices at impasse during a successful coloring is a linear function of the graph size. Phrased another way, the inter-impasse distance is constant and not a function of the size of the graph. This is not surprising since (for non-pathological cases) being at impasse is essentially a localized phenomenon.
- The number of steps in the wandering operation, whether it succeeds or fails, is constant on average. This was measured experimentally over a wide variety of graph sizes, ordering heuristics and color assignment strategies. The average is consistently in the 6–7 range. This differs significantly from the Kempe chain approach where the size of the Kempe chains searched increase as the graphs get larger.
- The probability that the wandering process fails to resolve the impasse is a local phenomenon. This is roughly equivalent to assuming that each call to wander is independent of previous actions taken in coloring the graph.

With these assumptions we see that the probability of coloring a graph without need to restart the coloring process is given by

$$(1-p)^{aN} = C^N$$

where p is the probability that the impasse will not be resolved and a is the constant of proportionality for the number of impasses. Since processing a non-impasse vertex and an impasse vertex both appear to take constant time, we get an expected running time proportional to

$$NC^N + 2NC^N(1-C^N) + 3NC^N(1-C^N)^2 + \cdots = N(1/C)^N$$

Though exponential, since $1/C > 1.0$, the value of $(1/C)^N$ is inconsequential over a wide range of N because we estimate $1/C$ to be in the vicinity of 1.00016. This would predict a ratio of running times between graphs of 15,000 vertices and graphs of 5000 vertices of about 15. The observed ratio is closer to 10.

EXPERIMENTAL RESULTS

In this section we give results for various combinations of parameters to the algorithm. We experimented with three static ordering strategies, the classical largest first and smallest last (randomly ordering the vertices within each grouping so as to have randomness in the event of restarts) and "rose petal" ordering. This last is a static ordering created by selecting a vertex for the center and then doing a breadth first search of the graph. The central vertex is chosen randomly (without repeats) so that a different ordering results on each restart, at least until all choices have been exhausted. Largest first did very poorly, actually being unable to four-color some graphs of size N in N restarts for N less than 1000. Rose petal ordering produced the best results and suggested to us that the dynamic saturation ordering might be an appropriate choice. We modified the program to allow the static orderings to be overlaid with saturation ordering, with saturation coming into play at different levels. For example, a level of zero is equivalent to turning our attention immediately to vertices at impasse, instead of waiting for them to be processed according to the order dictated by the static ordering. Notice that when a vertex is first brought to impasse there is necessarily at least one escape route, but if we delay no escape routes might exist when it becomes time to process the vertex and a restart will be necessary. Table 2 presents the results for a number of different combinations and graph sizes for MPG5 graphs. For the smaller sizes fifty trials were run. For the larger sizes only ten trials were made. The complete code can be found in Archuleta[1]. The number of restarts is the main factor affecting running time, though how early within a linear pass through the graph the algorithm fails to resolve an impasse is also important. Largest first consistently performed the worst by a significant margin and is an inappropriate ordering for this algorithm. Smallest last performs best with a level of saturation of one (there is only one possibility left for a vertex, so color it that color), whereas rose petal ordering did better when the saturation level was two. A saturation level of three is equivalent to the saturation ordering without any secondary ordering heuristic. This ordering was observed to perform somewhat erratically, though the small sample size for the larger graph sizes makes drawing firm conclusions unjustified. Because the algorithm is so fast the time to maintain a precise ordering based on secondary ordering heuristics negates any advantage that might be gained.

We also experimented with different strategies for assigning an available color when more than one choice existed. We tried three approaches to assigning colors:

- Skewing the assignment to favor the lowest numbered color classes. This is the classical method of assigning the colors in the greedy method.
- Always giving out the color so as to equalize the number of colors in each color class.
- Assigning a color at random from the available colors. This method has the advantage that it inserts additional randomness into the algorithm.

We found that skewing the colors according to the classical scheme gave the best results. This is the color assignment method used in the tabular results presented in Table 2. We have not explored the reason why a skewed distribution outperforms a balanced one; it may be that the smaller size of color class four makes impasse resolution slightly more likely.

Table 2. Running Times (Seconds)/Restarts for the Wandering Fifth Color Algorithm

	Method			
Approximate Size	Rose Petal Saturation = 2	Smallest Last Saturation = 1	Pure Saturation (No secondary ordering)	Largest First Saturation = 1
15000	73.31/6.80	125.16/6.00	177.93/19.60	
12500	84.43/9.50	84.83/4.90	35.19/2.60	
10000	33.65/3.20	44.86/2.50	38.59/4.30	
7500	21.70/2.30	37.19/2.70	19.17/2.00	
5000	12.90/1.60	13.61/0.84	15.36/2.14	66.82/16.70
2500	4.42/0.44	5.95/0.46	4.01/0.46	10.11/2.94
1000	1.52/0.20	2.14/0.12	1.52/0.18	2.21/0.70

FURTHER RESEARCH

We close with some remarks regarding future research. The primary reason that the running time of the algorithm is not linear is that when we cannot resolve an impasse we completely discard the current coloration. For sizes that we have considered this does not occur often enough for it to be of serious concern. One approach to preventing restarts is to switch to a more sophisticated impasse resolution algorithm when wandering fails, for example the one used by Morgenstern. Because of the long running times he reports, using sequences of Kempe interchanges will be profitable only if the expected number of restarts becomes very high. Another approach, based on the intuition that an impasse is a local phenomenon, is to excise a small portion of graph centered around the impasse vertex, say all vertices within distance three or four, and to consider all possible four-colorations, looking for one that agrees with the remainder of the graph on the boundary of the excised region. Because the subproblem we will create in this manner is small, an exhaustive backtracking algorithm can be used.

Table 3. Sizes of Color Classes Using Rose Petal Ordering with Saturation Level = 2

	Color Assignment Method		
Approx. Size	Skewed Distribution	Balanced Distribution	Random Assignment
15000	4005/3893/3752/3382	3759/3758/3757/3759	3763/3759/3750/3760
10000	2727/2666/2563/2315	2569/2568/2568/2568	2564/2571/2572/2565
5000	1374/1335/1275/1166	1288/1288/1288/1286	1289/1290/1285/1286
1000	279/272/262/237	264/263/262/261	263/261/264/262

References

[1] Archuleta, R.A. and H.D. Shapiro, *A Fast Probabilistic Algorithm for Four-Coloring Large Planar Graphs*, Department of Computer Science, The University of New Mexico, Tech Report No. CS86-2.

[2] Brélez, D., *New Methods to Color the Vertices of a Graph*, **CACM**, Vol. 22, pp. 251-256, (1979).

[3] Kittell, I., *A Group of Operations on a Partially Colored Map*, **Bull. Amer. Math. Soc.**, Vol. 41, pp. 407-413, (1935).

[4] Matula, D., G. Marble and J. Isaacson, *Graph Coloring Algorithms*, in **Graph Theory and Computing**, Academic Press, New York, pp. 109-122, (1972).

[5] Morgenstern, C.A. and H.D. Shapiro, *Performance of Approximation Coloring Algorithms on Maximally Planar Graphs*, Department of Computer Science, The University of New Mexico, Tech Report No. CS84-7.

[6] Morgenstern, C.A., Private Communication.

[7] Saaty, T.L. and P.C. Kainen, **The Four-Color Problem**, McGraw-Hill, New York, (1977).

[8] Williams, M.H. and K.T. Milne, *The Performance of Algorithms for Colouring Planar Graphs*, **The Computer Journal**, Vol. 27, pp. 165-170, (1984).

[9] Williams, M.R., *Heuristic Procedures (If They Work — Leave Them Alone)*, **Software — Practice and Experience**, Vol. 4, pp. 237-240, (1974).

Techniques for Collision Resolution in Hash Tables with Open Addressing[*]

J. Ian Munro
Pedro Celis
Data Structuring Group, Department of Computer Science,
University of Waterloo, Waterloo, Ontario, N2L 3G1

Abstract

In this paper we focus on the problem of resolving collision in hash tables through open addressing. A number of techniques, both old and new, are surveyed. The results of analyses and extensive simulations are presented.

1 Introduction

Hashing was one of the first techniques proposed for implementing the operations of insert, delete and find in a data structure. Knuth [20] gives credit to Luhn for originating the idea of hashing in an internal IBM memorandum of January 1953.

Ideally, we would like a hashing scheme to determine solely from the identification of a record, called the *record key*, the exact location in which the record is stored. Given a record to be inserted or located, a *key to address transformation* is performed using a *hash function* $h(k) : \mathcal{K} \mapsto \{0, \ldots, m-1\}$ which takes as an argument a key k in the specified universe $\mathcal{K}$ and returns an integer $h(k)$ between 0 and $m-1$, where m is the size of the table. The record is then inserted in the table entry specified by $h(k)$. This causes no problems until a record with key k' has to be inserted and location $h(k')$ is already occupied. In this case we say a *collision* has occurred.

If, as is usually the case, the hash function essentially maps keys to locations at random, then collisions are almost certain to occur even if the table is sparsely populated. The famous "birthday paradox" (see for example [10]) asserts that among 23 or more people the probability that at least 2 of them share the same birthday exceeds 1/2. In other words, if we select a random function that maps 23 records into a table of size 365, the probability that no two keys map into the same location is only 0.4927. In general, a hash table of size m is likely (probability $> \frac{1}{2}$) to have at least one collision by the time it contains about $\sqrt{\pi m}$ elements.

In order to use a hashing scheme, two almost independent decisions must be made; a hash function must be selected as well as method for handling collisions. There are two popular ways of handling collisions, *chaining* and *open addressing.* The idea of chaining is to keep, for each location, a linked list of the records that *hash* to that location. This implies that each entry in the table must have enough space to contain a record and a link field. There are a number of interesting tradeoffs and techniques in connection with chaining. Our interest, however, lies in an approach

[*]This work was supported by the Natural Science and Engineering Research Council of Canada under Grant A8237

which calls for no additional storage, namely open addressing.

2 Open Addressing

Peterson seems to have introduced the notion of collision resolution by open addressing in his seminal paper of 1957 [27]. The idea is to do away with the links entirely, and to insert by probing the table in a systematic way. When a collision occurs, one of the colliding records is selected to keep the table location, while the other one continues probing until inserted. The sequence of table entries to be inspected when inserting or searching for a record is called the *probe sequence*. We can augment the hash function with another parameter, the *probe position* or try number, and use it to generate the probe sequence for a record. Thus the hash function becomes $h(k,i)\colon \mathcal{K} \times \{1,\ldots,\infty\} \mapsto \{0,\ldots,m-1\}$.

Schemes for inserting into a hash table with open addressing can vary in two ways: in how the probe sequence is generated, and in how to decide which of the colliding records is to be rehashed. We call the latter portion of the insertion algorithm the *reordering scheme*. This decision could be based on any knowledge of the record key values, their current probe positions, their future probe sequences and/or the order in which the records were inserted.

In discussing the various hashing schemes we will be interested in comparing the following three performance measures:

— I average number of probes required to insert a record;

— S average number of probes required to find a record in the table;

— U average number of probes required to determine that no record in the table has a given key value.

To determine the value of these performance measures we will often find convenient to study the following random variables:

— the probe sequence length for a key (**psl**),

— the longest probe sequence length (**lpsl**), which is the largest value of **psl** among all the records in the table;

We will compare the expected value (denoted by E[•]), and sometimes the variance (denoted by V[•]), of these random variables for both the case of full and nonfull tables. For large nonfull tables such expressions may be functions of α, the load factor, defined as $\alpha = n/m$, where n is the number of records in the table and m is its size. Several analyses of hashing schemes have been performed for infinite nonfull tables with load factor α, where $\alpha \leq 1-\epsilon$, $\epsilon > 0$. Throughout the paper we refer to these tables as α-full tables.

A very useful but often neglected performance measure of a hash table is the longest probe sequence length (**lpsl**). This metric provides a bound on the cost of both successful and unsuccessful searches. If all records in the table are stored in probe positions between 1 and **lpsl**, then **lpsl** probes into the table will be sufficient to find any record, or to determine that no record with such key exists. This elegant, but sadly underutilized, idea is due to Lyon [23].

The simplest open addressing hashing scheme is known as *linear probing*. It uses the hash function $h(k,i) = (h_1(k) + (i-1) * c) \bmod m$, where $h_1(k)$ is the initial hash function and c is a constant relatively prime to m. This method was discovered independently by Ershov [8] and Peterson [27]. Experience with linear probing shows that the algorithm works quite well until the table is relatively full. The problem with linear probing is that it uses only one circular path to resolve collisions and suffers from a piling-up phenomenon called primary clustering. If r consecutive (mod c) locations are occupied and a new value hashes into any of these spots (or the one before or after this segment) then at least $r+1$ consecutive locations will be occupied. Furthermore, to find this additional key will take about $r/2$ probes on average. This unfortunate event occurs with probability proportional to r, exacerbating the situation.

The performance of linear probing improves significantly if we allow c to be a function of k instead of a constant [1]. This gives rise to a new scheme, called *double hashing*, that uses two independent auxiliary hash functions $h_1(k)$ and $h_2(k)$ to compute $h(k,i) = (h_1(k) + (i-1) * h_2(k)) \bmod m$. The values $h(k,i), i = 1, \ldots, m$ will yield a permutation of the table locations provided $h_2(k)$ is prime relative to m. A simple, and indeed standard, way to guarantee this is to insist that the table size, m, be a prime. Following this simple requirement it is found that double hashing performs much better than linear probing for high load factors as it essentially eliminates the possibility of two colliding records having the same remaining probe sequence.

Two idealizations of open addressing schemes are frequently mentioned in the literature and used as models for analysis. These are *uniform hashing*, under which the hash function provides a random permutation of the numbers $\{0, \ldots, m-1\}$; and *random probing*, in which $h(k,i)$ is simply a number chosen at random from $\{0, \ldots, m-1\}$. The difference between these two schemes is that random probing is memoryless, meaning that a location may be probed several times before some other location is probed for the first time. Random probing is simpler to analyze and has essentially the same performance as uniform hashing for α-full tables under most reordering schemes. Random probing and uniform hashing are not usually implemented, since empirical evidence shows that their performance is close to that of double hashing which is much more reasonable to implement. Guibas [17] has proven that the behavior of double hashing is asymptotically equivalent to uniform hashing for load factors α not exceeding a certain constant $\alpha_0 = .31....$ The difficulty of the proof and the bound on the load factor are indications of the difficulty in other analyses of double hashing. The interest of random probing and uniform hashing lies in the fact that they are simpler to analyze and appear to approximate closely the performance of double hashing.

3 Reordering Schemes

The standard insertion algorithm is to inspect the probe sequence corresponding to the record to be inserted until an empty table location is found, and place the new record there. Hence the reordering scheme consists of giving preference to the record that was inserted first.

The standard search algorithm is to inspect locations in the order of the probe sequence of the key value. Under this scheme, a search can be stopped as unsuccessful when either an empty table location is probed or lpsl probes have been made. Furthermore, $S = \text{E[psl]}$ and $U \leq \text{E[lpsl]}$, regardless of the reordering scheme.

For linear probing and a nonfull tables it can be shown that [20,31]

$$\begin{aligned}\text{E[psl]} &= \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) - \frac{1}{2(1-\alpha)^3 m} \\ &\quad + O\left(m^{-2}\right) \\ \text{V[psl]} &= \frac{\alpha(\alpha^2 - 3\alpha + 6)}{12(1-\alpha)^3} - \frac{3\alpha+1}{2(1-\alpha)^5 m} \\ &\quad + O\left(m^{-2}\right)\end{aligned}$$

and for full tables

$$\text{E[psl]} = \sqrt{\pi n/8} + \frac{1}{3} + O\left(n^{-1/2}\right).$$

No closed form results for lpsl have been presented for this method, but Larson [21] gives some numerical results for an approximate analysis. According to his computations, for $m = 10^3$ and $\alpha = 0.6$, E[lpsl] = 23.6. If the load factor is increased to 0.9 this number goes to 288, and if instead of increasing the load factor we increase the table size to 10^6, the number goes up to 67.1.

Peterson proves that, for linear probing with the standard insertion algorithm, the average probe position in a given table in which a record is stored is independent of the order in which the records are inserted. Basically the same proof can be used to show that the average probe position is independent of the reordering scheme used. Hence, regardless of the reordering scheme used, $I = \mathrm{E}[\mathbf{psl}]$. Since the standard search algorithm is used, $S = \mathrm{E}[\mathbf{psl}]$ for the value of E[psl] presented above. Since no analysis for lpsl has been presented, the value of U has not been determined. However, ignoring Lyon's modification, we have that for nonfull tables

$$U = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) - \frac{3\alpha}{2(1-\alpha)^4 m} + O\left(m^{-2}\right)$$

and if the table is full, $U = n$.

For the standard insertion algorithm with uniform hashing the following equations can be established [27,13,14]: for a nonfull table

$$\mathrm{E}[\mathbf{psl}] = \frac{m+1}{n}\left[H_{m+1} - H_{m-n+1}\right] \approx -\alpha^{-1}\ln(1-\alpha)$$

$$\mathrm{V}[\mathbf{psl}] \approx \frac{2}{1-\alpha} + \alpha^{-1}\ln(1-\alpha) - \alpha^{-2}\ln^2(1-\alpha)$$

$$\mathrm{E}[\mathbf{lpsl}] = -\log_\alpha m - \log_\alpha(-\log_\alpha m) + O(1)$$

and for a full table

$$\mathrm{E}[\mathbf{psl}] = \ln n + \gamma - 1 + o(1)$$

$$\mathrm{E}[\mathbf{lpsl}] = 0.6315... \times n + O(1)$$

where $\gamma = 0.5772156649...$ is Euler's constant. If we also use the standard search algorithm then psl represents the cost of both insertions and successful searches, and lpsl represents a bound on the cost of unsuccessful searches. Hence $I = S = \mathrm{E}[\mathbf{psl}]$ and $U \leq \mathrm{E}[\mathbf{lpsl}]$. We see that uniform hashing, and so presumably double hashing, becomes substantially better than linear probing for relatively full tables. The remainder of this paper will focus on the fact that, unlike linear probing, reordering schemes can be used to advantage under these probing methods.

3.1 Brent's Method

Brent [3] was the first to propose moving stored records to reduce the expected value of the probe sequence length. During an insertion, a sequence of occupied table entries is probed until an empty location is found. Brent's scheme checks to see whether any of the records in these occupied locations can be displaced so that the cost of searching for the new value plus the extra cost of searching for the displaced element is smaller. More formally, let d_i be the number of locations the key in the i-th location probed by the new record has to move before finding an empty location. Then Brent's method selects the value of i that minimizes $i + d_i$, places the new record on its i-th choice and moves the record previously in that location d_i steps ahead. Figure 1 shows graphically how one such insertion of a record R might occur. In the example, instead of inserting the record R in its fifth choice and increasing the total table cost by 5, record R_3 is displaced to its next choice, and then R is placed in its third choice, the place formerly occupied by R_3. The increase in the sum of the probe sequences is thus reduced from 5 to 4.

No analysis of the cost of loading a table has been presented for this method, but simulations presented in [4] support the conjecture that $I = \Theta(1)$ for α-full tables and $I = \Theta(\ln n)$ for full tables. Since the stan-

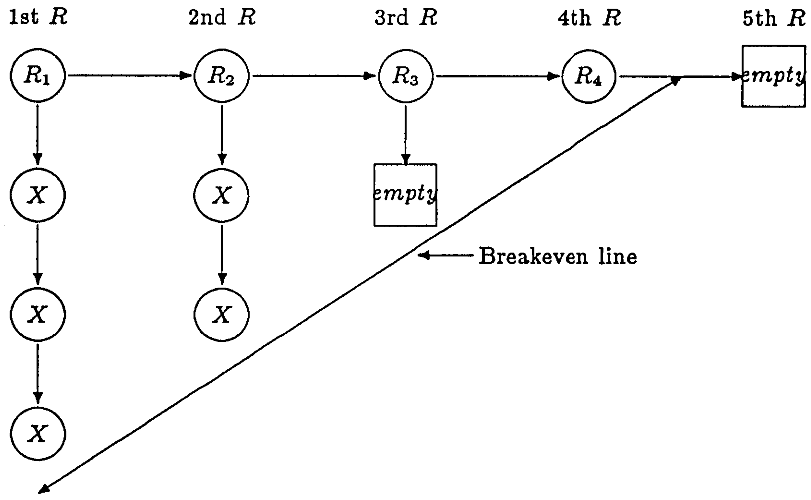


Figure 1: Sample insertion in Brent's method

dard search algorithm is used then S = E[psl] and $U \leq$ E[lpsl].

The tables produced by Brent's scheme have a very good E[psl], even when completely filled. For random probing and α-full tables the expected values are

$$\mathrm{E[psl]} = 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} - \frac{\alpha^5}{18} + \frac{2\alpha^6}{15} + \frac{9\alpha^7}{80} - \frac{293\alpha^8}{5670} - \frac{319\alpha^9}{5600} + \cdots$$

taking the limit numerically as $\alpha \to 1$ indicates that for full tables

$$\mathrm{E[psl]} \approx 2.4941...$$

This agrees with simulations using double hashing. So, if the standard search algorithm is used, a record can be retrieved in less than 2.5 probes, on average, regardless of the table size. There has been no successful analysis of the expected value for lpsl but, based on simulations and an intuitive argument, it is conjectured [16] to be $\Theta(\sqrt{n})$ for full tables, and $\Theta(\ln n)$ for nonfull tables [4].

Brent's method does not require any extra memory to perform an insertion. If the search for an empty location is done on a depth first basis, as suggested in [3], the expected number of times the $h_2(\bullet)$ hash function will be computed is $\alpha^2 + \alpha^5 + \alpha^6/3 + \cdots$ eventually approaching $\Theta(\sqrt{n})$ for full tables [20]. The disadvantage of searching in this manner is that a number of locations below the breakeven line will be probed. For example the fifth probe position of the record R to be inserted would be probed unnecessarily. The number of additional table positions probed during an insertion is approximately $\alpha^2 + \alpha^4 + \frac{4}{3}\alpha^5 + \alpha^6 + \cdots$ [20].

Another way of searching for the closest empty location is to do a level search as recommended in [14]. This causes an essentially minimal number of locations to be inspected. However, for double hashing, this implies $h_2(\bullet)$ be called up to $\Theta(n)$ times instead of $\Theta(\sqrt{n})$. A further disadvantage of this insertion approach is that duplicate keys are not detected by the insertion algorithm, so if duplicate insertion requests may occur, an unsuccessful search should precede each insertion.

If we are to use Lyon's trick to truncate unsuccessful searches, we must of course update the **lpsl** when insertions are made. This requires a minor twist that is easily incorporated into either of the insertion schemes. **lpsl** may be updated either because the new element is in a position late in its probe sequence, or because the element displaced is moved to a late spot. The former is detected as part of the insertion, the latter requires an extra search.

3.2 Binary Tree Hashing

Binary tree hashing is the natural generalization of Brent's method. Not only is the record being inserted allowed to displace other records in its probe sequence, but these displaced records may further displace other records in their probe sequences. This is illustrated graphically by Figure 2. In the figure, R's first choice is occupied by R_1, hence the next probe position of both R and R_1 are checked. These two locations are also occupied by R_2 and R_3 respectively, hence the next probe position of all four records is inspected (note that the four records are not necessarily distinct). At that point it is found that the next probe position for R_1 is empty, then R_1 will be advanced two positions in its probe sequence and R will be placed in its first choice, the location previously occupied by R_1. This method was discovered independently by Mallach [25] and by Gonnet and Munro [16].

Since this method is a generalization of Brent's, it is expected to produce better tables at a somewhat higher cost. An approximate model [16] yields the following for random probing and α-full tables[1]:

$$\mathrm{E}[\mathbf{psl}] = 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} - \frac{\alpha^5}{18} + \frac{2\alpha^6}{105} + \frac{83\alpha^7}{720} + \frac{613\alpha^8}{5760} - \frac{69\alpha^9}{1120} + \cdots$$

taking the limit numerically as $\alpha \to 1$ indicates that for full tables

$$\mathrm{E}[\mathbf{psl}] \approx 2.13414\ldots$$

This agrees with simulations using double hashing. There is no analysis for the expected value for **lpsl** but Gonnet and Munro, based on simulation results, conjectured them to be about $\lg n + 1 \approx 1.44 \ln n + 1$ for full tables.

Again the cost of searching is given by the formulas $S = \mathrm{E}[\mathbf{psl}]$ and $U \leq \mathrm{E}[\mathbf{lpsl}]$, since the standard search algorithm is used. As with Brent's scheme, the expected cost of inserting a record has not been successfully analyzed for either of the insertion algorithms described below.

The natural order for inspecting the table when searching for an empty location is by levels, as suggested in [25] and [24][2]. However, the amount of memory required to store the tree of locations probed is large, as Mallach noted. Simulations [6] give credence to the hypothesis that the expected value of I for a full table is about $.5n^{1/2} \ln n$ and the amount of memory required to store the tree about $.15n^{3/2} \ln n$. What is worse, the variance of these two measures is very high, so the probability of requiring, say, n^2 extra memory locations is not insignificant.

Gonnet and Munro [16] show how to use an algorithm for the transportation problem by Edmonds and Karp [9], to insert keys into the table. An enhancement of the algorithm has a worst case runtime of $\Theta(n^2 \ln n)$ [11] which suggests that I could be as high as $\Theta(n \ln n)$. However, simulations [6] suggest that the expected value for I for a full table is $\Theta(n^{1/2} \ln n)$, and the amount of extra memory required is $\Theta(n)$, each with a small variance. While this is a great improvement over the natural algorithm, it is still expensive both in time and memory.

As with Brent's method, a small number of additional probes are required to avoid duplicate keys and

[1] The formula for E[psl] for α-full tables is taken from [14] and differs slightly from the one in [16].

[2] The reader is warned that the analysis presented in [24] is not entirely accurate.

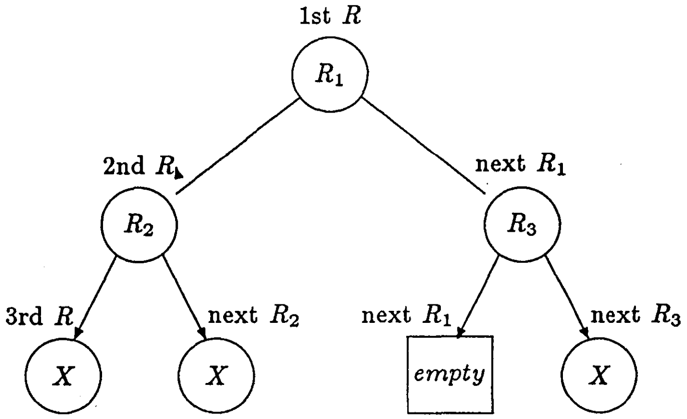


Figure 2: Sample insertion using Binary Tree Hashing

to keep track of the value of lpsl.

3.3 Optimal and Min-Max Hashing

Both Brent's method and binary tree hashing move stored keys forward in their probe sequences. Further reductions in the value for E[psl] could be obtained if records could also be moved back in their probe sequences. Lyon [23] presents simulations results on a different generalization of Brent's method which allows records to be moved back a limited number of steps. Poblete [28] has simulated a generalization of binary tree hashing where records are allowed to move back a limited number of steps. All of these methods present tradeoffs between the cost of insertion (I) and the cost of searching (S and U).

It is natural to ask what reordering scheme minimizes the value of E[psl], or, in other words, minimizes S for the standard search algorithm. Such a scheme is called optimal hashing [29,30,16]. Poonan [29] was the first to note that the optimal placement of keys in a hash table is a special case of the assignment problem [19], which can be solved in $O(n^2 \log n)$ time in the worst case [11]. Neither the expected cost of finding the optimal hash table nor the expected values of psl and lpsl have been determined. For E[psl] and full tables the following bounds exist [7,16]:

$$1.7 < \mathrm{E[psl]} < 2.135$$

Simulations [16,6] indicate that for full tables $\mathrm{E[psl]} \approx 1.82...$, $\mathrm{E[lpsl]} = \Theta(\ln n)$, and $I = \Theta(\sqrt{n})$.

Instead of determining the reordering scheme in the insertion algorithm that will minimize S for the standard search algorithm, we might be interested in minimizing U. This min-max approach has the feature that in addition to minimizing lpsl for the specific table (and hence its expected value), simulations indicate the average psl can be kept to about $1.83\ldots$, that is, very close to the cost of the optimal table. The expected value of lpsl is bounded by [13,4]

$$\ln n + \gamma + \frac{1}{2} + P(\ln n) + o(1) < \mathrm{E[lpsl]}$$
$$< 3\ln n + \lceil \lg(n-2) \rceil + 3\gamma + o(1)$$

where $P(x)$ is a periodic function with period 1 and magnitude $|P(x)| \leq .0001035$ for full tables, and by

$$-\alpha^{-1}\ln(1-\alpha) < \mathrm{E[lpsl]}$$
$$< -3\alpha^{-1}\ln(1-\alpha) + \lceil \lg(m-2) \rceil$$

for α-full tables.

Unfortunately, the approaches used in creating both optimal and min-max hash tables are time consuming and can require $\Theta(n)$ extra memory to process an insertion.

In summary, we can say that binary tree, optimal and min-max hashing reduce the expected values of psl and lpsl dramatically, but at a high cost for table creation. The expected number of operations to construct a table using one of these algorithms is high compared to the standard hashing scheme. They also require a nontrivial amount of extra memory during the creation phase. These methods are best suited for applications in which the set of keys is static and known in advance. In such cases, the cost of constructing the table can be amortized over a large number of search operations, and the additional memory space required can be released as soon as the table has been created.

3.4 Robin Hood Hashing

Robin Hood hashing [5,4] is a reordering scheme for the insertion algorithm that differs from the previous methods in that it does not attempt to reduce the expected value of psl. Instead it tries to reduce the variance of psl, and as a consequence the expected value of lpsl, without significantly increasing the cost of insertions.

The idea is very simple: resolve a collision by giving the location to the record that is further along its probe sequence. The name of the scheme comes from the fact that we take from the rich to give to the poor. For random probing and full tables we have the following

$$E[\text{psl}] = \ln n + \gamma + o(1)$$

$$E[\text{lpsl}] < 3\ln n + \lceil \lg(n-2) \rceil + 3\gamma + o(1)$$

and taking a limit numerically as $\alpha \rightarrow 1$ indicates that for full tables

$$V[\text{psl}] \approx 1.883$$

Hence if the standard search algorithm is used, both S and U will be $\Theta(\ln n)$ for full tables. The cost of inserting will be affected by the cost of determining the probe position of a stored record. If this is determined by doing a "search" for the stored record then $I = \Theta(\ln^2 n)$.

However, better search algorithms have been proposed. These algorithms exploit the fact that the variance of psl is so small, meaning that most records are stored in a probe position very close to the average. The idea is to change the order in which the candidate probe positions are inspected during a search, starting near the average probe position and moving away from the average in both directions. In other words, we start by inspecting first those positions that have a high probability of success. The unsuccessful search (U) will remain at lpsl since all candidate probe positions must be inspected (in any order) before the search can be stopped. The average cost of searching, however, is reduced to $S < 2.6$. We can use the same idea when "searching" for a stored record to determine its probe position to reduce the cost of insertion to $I = \Theta(\ln n)$.

4 Updates

Deletion of unused items is troublesome in open addressing, because unoccupied table positions that had a collision must be marked as "deleted" so as to prevent fallacious, unsuccessful searching. There will then be three kinds of table entries: empty, occupied and deleted. When inserting, deleted table entries are treated as if they were empty, but they are treated as if they were occupied during searches.

For linear probing, deleted entries can be easily avoided by moving back a record that was rejected from the location. However, as we have noted, the performance of linear probing is inferior to the other methods and therefore performs poorly even if no updates have been made.

For random probing and the standard insertion and search algorithms (modified to handle deleted entries correctly), after a sufficiently large number of delete/insert pairs, the cost of searching will be given by $S = (1 - \alpha)^{-1}$. Since all locations in the table are now either occupied or deleted, and the value of the longest probe, **lpsl**, can only increase, unsuccessful searches take m probes. Hence the performance deteriorates to something worse than the performance of linear probing.

Gunjin and Goto [18] present a partial analysis for an algorithm that retraces the probe sequence of every record in the table and marks as empty all those deleted entries that are no longer jumped over by some other record. Their algorithm will improve the processing of unsuccessful searches only. If one is willing to use m additional bits of storage it is better to rehash the table in situ [2]. This will require less time, reduce even further U and reduce greatly S.

We feel the most encouraging result in the area of deletions in open in addressing is a modification to the Robin Hood algorithms [5,4]. The following modification has been studied: to delete a record, mark the table entry as deleted but keep the key value; when inserting, a deleted element is discarded if and only if it would be displaced if it were not flagged as deleted.

This modification causes the expected value of psl to increase without bound, but simulations indicate that the variance remains bounded by a small constant, and is never greater than that of a full table in which no deletions have occurred (i.e. $< 1.88\ldots$). The expected value will increase without bound because once a location contains a record at probe position i, then in the future it can only contain records that are at or past probe position i.

Employing the Robin Hood search schemes noted above, the cost of a successful search after an arbitrary number of insertions and deletions remains bounded by the cost of a successful search in a full table with no deletions. Unsuccessful searches are similarly bounded by those of a full table with no deletions.

Since the average probe position grows without bound, it would appear that the cost of doing an insertion also grows without bound, because each key has probed positions 1 to about the average probe position. This pitfall can be avoided by keeping track of the value of the smallest probe position among the records (deleted or otherwise) in the table. The insertion procedure then starts at a probe position equal to this value, since a placement before that position is not possible. Empty locations are treated as containing a deleted record in probe position 0.

The most efficient way of keeping track of the smallest probe position is to have counters of how many records are at each probe position. We expect only $\Theta(\ln n)$ counters to be needed as simulations indicate that about $1.15 \ln n + 2.5$ different probe positions will contain all the records in the table.

5 Conclusions

Hashing with open addressing does lead to very good tables particularly in the static case, even when the load factor approaches or reaches 1. Brent's scheme and Robin Hood hashing are reasonably fast insertion techniques. As we have noted Robin Hood seems to lead to good dynamic tables even when deletions are permitted. This may be surprising to many who, like us, felt that open addressing was inherently poor when a significant number of deletions were to be made. The analysis of Robin Hood with deletions is the most obvious open problem. We note, however, that a number of results we have quoted are experimental and some analyses are based on simplifying models. More complete analyses of many of the methods are yet undone.

Finally, there is the search for even better approaches to conflict resolution by open addressing.

References

[1] **de Balbine, G.** *Computational Analysis of the Random Components Induced by a Binary Equivalence Relation,* Ph. D. Thesis, Calif. Inst. of Technology, 1969

[2] **Bays, C.,** " The Relocation of Hash-Coded Tables", *Communications of the ACM,* Vol. 16, No. 1, May 1973

[3] **Brent, R.P.,** "Reducing the Retrieval Time of Scatter Storage Techniques", *Communications of the ACM,* Vol. 16, No. 2, pp.105-109, February 1973

[4] **Celis, P.** "Robin Hood Hashing", Ph. D. Thesis, University of Waterloo, January 1986.

[5] **Celis, P., P.-Å Larson and J.I. Munro,** "Robin Hood Hashing", *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science,* pp.281-288, October 1985

[6] **Celis, P. and J. I. Munro,** "A Simulation Study of Several Hash Schemes", In preparation

[7] **Celis, P. and J. I. Munro,** "An Improved Lower Bound for Optimal Hashing", In preparation

[8] **Ershov, A.P.,** *Doklady Akad. Nauk. SSSR,* Vol. 118, pp.427-430, 1958

[9] **Edmonds, J. and R.M. Karp,** "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", *Journal of the ACM,* Vol. 19, No. 2, pp.248-264, April 1972

[10] **Feller, W.,** *An Introduction to Probability Theory and its Applications, Vol. I,* John Wiley & Sons, New York, 1968

[11] **Fredman, M.L. and R.E. Tarjan,** "Fibonacci Heaps and Their Use in Improved Network Optimization Algorithms", *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science,* pp.338-346, October 1984

[12] **Gonnet, G.H.,** "Average Lower Bounds for Open-Addressing Hash Coding", *A Conference on Theoretical Computer Science,* pp.159-162, University of Waterloo Waterloo, Ontario, August 1977

[13] **Gonnet, G.H.,** "Expected Length of the Longest Probe Sequence in Hash Code Searching", *Journal of the ACM,* Vol. 28, No. 2, pp.289-304, April 1981

[14] **Gonnet, G.H.,** *Handbook of Algorithms and Data Structures,* Addison-Wesley, Reading Massachusetts 1984

[15] **Gonnet, G.H. and P.-Å. Larson,** "External Hashing with Limited Internal Storage", *Technical report CS-82-38,* Computer Science Dept., Univ. of Waterloo, October 1982

[16] **Gonnet, G.H. and J.I. Munro,** "Efficient Ordering of Hash Tables", *SIAM Journal on Computing,* Vol. 8, No. 3, pp.463-478, August 1979 (a preliminary version was presented at the 9th ACM STOC May 1977)

[17] **Guibas, L.J.,** "The Analysis of Hashing Techniques that Exhibit K-ary Clustering", *Journal of the ACM,* Vol. 25, No. 4, pp.544-555, October 1978

[18] **Gunji, T and E. Goto,** "Studies on Hashing PART-1: A Comparison of Hashing Algorithms with Key Deletion", *Journal of Information Processing,* Vol. 3, No. 1, 1980

[19] **Kōnig, D,** "Graphok és Matrixok", *Matematikai és Fizikai Lapok,* Vol. 38, pp.116-119, 1931

[20] **Knuth, D.E.,** *The Art of Computer Programming, Vol. III: Sorting and Searching,* Addison-Wesley, Reading Massachusetts, 1973

[21] **Larson, P.-Å.,** "Expected Worst-case Performance of Hash Files", *The Computer Journal.* Vol. 25, No. 3, 1982

[22] **Larson, P.-Å.,** "Analysis of Uniform Hashing", *Journal of the ACM,* Vol. 30, No. 4, pp.805-819, October 1983

[23] **Lyon, G.E.,** "Packed Scatter Tables", *Communications of the ACM,* Vol. 21, No. 10, pp.857-865, October 1978

[24] **Madison, J.A.T.,** "Fast Lookup in Hash Tables with Direct Rehashing", *The Computer Journal,* Vol. 23, No. 2, pp.188-189, May 1980

[25] **Mallach, E.G.,** "Scatter Storage Techniques: A Unifying Viewpoint and a Method for Reducing Retrieval Times", *The Computer Journal,* Vol. 20, No. 2, pp.137-140, May 1977

[26] **Maurer, W.D. and T.E. Lewis,** "Hash Table Methods", *ACM Computing Surveys,* Vol. 7, No. 1, pp.5-19, March 1975

[27] **Peterson, W. W.,** "Addressing for Random-Access Storage", *IBM Journal of Research and Development* Vol. 1, No. 2, pp.130-146, April 1957

[28] **Poblete, P. V.,** *Studies on Hash Coding with Open Addressing,* M. Math Essay, University of Waterloo, Aug. 1977

[29] **Poonan, G.,** "Optimal Placement of Entries in Hash Tables", *ACM Computer Science Conference (Abstract only),* Vol. 25, 1976, (Also DEC Internal Tech. Rept. LRD-1, Digital Equipment Corp., Maynard Mass)

[30] **Rivest, R.L.,** "Optimal Arrangements of Keys in a Hash Table", *Journal of the ACM,* Vol. 25, No. 2, pp.200-209, April 1978

[31] **Schay, G. and W. G. Spruth** "Analysis of a file addressing method" *Communications of the ACM* Vol. 5, No. 8, pp.459-462 August 1962

PERFORMANCE ANALYSIS OF CONCURRENT MAINTENANCE POLICIES FOR SERVERS IN A DISTRIBUTED ENVIRONMENT

by

Farokh Bastani, Wael Hilal, and Ing-Ray Chen

Department of Computer Science
University of Houston - University Park
Houston, Texas 77004

ABSTRACT

One way of improving the performance of a server in a local area network environment is to assign the task of maintaining its data structures to separate maintenance processes. It is frequently suggested that the maintenance processes can be treated as low priority processes. However, our performance analysis shows that this policy can result in unstable systems. That is, eventually the response time for client requests becomes infinite. We propose and analyze several deterministic and stochastic scheduling strategies which ensure that the system is always stable. We also discuss the design of a directory server which incorporates a concurrent maintenance process. We present the experimental measurement of its performance and compare it with an alternative technique using a self-reorganizing data structure.

Index Terms: Degradable systems, maintenance processes, performance analysis, process scheduling strategies, self-reorganization.

I. INTRODUCTION

Many computing environments can be modeled as a set of servers and a set of clients. A typical example of this structure is local area network servers in which each server accepts requests for some operations from the clients and sends back responses. The server can be designed using two communicating processes, namely, a foreground process that interfaces with clients and serves their requests, and a background process that maintains the data-structures. The response time of the foreground process, and hence the response time for serving requests, is improved by shifting the time-consuming task of maintaining the data structures to the background process. For efficiency purposes, the background process updates the data-structure after a group of requests have been served and not after every request. If the primary data-structure is a binary tree, for example, the foreground process can simply mark a tree node as "deleted" in response to a delete request without actually deleting the node leaving the job of the actual deletion to the background process. For a request to add a new node, the foreground process can add this node on a simple auxiliary data-structure leaving the actual insertion of the new node to the background process. As more and more requests are served, the performance of the foreground process deteriorates gradually until the background job performs an "upgrade" to the primary data-structure. Systems whose response time deteriorates as additional requests are served are said to be "degradable". Their response time improves after the upgrade is performed. It is often suggested that the foreground process should be given a higher priority than the background process(es). However, our performance analysis shows that such a policy would result in unstable systems (i.e., systems with infinite queues and waiting times in their steady states).

The approach with foreground/background processes was originally suggested by Dijkstra et al [DIJ 78] in the specific context of concurrent garbage collection. Lampson [LAM 84] and Manber [MAN 84] have proposed this approach for more general applications. An example of an operational system is the concurrent garbage collector for the file server in the Cambridge Ring Network [NEE 82].

Two important issues are of interest in studying programs used in such systems. The first issue is a framework for systematically developing these programs [DIJ 78, BEN 84, JONES 83, BAS 85(A)]. The second important issue is the evaluation of the performance of such systems which is the subject of this paper. One method of performance analysis of such systems is to model the client, foreground and background processes with their average rates of generating requests, deteriorating the data structure (e.g., by creating garbage), and upgrading the data structure (e.g., by removing garbage), respectively [HIC 84]. However, the assumption of uniform flows can mask the effect of the variance of the actual distributions on the behavior of degradable systems. This is evident to some extent in the example of the random activity of termites given by Courtois [COU 85] in which a random process develops pockets of concentrated activities over time. Methods used in analyzing hardware performability problems [FUR 84] (e.g., the loss of a processor in a multiprocessor system) can also be

This research was supported in part by the National Science Foundation under Grant MCS-83-01745. The experiment was performed using equipment donated to the department by AT&T.

used for analyzing software systems. The difference is that in degradable software systems the reliability (correctness) of the data structure is not affected.

In this paper, we present queueing models and performance results for a multiple client degradable server as opposed to the single client model analyzed in [BAS 85(B), MOI 85, YEN 85]. Section II describes an abstract model for multiple client degradable servers. It also contains performance studies of degradable servers using both queueing theory and simulation. Several strategies for scheduling the background process are also discussed. The average waiting time was taken as our performance measure. Section III includes an operational example of a directory server incorporating concurrent foreground and background processes as an example of such systems. It also discusses the experimental evaluation of the directory server as implemented in a 3BNET network [AT&T 84(A)] of 3B2 computers [AT&T 84(B)]. Its performance was compared with that of an implementation incorporating a self-reorganizing data structure without having any background processes. The experimental results indicate that there are cases where self-reorganization has better performance than implementations using separate maintenance processes. Additional experiments have been carried out, though we were not be able to report them in this paper. These indicate that concurrent maintenance has better worst case performance while self-reorganization has better average case performance. Finally, section IV summarizes the paper and discusses some research directions.

II. MODELING AND PERFORMANCE ANALYSIS

The server is modeled by two processes, namely, a foreground process that interfaces with clients and serves their requests, and a background process that maintains the data-structures. The average waiting time for the foreground requests is an important performance measure and is used in our study to evaluate different maintenance policies. In this section, we show that the background process should not be restricted to run only during idle periods. Instead, it needs to have a well-defined scheduling policy that is tested for stability. This policy can be either deterministic or stochastic. An example of a deterministic policy is maintaining the data-structures after satisfying a predetermined number of requests. An example of a stochastic policy is one that probabilistically adjusts the relative priorities of the foreground and background processes.

As more and more housekeeping tasks accumulate, the performance of the foreground process deteriorates. Let the degradation level, d, represent the amount of accumulated housekeeping tasks. Then, the mean service time of the foreground process increases as d increases. This varying service distribution complicates the analysis of the performance of such systems. In this section, we first discuss an approximate queueing model analysis and demonstrate that such systems are basically unstable, i.e., the mean service time approaches infinity with probability 1. Then we investigate the performance of several deterministic and stochastic policies for scheduling the background process using queueing theory and simulation.

2.1. QUEUEING MODEL: In order to illustrate the analysis technique, we make the following simplifying assumptions:

(1) The data structure is a binary search tree. The permissible operations are to create a new node, to delete an existing node, and to inspect or update an existing node.

(2) The arrival of requests from the clients is a Poisson process with average arrival rate of λ. Consequently, the client population is assumed to be large.

(3) The distribution of the service time of the foreground process is exponential with mean $1/M_f(d) = \{1+\log(N+d)\}/\mu_f$, where μ_f is a constant, N represents the number of nondeleted nodes in the tree (assume that N > 0), and d represents the number of deleted nodes in the tree. This expression for $M_f(d)$ is justified since the average search time for a binary tree is proportional to the logarithm of the number of nodes in the tree.

(4) The distribution of the service time of the background process is exponential with mean $1/\mu_b$. This is optimistic to some extent since the background process also slows down as d increases.

Consider the state transition diagram shown in Figure 1. s_0 represents the state in which the data structure is in the ideal state, namely, immediately after an upgrade. In state s_{ij}, $j \leq i$, j-1 requests have been serviced, the jth request is being serviced and there are i-j pending requests. In state $s_{i,i+1}$, the foreground process is idle and the background process is active.

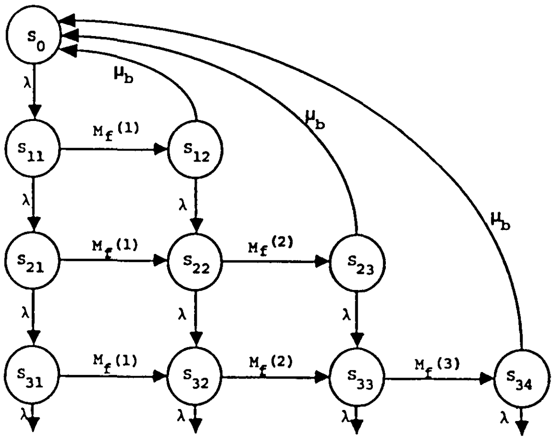


Figure 1. State Transition Diagram for a Simplified Model of Foreground/Background Processes.

Even with these assumptions, the analysis is very complex because $M_f(d)$ is not constant. A simple model which has a <u>smaller</u> average service time than the above is shown in Figure 2. The average service time is smaller for two reasons (1) in Figure 2 it is assumed that the background process instantaneously upgrades the data structure, and (2) during a period with N requests, the first model has average service time equal to $\sum_{i=1}^{N} 1/M_f(i)/N$, while this model has this value as its <u>maximum</u> possible service time.

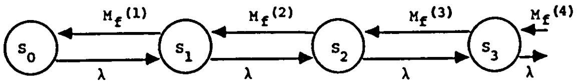


Figure 2. A System having a Smaller Average Service Time than the System in Figure 1.

For the transition diagram shown in Figure 2, the steady state equations are:

state s_0: $\lambda P_0 = M_f(1) P_1$

state s_i: $[\lambda + M_f(i)] P_i = \lambda P_{i-1} + M_f(i+1) P_{i+1}$

also: $\sum_{i=0}^{\infty} P_i = 1.$

From the recurrence equation, we have:

$$P_i = \{\prod_{j=1}^{i} [\lambda / M_f(j)]\} P_0.$$

$$\text{Now, } \sum_{i=0}^{\infty} P_i = \{1 + \sum_{i=1}^{\infty} \prod_{j=1}^{i} [\lambda / M_f(j)]\} P_0 = 1$$

$$\text{i.e., } P_0 = 1/\{1 + \sum_{i=1}^{\infty} \prod_{j=1}^{i} (\lambda [1 + \log(N+j)] / \mu_f)\}$$

$$= 1/\{1 + \sum_{i=1}^{\infty} (\lambda / \mu_f)^i \prod_{j=1}^{i} (1 + \log(N+j))\}$$

$$= 0 \text{ for any } \lambda > 0.$$

Hence, with probability 1, eventually the system is in s_∞, that is, the average response time is infinite. This conclusion is true <u>no</u> <u>matter</u> how fast the foreground process is.

This instability can be qualitatively attributed to the fact that if the system degrades a little then the probability that it will degrade even more increases.

In the following two subsections we propose two methods of avoiding such unstable behavior. Both techniques involve dynamically changing the priority of the background process relative to the foreground process. The first approach, called deterministic control, makes the background process a higher priority process whenever some parameter of the system exceeds a preset value. In the second approach, called stochastic control, at any given time the background process will preempt the foreground process with some probability.

2.2. DETERMINISTIC CONTROL: In this section, two deterministic policies for scheduling the background process are investigated along with their analytical models and performance results. The two analytical models yield results that agree very well with the predictions of the simulation experiments. The service time for the background process is assumed to be a constant while the service time of the kth request since the last upgrade is assumed to be equal to:

$$x(k) = \{1 + \log(N+k)\} / \mu_f$$

POLICY I: In the first policy, the background process is started whenever M requests are served by the foreground process since the last upgrade of the data-structure. Hence, assuming constant service time for the background process is justified.

The system in this case can be modeled by an M/G/1 system. The effect of the delay caused by the background process is taken into consideration by extending the service of the Mth customer since the last upgrade to include the time for upgrading the data structure and is equal to:

$$X(M) = [\{1 + \log(N+M-1)\} / \mu_f] + 1 / \mu_b$$

The arrival rate of the M/G/1 queue is λ, the average arrival rate of requests as before, but the queue utilization combines the foreground and the background utilizations and is equal to:

$$\rho = (\lambda / M) * [\{\sum_{k=0}^{M-1} [1 + \log(N+k)] / \mu_f + (1 / \mu_b)\}]$$

The average foreground waiting time is approximately equal to:

$$W_{f1} = \lambda * E[X^2] / \{2 * (1 - \rho)\}$$

where $E[\cdot]$ denotes the expectation and

$$E[X^2] = (1/M) * \{\sum_{k=0}^{M-2} ([1 + \log(N+k)]^2 / \mu_f^2)$$
$$+ [(1 + \log(N+M-1)) / \mu_f + 1 / \mu_b]^2\}$$

The analysis can be used to determine the value of M that minimizes the average waiting time and the value of M after which the system becomes unstable. It is found by simulation that these values are quite accurate. The latter value could be obtained by finding the smallest value of M such that:

$$\rho \geq 1$$

$$\text{i.e., } (\lambda / M) * \{\sum_{k=0}^{M-1} ([1 + \log(N+k)] / \mu_f) + 1 / \mu_b\} \geq 1$$

Figure 3 shows an example of how the average waiting time of the client's request is affected by the number of requests M that are processed before an update can be performed. For small values of M, the average waiting time decreases as M increases because of the large overhead in running the background process frequently and the accumulation of client requests during this data-structure update. The average waiting time reaches a minimum when M equal to 3 in this example and then increases as M increases as a result of the increase in system utilization and increase in the second moment of the service time.

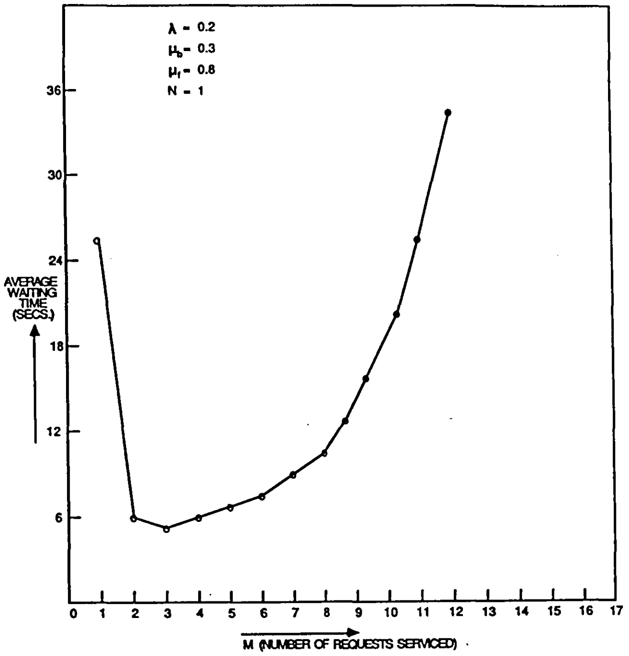


Figure 3. Performance of Policy I

POLICY II: In the second policy, the background process is run and is completed approximately every T seconds to update the data-structure. No preemption is permitted since it is costly in practical systems. The period T represents the time between the start of two consecutive updates performed by the background process. However, the actual period can be either equal to T if the cpu is idle (no foreground requests) or slightly different from T if the cpu is busy when it is time to run the background process. A decision is made whether to switch to the background process or handle one more client request depending on the arrival time of the request, on its service time which is known before hand, and on the time remaining in the current period of length T. In the period $(T-1.0/\mu_b)$ devoted to foreground requests, the system can serve a maximum of n customers such that

$$\sum_{i=0}^{n-1}[1 + \log_2(N+i)]/\mu_f \leq T - 1/\mu_b$$

Consider p(i) to be the probability that i requests will arrive and will be served in period T that starts when the background process is run (ignoring any old requests). Assuming poisson arrivals, the values of p(i) can be approximated by the following equations:

$$p(i) = [(\lambda T)^i/i!]e^{-\lambda T}, \text{ for } i < n;$$

$$p(i) = \sum_{i=n}^{\infty}[(\lambda T)^i/i!]e^{-\lambda T}, \text{ for } i \geq n.$$

The last equation assumes that that if more than n customers have arrived, the number of customers that are served is n. It should be clear that deciding if a request has been processed by the foreground process does not depend only on the number of arrivals and total time allotted for service in one period, but also on the detailed pattern of previous arrival times and service times in the underlying period and its predecessor periods as well. Hence, the above equations are an approximation of the complex real model. The mean number of customers $\underline{a}$ that are served every period T can be computed by the formula:

$$\underline{a} = \sum_{k=1}^{n} kp(k)$$

The average utilization of the cpu is equal to:

$$\rho = [\lambda/\underline{a}] \sum_{i=1}^{n} p(i)\{\sum_{k=0}^{i-1}[1.0+\log_2(k+N)]/\mu_f\} + 1.0/(\mu_b T).$$

The first term of the above equation represents the fraction of time in which the foreground requests are served and the second term represents the fraction of the time in which the background process is run. The above system can be modeled by an M/G/1 queue with adjusted rates. The average waiting time for the foreground requests are equal to the following:

$$W_{f2} = \{\lambda * E[X^2]\} / \{2 * (1 - \rho)\}$$

The average waiting time formula for the second scheduling policy is computed for $\mu_f = 0.8$, $\mu_b = 0.3$, and $\lambda = 0.2$ and several values of T and the results are shown in Figure 4. If T is very small, the system is unstable. Then, as T increases the average waiting time decreases because client requests have a larger probability of being served. This decrease continues until a minimum is reached (at T = 10.0 in this example). After this point, any increase in T will increase the average waiting time because of increasing system utilization and average service times.

The results of the analyses of the two policies agree quite well with the simulation predictions. The analytical models are found to be useful in providing the parameters for designing successful foreground/background operations.

2.3. STOCHASTIC CONTROL: In the following, we consider three policies reported in order of their effectiveness. We present some approximate analytical results along with some simulation results.

Case 1: Here, after every departure, the background process is invoked with probability (1-p). An approximate expression for the average service time is:

$$1/\mu_{avg} = \{1 + E[\log(d+N)]\}/\mu_f + (1-p)* \{(\lambda / \mu_{avg})/\mu_b + (1 - \lambda / \mu_{avg})[\lambda / (\lambda+\mu_b)]/\mu_b\}$$

The first term is the average service time and the second term is the delay due to the background process. There are two possibilities when the background process is invoked: If the system is not in the idle state (the probability of this event being λ / μ_{avg} in Figure 2 with $M_f(d) = \mu_{avg}$), then the delay is the full background service time. If the system is in the idle state (probability equal to $1 - \lambda / \mu_{avg}$), then the delay is only the

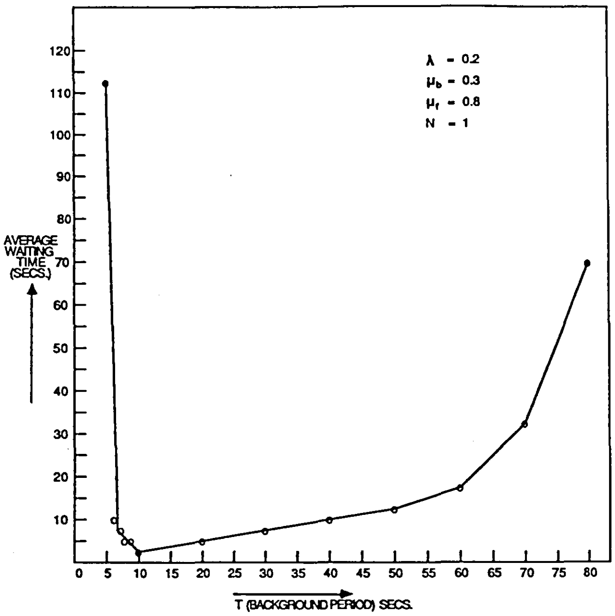


Figure 4. Performance of Policy II

residual time seen by the next arrival, i.e., $[\lambda / (\lambda + \mu_b)]/\mu_b$.

Table I shows the computed value of p which minimizes the average service time and the corresponding value found using simulation for some arbitrarily selected values of λ, μ_f, and μ_b. N = 1 in all the cases. The approximation ranges from medium to good.

Case 2: As in case 1, the background process is invoked to do a complete clean-up operation with probability (1-p′) after every departure. In addition, the background process is allowed to perform a partial clean-up during idle periods of the foreground process, i.e., it will upgrade the data structure as much as possible during the idle period. The value of p′ which minimizes the average service time should be greater than that in Case 1. An approximate result can be obtained by computing the value of p′ from the value of p for Case 1.

p = effective prob{do not clean up after
a departure}

= prob{do not clean up in Case 1 |
decide to clean up in Case 2}
* prob{decide to clean up in Case 2}

+ prob{do not clean up in Case 1 |
decide not to clean up in Case 2}
* prob{decide not to clean up in Case 2}

$$= 0 * (1-p') + \{P_0 * \lambda / (\lambda+\mu_b) + (1-P_0)*1\} * p'$$

where P_0 = prob{foreground process is idle}

$$\simeq 1 - \lambda / \mu_{avg}.$$

Hence, $p' = p/\{1 - (1 - \lambda / \mu_{avg}) \mu_b/(\lambda + \mu_b)\}$.

TABLE I

λ	μ_b	μ_f	CASE I			CASE II			CASE III	
			predicted P	observed P	Wmin	predicted P'	observed P'	Wmin	observed P"	Wmin
0.2	0.3	0.8	0.5	0.49	13.41	0.61	0.6	10.98	0.83	8.3
0.2	0.5	1.1	0.17	0.25	3.75	0.3	0.7	2.80	0.75	2.46
7.0	8.0	26.0	0.62	0.60	0.92	0.67	0.62	0.85	0.88	0.52
0.2	0.5	0.5	0.03	0.00	13.94	0.04	0.03	13.93	0.05	13.61
0.2	0.25	1.6	0.73	0.77	4.80	1.0	0.97	1.75	0.995	1.591

P, P', P" : Probabilities which minimize the average service time

Wmin : Waiting time corresponding to the tabulated values of P, P', P", respectively

Table I shows the computed value of p′ which minimizes $1/\mu_{avg}$ and the corresponding value found using simulation for the same set of values of λ, μ_b, μ_f as in Case 1. Again, N = 1. This is a very approximate calculation, and hence the agreement is worse than that in Case 1.

Case 3: In Cases 1 and 2, the decision to perform a complete clean-up is independent of the degradation level. Here, after every departure, the background process is invoked to do a complete clean-up with probability $(1-p''^{d})$, where d is the value of the degradation level. As in case 2, the background process is allowed to perform partial clean-ups during idle periods of the foreground process. The analysis of this case is complicated by the dependence of $1/\mu_{avg}$ on p′′ as well as on d. One possibility is to use an estimate of the average value of d. However, this double approximation renders a poor estimate. Nevertheless, we intuitively expect that the value of p′′ which minimizes $1/\mu_{avg}$ should be more than the corresponding ones of p and p′ for the same values of λ, μ_b, μ_f, and N. As shown in Table I, this is confirmed by simulation for the case where a complete clean-up is done with probability $(1-p''^{d})$.

A simulation study of these three cases appears in Figure 5. As can be seen, Case 3 yields the best results, i.e., the minimum average waiting time. Hence, we have adopted this strategy in the control of the relative priorities of the foreground and background processes in the experimental evaluation discussed in Section III.

III. AN EXPERIMENTAL EVALUATION OF CONCURRENT MAINTENANCE

In Section 3.1, a directory server for a hierarchical file system is presented to illustrate the foreground/background methodology in practice. The performance of an actual implementation of this system is presented in Section 3.2.

3.1 A DIRECTORY SERVER: The operations that the directory server can do are to create an entry for a new file, and to lookup, update or delete the entry for existing files. Files are identified by UNIX-like path names, e.g., /usr/xyz/filel. There

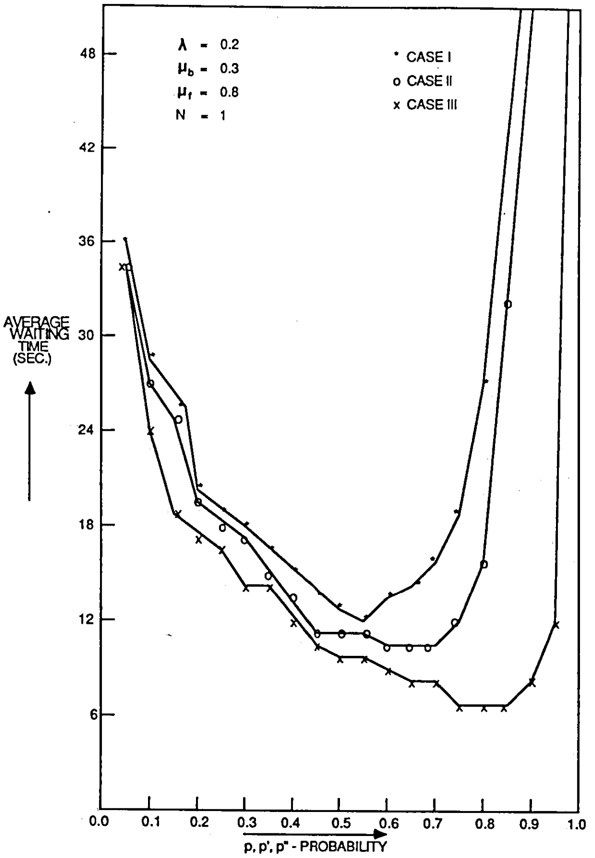


Figure 5. Comparison of Case I, II and III Using Simulation.

are two types of files, namely, directory and nondirectory files.

The data structure is a multiway tree in which the parent points to the first child and each child points to the next child. Since it is stored on secondary storage, performance can be improved by clustering the children of a node into as few pages as possible. In our case, the size of each entry is about 80 bytes, while each disk page consists of 512 bytes. Hence, six entries can reside in one page. The data structure diagram is shown in Figure 6 and the PASCAL declaration for the data structure is as follows:

```
const MaxEntriesPerPage = 6;

type address = ^page;
     entry = record
                status: (empty,occupied,deleted);
                info: . . .
                case type:
                    (directory,nondirectory) of
                  directory: (FirstChild: address);
                  nondirectory: ()
             end;
     page = record
               PreviousSibling,NextSibling: address;
               table: array [1..MaxEntriesPerPage]
                      of entry
            end;
```

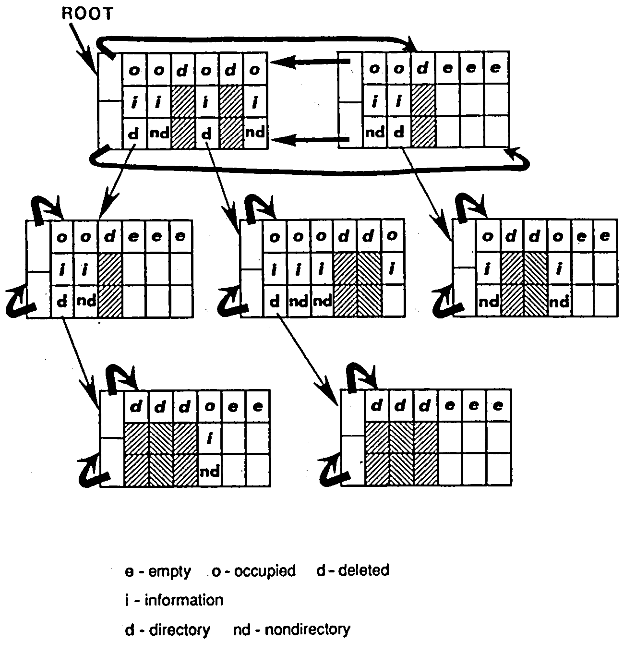


Figure 6. Data Structure Diagram for the Directory Server

```
var root: page;
```

The task of the foreground process is to carry out the client's request as fast as possible. It achieves this as follows:

(a) For create(a/b/.../j/k,type of file): If the file does not exist in the tree, and a/b/.../j exists and is of type directory, then k is added to an empty slot in the list of children of a/b/.../j. The status of the slot is changed to "occupied".

(b) For delete(a/b/.../j/k): If the file exists in the tree and it is a nondirectory file or there are no occupied slots in the list of children for the file, then the status of the slot containing the entry for k of children is changed to "deleted" in the list of children a/b/.../j.

(c) For LookUp(a/b/.../j/k) or update(a/b/.../j/k), the information corresponding to k in the list of children of a/b/.../j is either checked or modified depending on the operation. Pointers and status fields are not affected.

The background process removes slots which are marked deleted. In particular, it tries to bring the data structure into a state satisfying the following assertions:

(a) No page contains a slot marked "deleted";

(b) All slots marked "occupied" appear at the left most end of the table in each page;

(c) If a page contains a slot marked empty then it is the last page in this list.

The key procedure used by the foreground process is the following search function. It accepts a pathname and tries to locate it in the tree structure.

```
function search(root: page; s: path name)
returns found: boolean X depth: integer X p: page;

Precondition:

  root <> nil;

Postcondition:

  1) p is locked; all other pages locked by
     the procedure are unlocked;

  2) if the path specified by s exists
     in the tree then

     a) found = true
     b) depth = length(s)
     c) p <> nil; the path named by s
        leads to p

     else

     a) found = false
     b) 1 <= depth <= length(s)
     c) the path corresponding to the first
        "depth-1" names in s leads to the
        parent of p
     d) the path corresponding to the first
        "depth" names in s does not exist
        in the tree
     e) p <> nil; p is the last page
        in this list;
```

An example of its use is the following pseudocode for "create":

```
create(s,type) =>

found X depth X p:= search(s);
if found then
  error - file already exists
elsif depth < length(s) then
  error - parent does not exist
else
  {if there is no empty slot
        in p^.table then
    {q:= allocate and initialize a new page;
     lock q;
     add q to the list;
     unlock p;
     p:= q}
   add entry for the last name in s
    into an empty slot in p^.table,
    say p^.table[i];
  p^.table[i].status:= occupied;
  if type = directory then
    {r:= allocate and initialize a new page;
     p^.table[i].FirstChild:= r}
    }
unlock p
```

The background process does a breadth first traversal of the tree structure. If it finds a slot which is marked "deleted", it shifts everything in that chain of siblings to the left, deallocating all pages after the last page which contains an empty slot.

During the scanning phase, the background process does not lock any pages. In fact, it even ignores the locks of the foreground process. Once it finds an empty slot, it enters the cleanup phase. It then locks either 2 or 3 pages at a time in order to perform the cleanup operation. A detailed description of the implementation of these two processes appears in [CHE 85].

Noninterference of the foreground and the background processes is ensured by the following rely conditions:

(a) The foreground process relies on the condition that the tree structure formed by considering only occupied slots is left unchanged by the background process.

(b) The background process relies on the condition that the only actions of the foreground process with regard to the status of a slot is to change an "empty" slot to an "occupied" slot, or an "occupied" slot to a "deleted" slot.

(c) Absence of deadlocks can be guaranteed provided that each process locks pages in the left-to-right order along sibling chains.

(d) In order to ensure that some progress is made, each process relies on the other process to release a locked page within a finite time. In fact, maximum concurrency is possible provided that only the absolutely necessary pages are locked.

3.2 Implementation and Measurements:

The directory server discussed in Section III was implemented in C using UNIX interprocess communication primitives. It runs on a dedicated 3B2 computer [AT&T 84(A)] in a 3BNET local area network environment [AT&T 84(B)]. The server consists of five processes as shown in Figure 7.

(a) The "request receiver" accepts packets from the ethernet using UNIX 3BNET system calls. The packets are forwarded to the "foreground process".

(b) The "foreground process" executes the client's request by using UNIX file access and lock control facilities in order to search/update the tree structure. If it needs to allocate a page then it sends a request to the "file space manager" which, in turn, sends back the record number of a free page in the file containing the tree structure. The foreground process enqueues a response for the "request transmitter".

(c) The "response transmitter" sends a packet to the client via the ethernet.

(d) The "background process" removes slots which are marked "deleted". If any page becomes free then it informs the "file space manager".

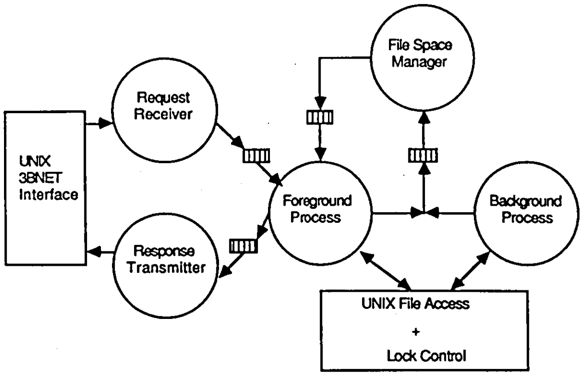


Figure 7. Structure of the Directory Server

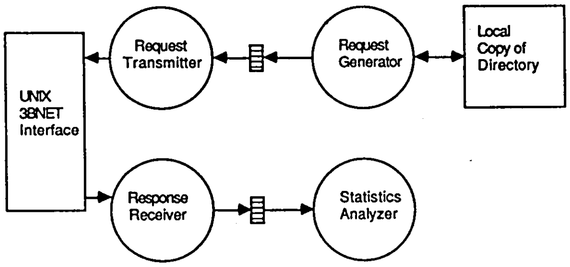


Figure 8. Structure of Clients

(e) The "file space manager" keeps a list of free file pages. It extends the size of the file only when the free list is empty and a new page must be allocated.

The experiment was conducted using a varying number of clients on a varying number of 3B2 computers. Each client consists of three processes as shown in Figure 8.

(a) The "request transmitter" is similar to the "response transmitter" in Figure 7.

(b) The "response receiver" accepts packets from the server. When the client generates a request it timestamps it. This timestamp is sent back with the response by the server. The "response receiver" can then compute the total time it has taken to process the request. This approach eliminates the need for a global clock.

(c) The "request generator" repeatedly executes the following code:

```
repeat

    generate a request

    update the local copy of the directory

    sleep for a period generated randomly using
    exponential distribution with mean ($\lambda$)

forever;
```

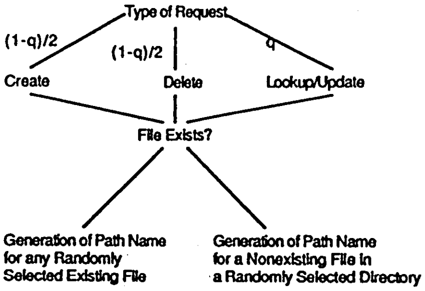

Type of Request
(1-q)/2
(1-q)/2
q
Create
Delete
Lookup/Update
File Exists?
Generation of Path Name for any Randomly Selected Existing File
Generation of Path Name for a Nonexisting File in a Randomly Selected Directory

Figure 9. Request Generator Program

It generates a request in two steps (see Figure 9). At first it determines the type of the request. It is "LookUp/update" with probability q, "create" with probability (1-q)/2, and "delete" with probability (1-q)/2. (If the probability that a request is "create" is not the same as the probability that it is "delete", then in the steady state either the directory will be empty or infinite. The steady state size of the directory, when the two probabilities are the same, is fixed by generating N "create" requests initially.)

Once the type of the request has been determined, the next step is to generate the full path name for either an existing file or a nonexisting file. This is done by consulting the local copy of the directory.

Thus, the parameters of the experiment are:

λ --> arrival rate

q --> LookUp/update probability

N --> steady state size of the directory

Figure 10 shows the results of the experiment. It also shows the measured response time for a server having no maintenance processes, but incorporating self-reorganizing data structures. Specifically, (a) during a search operation for the "create" request it keeps track of the first page containing an available slot, and (b) during a delete operation, it marks the slot as "deleted" and deallocates the page if it does not contain any "occupied" slots.

From Figure 10, we observe that self-reorganization has a significantly better performance than the implementation using a separate maintenance process. This is partly due to the system overhead in locking/unlocking pages and partly due to the contention with the background process. Since the code for the concurrent foreground/background processes is usually more complex than that for self-reorganization, careful study is required to determine situations where separate maintenance processes would be useful.

We have performed a series of experiments in

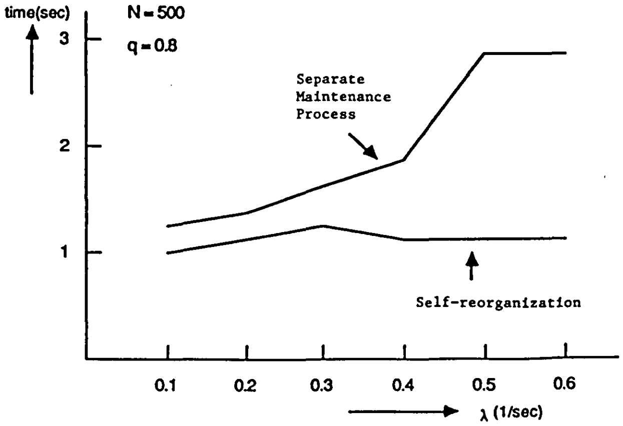


Figure 10. Experimental Results

order to compare the performance of stochastic control (a variant of case 2), deterministic control (policy I), and self-reorganization maintenance strategies. Because of time constraints, we were not able to include these experiments in this paper but they will reported in a forthcoming paper. The new results indicate that deterministic control has the best average case performance while stochastic control has the best worst case performance and the least variance among these three strategies. The performance of stochastic control can be improved significantly by using efficient synchronization primitives or coarser grained atomic actions. In general, concurrent foreground/background maintenance processes are competitive for real-time applications.

IV. SUMMARY

In this paper we have shown that the use of low priority maintenance processes for maintaining degradable data structures can result in unstable systems. This can be avoided by dynamically altering the relative priorities of the foreground and background processes. We have analyzed two control strategies for this purpose, namely. stochastic and deterministic policies. We have also presented the experimental evaluation of the performance of a concurrent directory server for a UNIX local area network environment and compared it with a server utilizing "on-the-fly" maintenance.

Some possible research areas include (1) refining the performance analysis of degradable systems, and extending it to cases where several foreground processes are active simultaneously, and (2) performing a comparative analysis of concurrent maintenance processes as opposed to the use of self-reorganizing data structures.

V. REFERENCES

[AT&T 84(A)] AT&T 3B2/300 Computer Owner/Operator Manual, AT&T Technologies, Inc., Oct. 1984.

[AT&T 84(B)] AT&T 3B2/300 Computer - AT&T 3BNET Manual, AT&T Technologies, Inc., Oct. 1984.

[BAS 85(A)] F.B. Bastani, S.S. Iyengar, and I.L. Yen, "Concurrent maintenance of multilevel data structures," Tech. Rep. UH-CS-85-3, Dept. Comp. Sc., Univ. Houston - Univ. Park, Jan. 1985.

[BAS 85(B)] F.B. Bastani, I.L. Yen, A. Moitra, and S.S. Iyengar, "Impact of parallel processing on software quality," Tech. Rep. UH-CS-85-5, Dept. Comp. Sc., Univ. Houston - Univ. Park, Apr. 1985.

[BEN 84] M. Ben-Ari, "Algorithms for on-the-fly garbage collection," ACM Trans. Prog. Langs. and Sys., Vol. 6, No. 3, pp. 333-344, Jul. 1984.

[CHE 85] I.R. Chen, A Distributed Directory Server in a UNIX Network Environment, M.S. Thesis, Dept. Comp. Sc., Univ. Houston - Univ. Park, July 1985.

[COU 85] P.-J. Courtois, "On the time and space decomposition of complex structures," Comm. ACM, Vol. 28, No. 6, June 1985, pp. 590-603.

[DIJ 78] E.W. Dijkstra, et al., "On-the-fly garbage collection: An exercise in cooperation," Comm. ACM, Vol. 21, No. 11, pp. 966-975, Nov. 1978.

[FUR 84] D.G. Furchtgott, and J.F. Meyer, "A performability solution method for degradable nonrepairable systems," IEEE Trans. Comput., Vol. C-33, No. 6, June 1984, pp. 550-554.

[HIC 84] T. Hickey and J. Cohen, "Performance analysis of on-the-fly garbage collection," Comm. ACM, Vol. 27, No. 11, pp. 1143-1154, Nov. 1984.

[JON 83] C.B. Jones, "Tentative steps towards a development method for interfering programs," ACM Trans. Prog. Langs. and Sys., Vol. 5, No. 4, pp. 596-619, Oct. 1983.

[LAM 84] B.W. Lampson, "Hints for computer system design," IEEE Softw., Vol. 1, No. 1, pp. 11-28, Jan. 1984.

[MAN 84] U. Manber, "Concurrent maintenance of binary search trees," IEEE Trans. Softw. Eng., Vol. SE-10, No. 6, pp. 777-784, Nov. 1984.

[MOI 85] A. Moitra, S. Iyengar, F. Bastani, and I. Yen, "Multilevel data structures: Models and performance," Tech. Rep. 85-679, Dept. Comp. Sc., Cornell Univ., May 1985.

[NEE 82] R.M. Needham, and A.J. Herbert, The Cambridge Distributed Computing System. Addison-Wesley Pub. Co., 1982.

[YEN 85] I-L. Yen, The Role of Parallel Processing in Application Programs, M.S. Thesis, Dept. Comp. Sc., Univ. Houston - Univ. Park, May 1985.

Construction Through Decomposition: A Divide-And-Conquer Algorithm for the N-Queens Problem

(Extended Abstract)

Bruce Abramson[1] and Mordechai M. Yung[2]

Department of Computer Science
Columbia University
New York, N.Y. 10027

Abstract

Configuring N mutually non-attacking queens on an N-by-N chessboard is a classical problem that was first posed over a century ago. Over the past few decades, this problem has become important to computer scientists by serving as the standard example of a globally constrainted problem which is solvable using backtracking search methods. A related problem, placing the N queens on a toroidal board, has been discussed in detail by Polya and Chandra. Their work focused on characterizing the solvable cases and finding solutions which arrange the queens in a regular pattern.

This paper describes a new *divide-and-conquer* algorithm that solves both problems, investigates the relationship between them, and develops a partial ordering for the related enumeration problem. The connection between the solutions of the two problems illustrates an important, but frequently overlooked, method of algorithm design: detailed combinatorial analysis of an overconstrained variation can reveal solutions to the corresponding original problem.

> The problem of the eight queens is a well known example of the use of trial-and-error methods and of backtracking algorithms. It was investigated by C.F. Gauss in 1850, but he did not completely solve it. This should not surprise anyone. After all, the characteristic property of these problems is that they defy analytic solution. Instead, they require large amounts of exacting labor, patience and accuracy. Such algorithms have therefore gained relevance almost exclusively through the automatic computer, which possesses these properties to a much higher degree than people, even geniuses, do.
>
> *-Nicklaus Wirth, Algorithms + Data Structures = Programs, p.143.*

1. Introduction

The preceding quote typifies the prevalent view of the N-queens problem. First posed by Max Bezzel in 1848, the problem of placing eight queens on an eight-by-eight chessboard in mutually nonattacking positions was solved in 1850 by Franz Nauck, who used trial-and-error methods to find 12 solutions. In 1874, S. Gunther proved Nauck's list exhaustive and J.W.L. Glaisher generalized the problem to placing N queens on an N-by-N board. Gunther and Glaisher proposed the following solution: Represent the board as a symbolic N-by-N matrix. Certain easily recognizable terms in this matrix' determinant indicate solutions to the N-queens problem. This approach, although helpful in the 8-by-8 case, is actually a brute force search on the N! terms in the determinant which looks for all solutions. In the early 1950's, A.M. Yaglom and I.M. Yaglom [25] found a pair of patterns which yield one solution for each N. A sketch of their solution, which remains relatively unknown, can be found in the appendix. Their suggestion cannot be generalized to give more than a unique solution, and is not based on a general algorithmic technique. For more details of the problem's early history see [2] [8] [17].

The N-queens problem has three variants: finding one solution, finding a family of solutions, and finding all solutions. Yaglom and Yaglom's patterns solve the first problem while Gunther and Glaisher's exhaustive search solves the third. The inherent difficulty of the N-queens

[1] Current address: Computer Science Department, University of California at Los Angeles. Supported in part by NSF grant IST-8418879

[2] Supported in part by NSF grant MCS-8303139 and an I.B.M. fellowship

variants is generally accepted; recent work has followed one of two basic approaches. The first accepts the problem as intractable and uses it as an example of trial-and-error methods like backtracking, while the second modifies the problem to one that does not defy combinatorial analysis. Examples of the former approach abound. Various fields, including algorithm design [12] [24] [6], program development [5] [23], and artificial intelligence [19] [10] [11] have relied on the N-queens problem to illustrate issues related to systematic heuristic search. In Artificial Intelligence the problem is one of the classic constraint satisfaction problems. Another such A.I. problem, the labeling of polyhedral scenes [22], has recently been proven to be NP-Complete [14]. As for the latter approach, analytically solvable variations of the N-queens problem can be grouped into two broad categories: those that reduce constraints [21] and those that increase them [20] [4].

This paper addresses the "family of solutions" variant. The method used in constructing this family involves the divide-and-conquer technique which suggests splitting the input into distinct subsets. This splitting property generally appears to be inapplicable when global constraints are involved. The basis of the solution is a combinatorially analyzable related problem, placing N queens on a toroidal board. Previous work on this variation focused on classifying the solvable cases [20] and finding *regular solutions*, setups that are obtained by placing queens on the board in a regular pattern [4]. The family of regular solutions solves the planar problem for a large class of N, but leaves almost as many board sizes unsolved. A non-regular family of solutions to the toroidal problem is developed here. This non-regular family is then modified to yield a general family of solutions to the N-queens problem for almost all N. This solution clarifies the relation between the toroidal and planar cases.

Sections 2 and 3 consist mainly of background material: section 2 discusses the N-queens problem as an example of backtracking techniques and constraint satisfaction problems, while section 3 introduces and outlines previous work done on the combinatorialy solvable toroidal problem. Section 4 then introduces the non-regular family of solutions to the toroidal problem, which leads to the general solution to the planar problem in section 5. Some conclusions and directions for future work are discussed in section 6. Along the way, some light is shed on the enumeration problem: a lower bound and a partial ordering on the number of toroidal solutions are shown.

2. Solutions Based on Search Techniques

The N-queens problem is an example of *constraint satisfaction*, a family of problems that involves assigning values to variables subject to a set of binary constraints [7] [19]. The standard solution method applied to these problems is backtracking search [12], a worst case exponential method which is usually supported by heuristics [10] [19] [15]. For a recent complexity analysis of backtracking search and general heuristic techniques see [3]. The popularity of the N-queens and other chessboard puzzles can be attributed to two factors: the long history of the chessboard and the relative simplicity with which problems related to it can be stated. In the N-queens problem, rows can be regarded as variables and columns as values. Given the board size, N, as input, the <variable,value> pairs can be expressed as a permutation P of 0 to N-1, where P(i) is the column of the queen in the i^{th} row. This representation alone is enough to guarantee that no two queens will be in the same row or column, leaving only the diagonal constraints to be verified. Since the diagonals going from top left to bottom right can be characterized by (i-P(i))=constant, and those from top right to bottom left by (i+P(i))=constant, P is a solution if and only if for $i \neq j$, $(i-P(i)) \neq (j-P(j))$, and $(i+P(i)) \neq (j+P(j))$.

The general constraint satisfaction problem has been discussed in detail in [10] [19]. We have shown that the general problem (over the integers) is NP-complete [1]. Most problems of interest, however, are not random instances of general constraint satisfaction, but special, structured cases of it, in which the constraints follow specific patterns. Many of them are rather difficult, including relatives of several NP-complete problems [9] and various hard problems in computer vision and artificial intelligence [11] [10] [14]. The N-queens, for example, can be viewed as finding an independent set of a special graph. Simply view the board as a graph with a vertex for each square and an edge between any two vertices representing squares in the same row, column or diagonal. A solution to the N-queens problem is a set of N squares sharing neither row, column, nor diagonal, or a set of N independent vertices.

3. The Toroidal Problem and Regular Solutions

Several interesting variations to the N-queens problem have been proposed. One that has attracted a good deal of attention is the N-superqueens problem. A *superqueen*, introduced in [20], is a queen which upon reaching an edge of the board can wrap around to the opposite edge, in effect treating the board as a torus. The superqueen places additional constraints on the board by connecting previously separate diagonals. The resulting (toroidal) board has N rows, N columns, and two sets of N diagonals (characterized by {(row-column)=constant mod N} and {(row+column)=constant mod N}), each containing N squares. This symmetry makes the N-superqueens problem easier to analyze combinatorially than the N-queens. Polya [20] showed that an N-superqueens solution exists if and only if N is not divisible by 2 or 3. Since the N-superqueens is an overconstrained variation, any N-superqueens solution solves the N-queens as well.

Chandra [4] developed the theory of *independent permutations* which he used to characterize a family of solutions to the N-superqueens problem, the *regular solutions*. A regular solution is one in which the permutation P can be characterized by P(i)=Ai+B (mod N). The permutation P(i)=2i, for example, starts with a superqueen in the upper left hand corner and proceeds around the board placing queens one row down and two columns across. This solution is called the *knight-walk*. Chandra also built on the result of [20] by showing that for any N, if the N-superqueens problem is solvable and M is the smallest factor of N (M > 1), a family of regular solutions exists, and is characterized by P(i)=Ai, A=2,4,...2^k, where k= [$\log_2 M$] -1. Since P(0)=0, each member of this family has a superqueen in the upper left hand corner. Only regular solutions exist for N < 13. For N=13, however, the non-regular permutation P=(0 3 8 11 5 1 10 4 7 12 2 9 6) is a solution as well. Some regular extensions of the toroidal problem are discussed in [4] [13].

As an immediate extension of their existence for any solvable N-superqueens problem, the family of regular solutions described above solves the N-queens problem for 2/3 of the odd numbers. Furthermore, removing the top row and leftmost column from the N-by-N board deletes only the queen in the top left hand corner square. This leaves an (N-1)-by-(N-1) board with (N-1) mutually nonattacking queens, solving the (N-1)-queens problem, and thereby constructing solutions for 2/3 of the even numbers. No other simple board modifications are possible, since removal of any other row and column would either delete two queens, shift the diagonals, or both. Regular solutions and upper left hand corner-removal, then, constitute a solution scheme for 2/3 of all N.

4. The Decomposition Solution to the Superqueens Problem

This section shows how to construct a family of non-regular solutions to the N-superqueens problem, the decomposition solutions. The main idea is to use the factorization of N to apply a divide-and-conquer approach to the problem. Basically, if N can be factored as AB where both A and B are solvable, the N-superqueens problem can be reduced to solving A appropriately chosen copies of the B-superqueens problem.

Definition: Let N=AB, where there are solutions to the A-superqueens and B-superqueens problems. A decomposition solution breaks the N-by-N board into an A-by-A grid of B-by-B tiles. Tiles corresponding to an A-superqueens solution are filled with a B-superqueens solution; the same B-superqueens solution is used throughout.

In order to simplify further discussions, some additional definitions are necessary.

Definition: Let LR(N) refer to the set of diagonals running from top left to bottom right on an N-by-N board. Let OLR(N) refer to those members of LR(N) which are occupied by a queen. Similarly, let WLR(N) refer to the set of N wrapped diagonals on a planar representation of an N-by-N toroidal board running from top left to bottom right. Let OWLR(N) refer to those members of WLR(N) which are occupied by a superqueen. Let the diagonals from top right to bottom left be named correspondingly (that is, RL(N), ORL(N), WRL(N), and OWRL(N)).

As an illustration of the manner in which a decomposition solution solves the problem, look at the example in figure 1, where N=35, A=7, and B=5. The knight-walk is used as both the 5 and 7-superqueens solutions. A board set up according to the definition of decomposition clearly contains 35 superqueens, with no two in any one row or column. That no diagonal contains two superqueens, however, is a bit less obvious. The trick here is to realize that the diagonals that result from tiling a plane with 5-by-5 boards are equivalent to those resulting from placing a 5-by-5 board on a torus. As shown in the diagram, a diagonal in WLR(35) passes alternately through members of LR(5) of lengths 2 and 3. These are exactly the LR(5) diagonals which combine to form a single element of WLR(5). The use of a 5-superqueens solution guarantees that only one of these LR(5) diagonals contains a superqueen. This reduces the realm of possible conflicts to superqueens in the same position on different boards. Examination of the figure reveals that if a member of WLR(35) passes through corresponding LR(5) diagonals on two 5-by-5 tiles, the tiles lie along the same LR(7) diagonal. The use of a 7-superqueens solution precludes the possibility of two such boards being chosen, and thus no two superqueens can be in the same member of WLR(35). Thanks to the symmetry of the board, an identical argument can be used for WRL(35).

Theorem 1: A decomposition solution solves the N-superqueens problem.

In order to discuss the general case, some machinery must be developed for referring to individual squares on a decomposed board. Each square's location can be specified

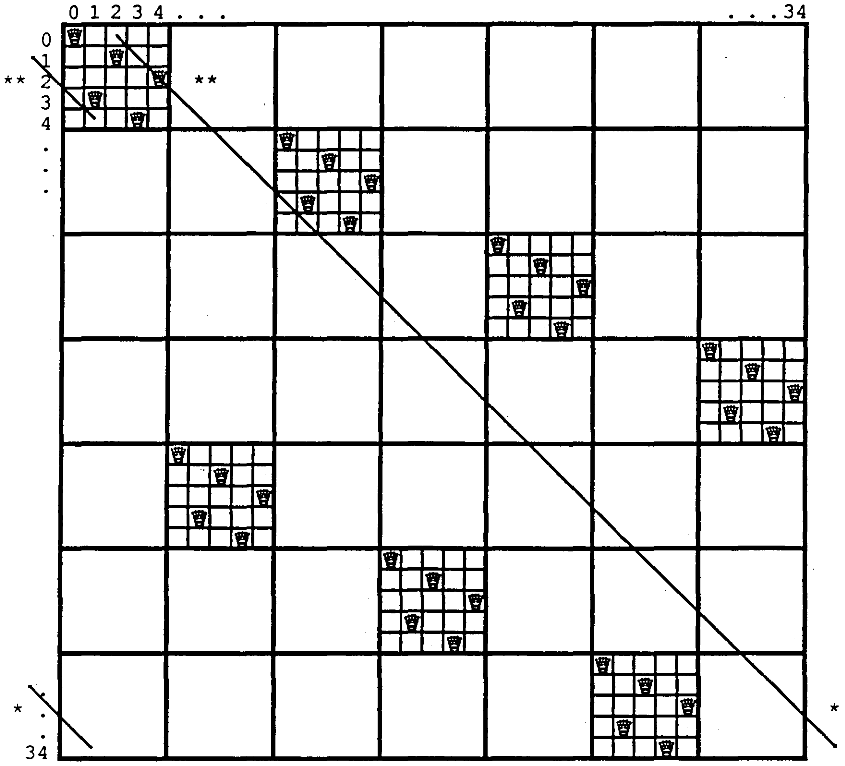


Figure 1: An example of a Decomposition Solution. N=35, A=7 and B=5. A diagonal is drawn, along with its continuations in both WLR(B) ** and WLR(N) *.

in two coordinate systems: the N-by-N system and the B-by-B system within the A-by-A grid. In converting between the systems, let $<i_N,j_N>$= tile $<i_A,j_A>$ square $<i_B,j_B>$, where i_N,j_N=0 to N-1, i_A,j_A=0 to A-1, i_B,j_B=0 to B-1. Since superqueens solutions are used throughout, all boards must be treated as tori, and therefore, all arithmetic is done modulo the subscripted number. The relationship between the systems is simple. Each tile contains B rows and B columns. Thus, $i_N=Bi_A+i_B$ and $j_N=Bj_A+j_B$. In other words, to determine the location of a square on the N-by-N board, find out which tile it is in, multiply each tile index by B to determine how many rows or columns preceded it, and add the number of rows or columns preceding it in its tile. Furthermore, since rows and columns within a tile are counted modulo B, the row following the (B-1)st in any tile is the 0th row of the following tile. The same is true of columns. In the example of figure 1, where N=35, A=7, and B=5, if $<i_N,j_N>$=<14,23>, the square in question is tile <2,4> square <4,3>. Moving one square to the right, <15,23> is tile <3,4> square <0,3>. Two squares *correspond* if they have the same B coordinates. In the example of figure 1, <7,4>= tile <2,1> square <2,4> and <12,14>= tile <3,3> square <2,4> correspond with each other. Using this notation, the proof of the theorem becomes a straightforward generalization of the case shown in figure 1.

One implication of theorem 1 is that every ordered split of N into its (not necessarily prime) factors corresponds to at least one decomposition solution. Given such a split, $N=f_1f_2...f_k$, the first factor plays the role of A, breaking the N-by-N board into a grid of (N/f_1)-by-(N/f_1) tiles. Any solution to the f_1-superqueens problem can be used to choose tiles. The rest of the factors are used recursively to fill the tiles specified by the chosen f_1-superqueens solution. Ergo, if M is a proper divisor of N, any M-superqueens solution gives rise to at least one N-superqueens solution. Since there is at least one N-superqueen solution that is not built up from any M-superqueen solution, namely the knight-walk, there are fewer M-superqueens solutions than N-superqueens solutions. This induces a partial order on the number of N-superqueens solutions.

Corollary 1: If there is a solution to the N-superqueens problem, and M is a proper divisor of N, there are fewer M-superqueens solutions than N-superqueens solutions.

The solution family introduced in this section also reveals a lower bound on the number of solutions for a given N. By the corrolary, each ordered split of N corresponds to at least one decomposition solution. Consider the sequence N={5^i}, where (i ≥ 1). There are 2^{i-1} different ordered splits of 5^i, corresponding to the number of ways to distribute i indistinguishable objects (the 5's) into j distinguishable cells (the positions in the ordered split) for j=1 to i, with at least one object in each cell. These ordered splits indicate that there are at least 2^{i-1} different solutions. In other words, this shows a lower bound of O($N^{\log_5 2}$). Further exploitation of properties of integer sequences and techniques of combinatorial enumeration may be helpful in finding a better lower bound for this problem.

5. A General Solution to the Queens Problem

This section develops a general solution scheme which connects the toroidal and planar problems. A simple modification of toroidal decomposition yields a family of solutions to the previously unsolved planar boards. The key to this modification lies in the major difference between the knight-walk and decomposition. The knight-walk uses individual superqueens as basic blocks. The interplay between these blocks severely limits the ways in which the solution can be modified. Thus, the knight-walk is of limited use in the construction of a general solution. Decomposition, on the other hand, relies on pre-solved boards as blocks. The size of these blocks offers a great deal of flexibility in terms of modifications to the solution, allowing decomposition to serve as the infrastructure of a general solution.

Consider a specific type of decomposition, a D-solution, in which the A-superqueens solution contains a superqueen in the upper left hand corner and the B-knight-walk is used. The scheme for constructing a general N-queens-solution consists of appropriately modifying a D-solution. This section can be divided into two parts. The first part shows that replacing the top leftmost tile of a D-solution with a smaller tile does not violate any of the problem's constraints. The second part proves that such modifications provide decomposable N-queens solutions for nearly all remaining cases.

Lemma 1: If there is a solution to N-superqueens and N=AB, there is a solution to the B(A-1)-queens problem.

Consider a D-solution. A simple removal of the top row and leftmost column from the A-superqueens board corresponds to removing the top B rows and B leftmost columns from the N-superqueens board. Just as a solution exists for (A-1)-queens, one exists for (N-B)=B(A-1)-queens.

This modification alone is not enough. In order to proceed with the discussion, one more definition is needed:

Definition: For all P ≤ N, call the bottom P rows of the P rightmost columns of an N-by-N board the lower right sub-board of order P.

The notion of a lower right sub-board is important in showing that replacing the top leftmost tile of a D-solution with a smaller tile does not violate any of the problem's constraints. Consider the example of figure 2, which shows a 7-by-7 board and its lower right sub-board of order 5. The numbers drawn represent the 5 and 7 knight-walks, respectively. Note that the first three superqueens placed on the 5-by-5 sub-board fall in members of OLR(7). The remainder of the 5-superqueens fall in squares containing 7-superqueens. Thus, OLR(5) is a subset of OLR(7). A similar argument holds in the general case, as well.

Lemma 2: Given the knight-walk solutions to the B-queens and C-queens problems (B-knight-walk and C-knight-walk, respectively), C ≤ B, replacing B's lower right sub-board of order C with the C-queens solution does not add new diagonals to OLR(B).

Lemma 3: If there are solutions to N-superqueens and C-superqueens, N=AB, and C ≤ B, then there is a solution to the B(A-1)+C-queens problem.

These lemmas enlarge the class of board sizes solved by the methods discussed here to all N=B(A-1)+C for some A,B,C not divisible by 2 or 3, and C ≤ B. Clearly, knight-walk and decomposition are subsumed by this scheme. For a decomposition set C=B, and for a knight-walk set C=B=1. As an informal proof of the validity of the lemmas, note that lemma 2 is simply a general extension of the case discussed above (figure 2). For lemma 3, consider a D-solution. Rather than removing the top B rows and leftmost B columns (as was done in lemma 1), remove only the top (B-C) rows and leftmost (B-C) columns, and replace the remainder of the top left corner tile with the C-knight-walk. By Lemma 2, no diagonal constraints are violated, leaving a solution to the B(A-1)+C queens problem. This is a *general solution.*

Definition: Let N=(A-1)B+C, where there are solutions to the A, B, and C-superqueens problems, and C ≤ B. A general solution starts with a D-solution to the AB-superqueens problem. The top B rows and B leftmost columns of the decomposition are replaced with C rows and C columns. The C-knight-walk is then placed in the newly created C-by-C tile in the upper left hand corner of the board.

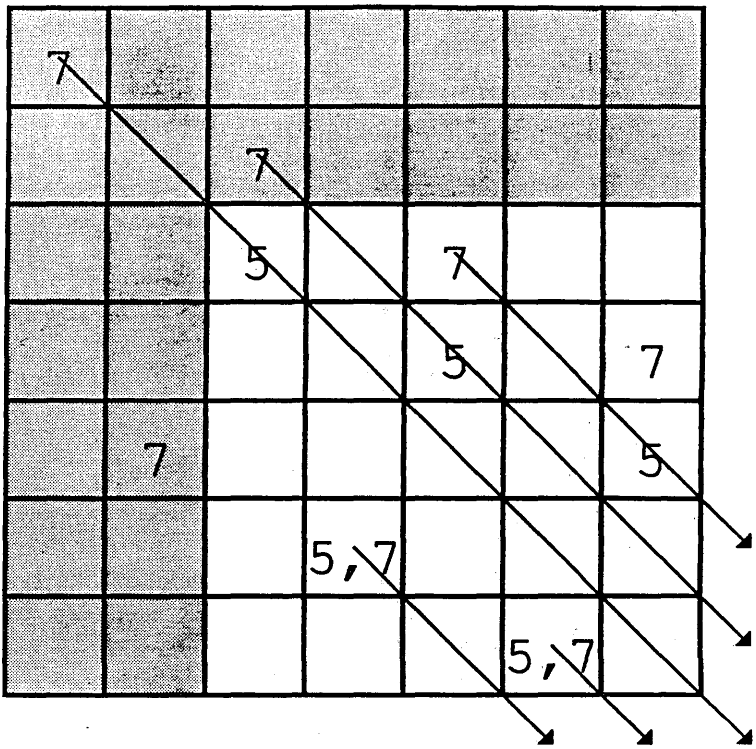


Figure 2: A 7-by-7 board and its lower right subboard of order 5. The knight-walk solutions have been drawn in.

The question remains, for which of the remaining board sizes does this general solution work? The following two lemmas provide the answer: (almost) all N. In order to find a general solution there must first be a D-solution that can be modified appropriately. The need for a D-solution, in turn, points to the importance of determining the conditions under which A and B exist such that $B(A-1) < N < BA$, and then finding an appropriate A and B.

Lemma 4: Let N be an odd number divisible by 3. If there exist odd numbers A and B, $A=2 \bmod 3$ and $B \neq 0 \bmod 3$, such that $B(A-1) < N < BA$, then there exists a solution to the N-queens problem.

Let $N=B(A-1)+C$. Then:

- $\{B(A-1)<N<BA\}$ implies $\{0<C<B\}$.
- $\{A=B=1 \bmod 2\}$ and $\{N=1 \bmod 2\}$ imply $\{B(A-1)=0 \bmod 2\}$, and thus $\{C=1 \bmod 2\}$.
- $\{A=2 \bmod 3\}$ and $\{B \neq 0 \bmod 3\}$ imply $\{B(A-1) \neq 0 \bmod 3\}$.
- $\{B(A-1) \neq 0 \bmod 3\}$ and $\{N=0 \bmod 3\}$ imply $\{C \neq 0 \bmod 3\}$.

In other words, if an appropriate D-solution can be found, a C that meets the requirements of lemma 3 can be found as well: odd, not divisible by 3, and no larger than B.

Lemma 4 characterizes the family of general solutions for a given N. In lemma 5, one member of the family is shown to be applied to almost all N. This solution is computable in linear time.

Lemma 5: For odd N divisible by 3, $N \neq 3,9,15,27,39$, setting $A=5$ guarantees the existence of a B that meets the specifications of lemma 4.

The proof of lemma 5 follows directly from the requirements of lemma 4. According to these specifications, B must be an odd number not divisible by 3, or a member of one of two equivalence classes, (i) $B=1 \bmod 6$ and (ii) $B=5 \bmod 6$. Let N be an odd number divisible by 3. Let B_1 be the largest B such that $4B_1 < N$. Since B_1 is well defined only for $N > 3$, the case of $N=3$ is ruled out of consideration. (In fact, there is no solution to the 3-queens problem). If $N < 5B_1$ then B_1 is as required by lemma 4 (set $C=N-4B_1$, and A,B, and C are all defined as explained above), so assume

$N > 5B_1$. Let B_2 be the smallest B larger than B_1, or the smallest B such that $4B_2 > N$. If B_1 is in equivalence class (i), $B_2=B_1+4$. Otherwise, B_1 is in equivalence class (ii), and $B_2=B_1+2$. In the first case, $5B_1 < N < 4B_2=4(B_1+4)$, which implies $B_1 \leq 14$. The only numbers which satisfy these conditions for B_1 are 1, 7, and 13. In the second case, $5B_1 < N < 4B_2=4(B_1+2)$, which implies $B_1 \leq 6$. The only number which satisfies these conditions for B_1 is 5. The intervals of candidates for N, then, are $5B_1 < N < 4B_2$, where B_1 is one of the aforementioned four numbers. In other words, one of the following is true: $\{5 < N < 20\}$, $\{25 < N < 28\}$, $\{35 < N < 44\}$, or $\{65 < N < 68\}$. The only odd numbers divisible by 3 in these intervals are 9, 15, 27, and 39.

Although a family of solutions is given by any triple (A-1,B,C) meeting the specifications of lemma 3, assigning A a value other than 5 does not solve any of the remaining cases. Since lemma 4 required that A=2 mod 3, the next value that A could be assigned is 11. The minimal non-trivial value for B, however, is 5, (recall B=1 yields the knight-walk), so the smallest board size that A=11 could solve is 50. This is larger than the largest unsolved board. As for the hitherto unsolved even numbers, the construction described in the general solution guarantees the existence of a queen in the upper left hand corner. Removing the top row and leftmost column gives solutions to all even numbers but 2,8,14,26 and 38. As far as these few cases are concerned, the N-queens problem is unsolvable if N=2 or 3. Lemmas 1 through 5 can be summarized as:

Theorem 2: The general scheme yields a non-empty family of solutions to the N-queens problem for all N, $N \neq 2,3,8,9,14,15,26,27,38,39$.

The algorithm in figure 3 gives one member of the general solution family for each N. In this algorithm, the number of arithmetic operations is constant. If bit operations are counted instead, when the input length is n ($n=\log_2 N$), the arithmetic calculations cost n. In either model, the dominant cost is the length of the output. When computing families of solutions, notice that decomposition relies on number factorization and arranging of ordered factorizations. The best well-known algorithm for factorization has an expected running time of $(\exp(\sqrt{n \log n}))$ [16], although a recent result has reduced the complexity to $(\exp(p \log p))$, where p is the length of the largest prime factor of N [18]. The family of general solutions for a given N can be found by considering all $O(N^2)$ potential triples <A,B,C>. Thus, any family can be computed in time bounded by $(N^2\, n^{1+\varepsilon})$.

```
Algorithm Nqueens;
Begin
Input(N);               {The board size}
Case N of:
  2,3 : Output (No Solution);
  8,9,14,15,26,27,38,39 : Output (Special-solution(N));   {Table look up}
  Otherwise:  Begin  {Apply general solution scheme}
     Even <- false;  {Solve for an odd number.}
     If (N is even) then Begin     {Add one, solve, then drop the extra corner}
                          N <- N+1;
                          Even <- true;
                        End;
     If (N mod 3 ≠ 0) then Board <-Regular-solution(N)
      Or Board<-Decomposition-solution(N);
     Else Begin
            Find B,C s.t. 4B+C=N;      {Set A=5, find B and C}
            Board <- general-solution(N,5,B,C);
          End;
     If (Even) Then Board <- Board-minus-corner-queen;   {Drop the corner}
     Output(Board);
          End;
End.
```

Figure 3: An algorithm that finds one solution to any instance of the N-queens problem.

6. Conclusions

This paper investigated and presented new families of solutions to two related problems, the N-queens and the N-superqueens. Both are examples of constraint satisfaction and both have been extensively studied in the past. The exact nature of the relationship between the two problems has not been well understood, and the belief that N-queens is in fact an intractable problem, best solved by backtracking, persists. This paper tries to provide possible resolutions to each of these difficulties by using the divide-and-conquer approach. A family of non-regular solutions to the N-superqueens problem is shown. These superqueen-decomposition solutions are then combined to form queens solutions. This is an example of lifting a result found for a special case (toroidal board) to a general case (planar board), and raises the question of whether other well understood algebraic or combinatorial problems can be used to solve their less well understood relatives.

Some open questions are raised by this work. Specifically, can other methods produce different N-queens solution families, or give more information about the enumeration problem? More generally, can the approach used here help solve other problems? We believe that many special cases of interesting intractable problems should be studied in the same fashion: take a new approach and investigate carefully the special properties of the structure of the problem in question.

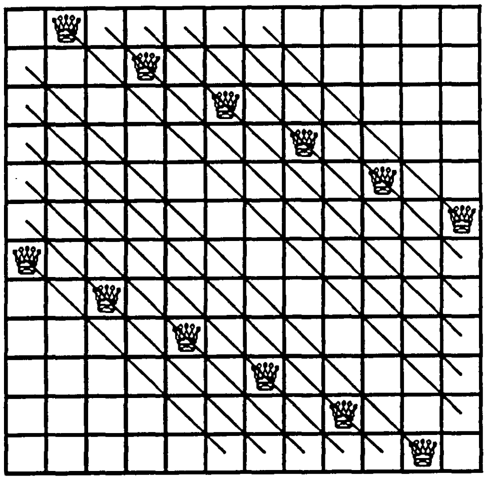


Figure 4a: An example of Yagloms' solution for N of the form N=6m or N=6m+4. Here N=12.

Acknowledgements

We would like to thank Zvi Galil and Richard Korf for their interest in our work and for their encouragement. Special thanks to Ashok Chandra and Martin Gardner who pointed out previous work done on the problem.

Appendix: The Yagloms' Solution

Yaglom and Yaglom described a solution to the N-queens problem for all N in [25]. They concentrated on giving one solution for even board sizes without placing queens in the main diagonal. That vacancy allowed the addition of a row, column, and queen to solve odd sized boards. For even N of the form N=6m or N=6m+4, the setup they describe is shown in figure 4a. It is basically the knight-walk of this paper (more accurately, the knight-walk minus the queen in the upper left hand corner). For board sizes of the form N=6m+2, however, a totally different setup is needed. They exhibited a pattern which works for these boards. Proceeding rightward from the leftmost column, placing successive queens in the diagonals specified by the following pattern (using the diagonal numbering scheme shown in figure 4b), solves the problem:

2n-4,n+1,n+2,n+3,...,3n/2-3,n/2+2,3n/2-1,n/2+1,3n/2-2, n/2+3,n/2+4,n/2+5,...,n-1,4

The example of N=14 is shown in figure 4b.

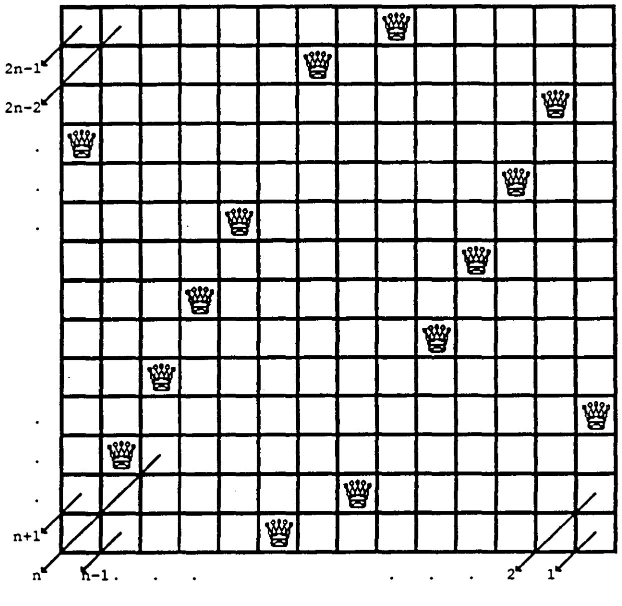


Figure 4b: An example of Yagloms' solution for N of the form N=6m+2. Here N=14. The diagonal numbers have been drawn in.

References

1. Abramson, B. and Yung, M.M. Construction Through Decomposition: A Linear Time Algorithm for the N-queens Problem. Columbia University, .

2. Ball, W.W.R. *Mathematical Recreations and Essays.* Macmillan, New York, 1973.

3. Carter, L., L. Stockmeyer and M. Wegman. The Complexity of Backtrack Searches. Symposium on Theory of Computing, ACM, May, 1985, pp. 449-457.

4. Chandra, A. K. "Independent Permutations, as Related to a Problem of Moser and a Theorem of Polya". *Journal of Combinatorial Theory 16*, 1 (January 1974), 111-120.

5. Dijkstra, E.W. Notes on Structured Programming. In *Structured Programming*, Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R., Ed., Academic Press, New York, 1972, pp. 1-82.

6. Floyd, R.W. "Nondeterministic Algorithms". *JACM 14*, 14 (1967), 636-644.

7. Freuder, E.C. "A Sufficient Condition for Backtrack-Bounded Search". *Journal of the Association for Computing Machinery 32*, 4 (October 1985), 755-761.

8. Gardner, M.. *The Unexpected Hanging and Other Mathematical Diversions.* Simon and Schuster, New York, 1969.

9. Garey, M.R. and Johnson, D.S.. *Computers and Intractability.* W.H. Freeman, New York, 1979.

10. Gaschnig, J. *Performance Measurement and Analysis of Certain Search Algorithms.* Ph.D. Th., Carnegie-Mellon University, May 1979.

11. Haralick, Robert M. and Elliot, Gordon L. "Increasing Tree Search Efficiency for Constraint Satisfaction Problems". *Artificial Intelligence 14*, 3 (October 1980), 263-313.

12. Horowitz, E. and Sahni, S.. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, Md., 1978.

13. Hwang, F.K. and Lih, Ko-Wei. "Latin Squares and Superqueens". *Journal of Combinatorial Theory, Series A 34*, 1 (January 1983), 110-114.

14. Kirousis, L.M. and Papadimitriou C.H, The Complexity of Recognizing Polyhedral Scenes. Symposium on Foundation of Computer Science, IEEE, October, 1985, pp. 175-185.

15. Knuth, D.E. "Estimating the Efficiency of Backtrack Programs". *Mathematics of Computation 29*, 129 (January 1975), 121-136.

16. Knuth, D.E. *The Art of Computer Programming.* Volume 2: *Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, 1981.

17. Kraitchik, M.. *Mathematical Recreation.* W.W. Norton, New York, 1942.

18. H.W. Lenstra, Jr. Factoring Integers with Elliptic Curves. Mathematisch Institut, University of Amsterdam, 1986.

19. Pearl, J.. *Heuristics.* Addison-Wesley, Reading, Massachusetts, 1984.

20. Polya, G. Uber die 'doppelt-periodischen' Losungen des n-Damen-Problems. In *Mathematische Unterhaltungen und Spiele*, Ahrens, W., Ed., Teubner, Leipzig, 1918, pp. 364-374.

21. Robert Wagner and Robert Geist. The Crippled Queen Placement Problem. Duke University, January, 1984.

22. Waltz, D. Understanding Line Drawings in Scenes With Shadows. In *The Psychology of Computer Vision*, Winston, P., Ed., McGraw-Hill, New York, 1975, pp. 19-91.

23. Wirth, N. "Program Development by Stepwise Refinement". *CACM 14*, 4 (April 1971), 221-227.

24. Wirth, N.. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

25. Yaglom, A.M. and Yaglom I.M.. *Challenging Mathematical Problems With Elementary Solutions.* Holden Day, San Francisco, California, 1964.

TWO FLOW ROUTING ALGORITHMS
FOR THE
MAXIMUM CONCURRENT FLOW PROBLEM

Jit Biswas* - David W. Matula**

* Department of Computer Science, University of Texas at Austin
** Department of Computer Science, Southern Methodist University

ABSTRACT

We describe two heuristic flow routing algorithms and analyse their convergence to the solution of the Maximum Concurrent Flow Problem (MCFP) [Ma83], [Ma85]. Implementation results demonstrate the algorithms to be robust. Both algorithms provide a feasible concurrent flow along with a bound on the maximum concurrent flow. The greedy flow augmentation procedure is most efficient but can yield suboptimal results. The flow rerouting procedure can be shown to produce results within an arbitrary tolerance of optimality. These flow routing algorithms make feasible the solution of a larger MCFP than can be handled by linear programming techniques [Pa84]. The algorithms may be interpreted as state space search algorithms regarding artificial intelligence models and therefore generalize to allow non-linear constraints. Applications are in packet switching networks [At81] and cluster analysis [Ma83].

1. Introduction and Summary

Informally the Maximum Concurrent Flow Problem (MCFP) can be stated as follows: Let there be a network of entities (cities, neighborhoods, computers, etc.) in which there exists traffic or flow (passengers, messages, etc.) between all pairs of entities. The flow is sustained through channels (air links, roads, communication lines etc.) which are capacitated. We seek to assign flows in a manner such that the flow relative to demand between any pair of entities (throughput) is equal to the flow relative to demand between any other pair of entities. We then ask for the maximum throughput of such concurrent flow that can be attained in the network, respecting capacity constraints.

For example, consider the interconnection network with sixteen processors shown in Figure 1, which is representative of the structure of the ILLIAC IV. In this case it can be shown [MD82] that there exists a set of paths including at most two paths between any pair of processors (vertices) such that concurrently a total of two units of flow (concurrent flow throughput) can be transmitted between each of the one-hundred and twenty vertex pairs over the respective paths subject to a capacity limitation of sixteen units of total flow over each of the thirty-two linkages (edges) between processors. Our methods here provide routing solutions for concurrent flow throughput for non regular topologies as well as the regular topology of Figure 1.

Section 2 provides a brief background to the maximum concurrent flow problem, giving two linear programming (LP) formulations and their complexities. Section 3 considers algorithmic characteristics for the MCFP from the artificial intelligence point of view. Section 4 develops criteria that are used to simplify the problem formulation and provide convergence bounds. Sections 5 and 6 analyse, successively, a greedy flow augmentation algorithm, and a more sophisticated flow rerouting algorithm, for determining a concurrent flow. Both algorithms are illustrated by application to a set of test graphs.

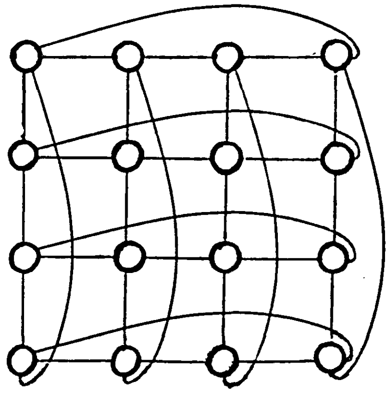


Figure 1: An interconnection network for sixteen processors with thirty-two direct linkages.

----

*Supported in part by the Army Research Office contract number DAAG 29-84-K-0060.

CH2345-7/86/0000/0629$01.00 © 1986 IEEE

2. Linear Programming Formulations of the Maximum Concurrent Flow Problem

There are two main formulations of the MCFP, the Node-Arc formulation and the Edge-Path formulation [Ma83], [Ma85], [Pa84]. In our formulation the terms edge and arc are synonymous as are the terms node and vertex. Notationally, we assume throughout this paper that the graph $G = (V,E)$ has vertex set $V = \{1,2,\dots,n\}$ and edge set E.

Node-Arc Formulation of the MCFP. Let the ordered pair (i,j) for $i<j$ denote the commodity flowing from vertex i to node j. In a graph on n vertices there are $\binom{n}{2}$ commodities. We impose conservation equations which set the sum of the incoming flow on edges equal to the outgoing flow on edges for each vertex. In addition there are demands for flow between each vertex pair and capacity constraints on each edge. Thus the MCFP can be formulated as:

Maximize the concurrent flow throughput z subject to the following constraints:

$$\sum_{k=1}^{n} f_{k\ell}^{(ij)} - \sum_{m=1}^{n} f_{\ell m}^{(ij)} = \begin{cases} -D_{ij}z & \text{if } \ell = i \\ +D_{ij}z & \text{if } \ell = j \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$

for $1 \le \ell \le n$, $1 \le i < j \le n$,

$$\sum_{ij} f_e^{(ij)} \le c_e \qquad \text{for all } e \in E \qquad (2)$$

where E is the set of edges, c_e the capacity of edge e, and D_{ij} the demand for flow between vertices i and j. $f_{k\ell}^{(ij)}$ is the flow of commodity ij flowing on edge $k\ell$ from vertex k to vertex ℓ. If there is no such edge the corresponding flow is always zero. Also, $f_e^{(ij)} = f_{k\ell}^{(ij)} + f_{k\ell}^{(ij)}$ for e the edge between k and ℓ. We assume here that the constraints for non-negativity of individual flows always hold for a general LP formulation and do not state them explicitly.

Complexity. There are $n\binom{n}{2}$ constraints of the type (1) and $|E|$ constraints of the type (2). The number of variables is $2|E|\binom{n}{2} + 1$. The ellipsoid algorithm of Khachiyan [PS82] then assures us that the MCFP is solvable by a polynomially bounded algorithm, but the size of this LP formulation renders standard LP techniques practically ineffective on all but relatively small problems. Even if we reformulate this multiple commodity flow interpretation into n "single-source, multiple-destination" type flows [He84], we still have the formidable size of $O(n^2)$ constraints and $O(n|E|)$ variables.

Edge Path Formulation of the MCFP. Let P_{ij} denote the set of all paths between vertices i and j. For $p \in P_{ij}$ let f_p denote the amount of flow being hosted along the path p, and D_{ij} the demand between vertices i and j. A maximum concurrent flow function $f\colon \bigcup_{ij} P_{ij} \rightarrow \text{Reals}^+$ of throughput z must satisfy

$$\sum_{p \in P_{ij}} f_p - D_{ij}z = 0 \qquad \text{for all } 1 \le i < j \le n. \qquad (3)$$

Let P_e denote the set of all paths containing the edge e with c_e the capacity of e. Then any feasible concurrent flow must satisfy the following capacity constraints:

$$\sum_{p \in P_e} f_p - c_e \le 0 \quad \text{for all } e \in E. \qquad (4)$$

The objective as before is to maximize the concurrent flow throughput z.

Although this formulation is implicitly exponential in the number of variables, an LP solution exists for this edge-path formulation where an appropriate choice of $\binom{n}{2} + |E| - 1$ of the $\sum_{ij} |P_{ij}|$ paths is sufficent to achieve optimality. Note that this implies that some optimal solution employs flow on an average of less than two paths from each P_{ij}.

Our iterative algorithms approximate a solution through the use of heuristics which reduce the gap between the current solution and the optimal solution employing flows on paths as in this formulation. They do so in a manner such that only a reasonable number of paths are required to be explored. The concept of deviating flow [FG73] for telecommunication routing in store-and-forward packet-switched networks is one similar instance of such an approach. In the LP network literature generally, the edge path formulation has not been pursued to a great extent.

3. Viewing the MCFP as a search problem.

According to Rich [Ri83], a search problem can be analyzed along several key dimensions that dictate the choice of an appropriate efficiency criteria or heuristic. In this section, we categorize the MCFP along some of the dimensions considered by

Rich. This discussion, we hope, acts as a motivation for the algorithms to follow in Sections 4 and 5.

It is clear that we are dealing with a predictable universe in the sense used by Rich. States of the system are completely specified in terms of the node and arc capacities and their utilizations at a given instant of execution. Constraints are therefore static and the LP formulation completely specifies all possible constraints.

The problem is not decomposable. However, decompositions into minimally interacting components is possible. This is evident from the fact that each node must receive (and send) flow from (/to) every other node. Even if we decompose the problem into several instances of a "single-source, multiple-destination" type problem, we still have a decomposition where the components interact quite heavily. In particular, addition of flow of some commodity along an edge reduces the capacity of that edge for other commodities by a certain amount, thereby affecting all other paths that contend for use of that edge. In spite of this interaction our approach will be to decompose the graph with heuristic labelling. We attempt by suitable heuristics to keep the effects of interaction appropriately controlled. The net effect will be to improve time and space requirements for solving the problem as compared with alternative LP techniques.

If decomposition is our strategy then it follows that optimization is restricted to being local and we must be prepared to undo solution steps. This does not necessarily imply backtracking. The first of the two algorithms utilizes greedy flow augmentation and does not have a flow rerouting feature. Though it works very well in many of the cases, its performance can be greatly affected by certain aspects of the network topology. We show (in our second algorithm) how *bad* flow allocations in previous steps of the algorithm may be *undone* by appropriate flow rerouting in future steps.

Is a good solution obvious? If *good* is interpreted only as an exact optimal solution then the answer is no. If tolerances are acceptable, solutions can be efficiently developed with an indication of their maximum deviation from a provable bound on the maximum concurrent flow throughput. In practice, it turns out for our first algorithm that sometimes the upper bound is much higher than the maximum throughput value, but for our second algorithm the achieved throughputs and derived upper bounds on maximum throughput are convergent.

What is the role of knowledge? Not a great deal of explicit knowledge (in the sense used by Rich) is required to solve the problem. Demands between the vertex pairs, capacities of arcs, and the configuration of the graph are all that is needed for the solution. However, there is a lot of implicit knowledge that depends upon the structure of the graph. If this knowledge can be used to formulate heuristics, the algorithm can be guided much more effectively. A very simple heuristic is the degree of each vertex. If a vertex is of low degree we would not like to have many paths carry flow into and out of it because this will saturate the corresponding edges too rapidly. Our first algorithm (which does not undo) suffers from not necessarily recognizing this constraint. Before presenting the algorithms, we shall develop results concerning problem simplification and concurrent flow bounds.

4. Problem Simplification and Convergence Bounds

We shall henceforth investigate the MCFP only for the case where all demands and capacities are unity, as the extension to variable demands and capacities is then reasonably straightforward. The following lemma shows we can further simplify the problem to consider only demands and paths between non-adjacent vertices of the graph.

Lemma 1: For any graph $G = (V,E)$, if z' is the maximum throughput of concurrent flow between all non-adjacent vertex pairs and z is the maximum throughput of concurrent flow between all pairs of vertices, then $z = \frac{z'}{1+z'}$.

Proof. Let e be an edge between two vertices v_i and v_j. If some particular optimal solution to the all pairs MCFP has any flow from v_i to v_j not passing through the edge e, we can always swap this flow with an equivalent amount of flow passing through e and thereby restrict all flow between end vertices v_i and v_j to e alone. Since the optimal solution in the all pairs case is z, this will bring down the available capacity for non-adjacent flow along e to (1-z). The solution in the non-adjacent pairs case with all capacities (1-z) is then z'(1-z). But this is the same as z, since we have already allocated z units of flow between adjacent pairs. Thus $z = z'(1-z)$, from which the result follows. □

Our algorithms shall each identify a feasible concurrent flow, where the throughput is then implicitly a lower bound on the maximum throughput of concurrent flow. To provide a performance guarantee for an approximate solution, or perhaps confirm an optimal solution, we shall also compute certain upper bounds on the maximum concurrent flow [Ma85]. Noting that each of $|A||\bar{A}|$ vertex pairs must have z units of flow crossing the cut $(A,\bar{A})$ of capacity $|(A,\bar{A})|$ for any cut $(A,\bar{A})$, we obtain the following.

Lemma 2: The maximum concurrent flow throughput z for any graph G satisfies

$$z \le \min_{(A,\bar{A})} \frac{|(A,\bar{A})|}{|A||\bar{A}|} . \qquad (5)$$

Equality of a feasible z with the ratio $|(A,\bar{A})|/|A||\bar{A}|$ for some cut $(A,\bar{A})$ is sufficient to confirm optimality of the derived throughput, but not all graphs yield equality for (5) [Ma85]. Identification of a cut $(A,\bar{A})$ for which the ratio $|(A,A)|/|A||\bar{A}|$ is anticipated to be relatively small is incorporated in our greedy flow augmentation algorithm and will be seen to confirm an optimal solution for a number of test graphs.

A more general bound based on the principle underlying our flow rerouting algorithm is first motivated by a dual problem. Let us say that the graph G is actually a network of thoroughfares and that there is some authority placing tolls on each edge of G. The objective of this authority is to increase total revenue, however, regulations require a specified average toll on the edges. Obviously, if there is above average flow on a given edge with above average toll, it means increased toll collections for the authority. From the users' point of view, the objective is to minimize the toll paid. The user will therefore attempt to take the minimum cost (shortest) path to the destination given any set of toll specifications. These two objectives need to be reconciled by altering tolls until equilibrium is achieved. Our flow rerouting algorithm determines the tolls dynamically by a specified function associating larger edge tolls with larger existing flows through a given edge. This serves the dual purpose of making certain paths less attractive to the user and the total revenue more attractive to the authority. Minimum cost paths are determined and flows rerouted away from more expensive paths, followed by toll revisions, until a balance within tolerance is achieved.

Let T_{ij} denote the toll on the minimum cost path between vertices i and j given specified edge tolls T_e for all $e \in E$. For z units of flow between each vertex pair, the toll paid is at least z T_{ij} for each pair ij. Thus the total toll paid for z units of traffic throughput is at least $z \sum_{i,j} T_{ij}$. Note that T_e is the total revenue from edge e if it is saturated to unit capacity. So the maximum revenue received is at most $\sum_{e\in E} T_e$.

Thus $z \sum_{i,j} T_{ij} \le \sum_{e\in E} T_e$ which gives us an upper bound for any concurrent flow throughput,

$$z \le \frac{\sum_{e\in E} T_e}{\sum_{i,j} T_{ij}} . \qquad (6)$$

More formally, for the graph G let $d{:}E(G) \to R^+$ denote a non-negative valued distance function on the edges of G with d(i,j) the shortest distance between vertices i, j. We further assume the distance function satisfies the triangle inequality, so that d(i,j) = d(e) for e an edge between i and j, and also that d is not zero on all edges.

Theorem 3: Any concurrent flow throughput for the non trivial connected graph G satisfies

$$z \le \min_{d} \frac{\sum_{e\in E} d(e)}{\sum_{i<j} d(i,j)} . \qquad (7)$$

where the minimum is over all distance functions d.

It has been noted that Theorem 3 extends to yield an equality for the maximum concurrent flow throughput [Ma85], providing the basis for our flow rerouting algorithm to obtain an arbitrarily good approximation, but we do not prove that result here. Two corollaries are of immediate interest.

Corollary 3.1: Let $\{d_{ij}\}$ be the distance matrix for the connected graph G (corresponding to d(e) = 1 for all $e \in E$). Then the maximum concurrent flow throughput $\hat{z}$ satisfies

$$\hat{z} \le |E| / \sum_{i<j} d_{ij} . \qquad (8)$$

Corollary 3.2: The maximum concurrent flow throughput z for any connected $n \ge 2$ vertex graph G satisfies

$$\hat{z} \le |E|/[2\binom{n}{2} - |E|] . \qquad (9)$$

Our preceding Lemma 2 is also noted to follow as a corollary of Theorem 3 simply by taking d(e) = 1 for $e \in (A,\bar{A})$, and d(e) = 0 for e not in the cut $(A,\bar{A})$, providing great utility to the result of Theorem 3.

5. The Flow Augmentation Algorithm

The flow augmentation procedure is a greedy heuristic yielding a feasible concurrent flow and a saturated cut whose density provides an upper bound on the maximum concurrent flow throughput. The central idea for each cycle is to determine a set of shortest paths, one between each vertex pair, and to augment the concurrent flow equally on all these paths until at least one more edge is saturated. All additional saturated edges are removed and if the resulting graph is still connected, the residual capacities in the remaining edges are determined and the cycle repeated. When the graph becomes disconnected, a cut is determined and its density computed as an upper bound on concurrent flow throughput according to Lemma 2.

Algorithm Augment: Concurrent Flow Augmentation

Given a graph G = (V,E), this algorithm determines a feasible concurrent flow and a cut saturated by the concurrent flow whose density is an upper bound on the maximum concurrent flow. When the throughput of concurrent flow equals the cut density, a maximum concurrent flow has been found. When unequal, the concurrent flow throughput may or may not be maximum.

1. Set G_r = G and set residual capacities in all edges of G_r to unity.

2. Generate a shortest distance tree from vertex i spanning all vertices j > i for each $1 \leq i \leq |V(G_r)|$. From these trees identify a shortest path between each pair of non-adjacent vertices forming an augmenting set A of shortest paths of G_r.

3. Augment the concurrent flow by Δz on all paths of A, where Δz is determined by the minimum ratio over all edges, of the residual capacity divided by the number of paths of A through the edge.

4. Determine the residual capacity of each edge by $r(e) \leftarrow r(e) - |A_e|\Delta z$, where $A_e \subset A$ includes those paths of A through e. Then delete edges with residual capacity zero, determining a new G_r.

5. If G_r is connected, return to step 2.

6. Determine the vertex partition $A_1, A_2, \ldots, A_k$ for the components of G_r and determine the upper bound $u = \min\limits_{1 \leq i \leq k} \dfrac{|(A_i, V-A_i)|}{|A_i||V-A_i|}$.

7. Rescale the concurrent flow to the all pairs case as per Lemma 1. □

Step 2 is critical in determining the complexity of Algorithm Augment. Utilizing breath first search the shortest distance trees of step 2 can be found in time $O(|V||E|)$. As we are only guaranteed saturation of a single edge in each cycle, an overall time bound of $O(|V||E|^2)$ is obtained. Note that the vertex partition of step 6 can generally be expected to yield just two components. When k>2, all cuts between the k components could be investigated to determine which has minimum density to tighten the upper bound. However, such a computation would require testing up to 2^{k-1} cuts and still could result in an upper bound much greater than the achieved concurrent flow, so our algorithm suggests checking only the partitions A_i, $V-A_i$ to preserve the polynomial boundedness of the procedure. Clearly, considerable savings could be obtained by generating and saving (in step 2) spanning trees that are only a slight variation of preceding trees, or by initially computing and saving a directed acyclic graph (DAG) from each vertex. We have chosen to implement only a simple straightforward version of Algorithm Augment for comparison purposes, and as a suggested first phase for more sophisticated algorithms whose convergence, at least within tolerance, is guaranteed.

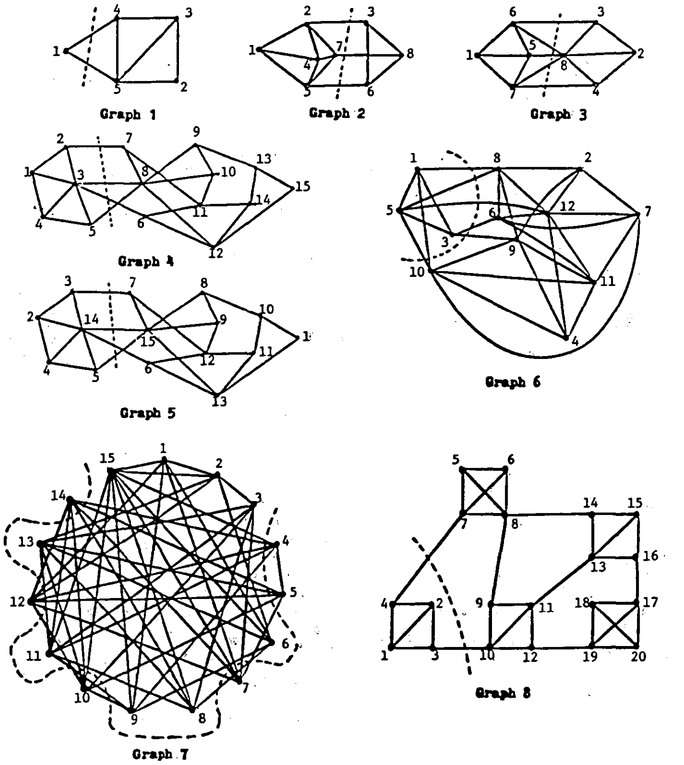


Figure 2: Eight graphs with sparsest cuts illustrated for each graph.

Eight example graphs are illustrated in Figure 2. The graphs have cuts illustrated whose density corresponds to the actual maximum concurrent flow throughput for the corresponding graph. The results of applying Algorithm Augment to these graphs are summarized in Table 1.

Table 1

Graph	|V|	|E|	Number Algorithm Cycles	Number Saturated Edges	Concurrent Flow Throughtput		Computed Cut Density Upper Bound on Throughput
					First Cycle	Final	
1	5	7	3	7	0.3333	0.5	$\frac{2}{4}$ = .5
2	8	14	4	4	0.1428	0.2	$\frac{3}{15}$ = .2
3	8	15	5	7	0.25	0.3125	$\frac{5}{16}$ = .3125
4	15	26	6	6	0.05	0.0761	$\frac{4}{50}$ = .08
5 (isomorphic to graph 4)	15	26	7	7	0.0666	0.0793	$\frac{4}{50}$ = .08
6	12	31	3	10	0.1666	0.2121	$\frac{4}{11}$ = .3636
7	15	49	24	25	0.125	0.2432	-
8	20	34	4	5	0.0222	0.0293	$\frac{3}{96}$ = .03125

Algorithm Augment finds a concurrent flow throughput which is confirmed to be maximum by the derived sparse cut bound for Graphs 1, 2 and 3. For Graphs 4, 5 and 8 the derived throughput of concurrent flow is only slightly less than the sparse cut bound determined, where in both cases the cut is the actual constraint on the maximum concurrent flow as verified by other methods. Note that a shortest path, such as 8, 3, 6 in Graph 4, can zigzag across the constraining sparse cut, and if chosen, an irreconcilable difference between the throughput achieved and the sparse cut bound will be obtained. The algorithm was applied to Graph 4 a second time after relabeling the vertices to obtain the isomorphic Graph 5. The results confirm, as expected, that selecting a different set of shortest paths in some cycle of Algorithm Augment can yield a different achieved concurrent flow throughput.

Graph 6 illustrates a wide disparity between throughput and the sparse cut bound. In this case the algorithm terminated by isolating vertex 3 yielding a bound of $\frac{4}{11}$ = .3636 rather than by separating the vertex set A = {1,3,5} for an achievable sparser cut upper bound of $\frac{7}{27}$ = .2592. The derived throughput is then actually 82% of the best possible throughput and considerably better than the obtained upper bound would indicate. For Graph 7 the bound of Corollary 3.2 indicates no more than $\frac{49}{161}$ = 0.3043 throughput is achievable. Further inspection of Graph 7 revealed the sparse cut illustrated in Figure 2, which was not found by Algorithm Augment. The cut illustrated yields an upper bound of $\frac{15}{54}$ = 0.2777 on throughput, so computed throughput was over 87% of optimum. It can be anticipated that the relatively slow convergence of Algorithm Augment illustrated for Graph 7 will prevail on such "near random" graphs, and other methods should be utilized for this class of graphs. Certainly the fact that all shortest paths have length one or two can be exploited by an alternative algorithm for the class of graphs of diameter two.

In summary, Algorithm Augment can be expected to generally determine the maximum concurrent flow when there exists a sufficiently sparse constraining cut with no shortest paths zigzaging across the cut. The main criticism of Augment is that the achieved throughput and derived bound can differ widely. The greedy method illustrated by Algorithm Augment is available to provide a quick and efficiently determined initial concurrent flow for input to other methods. The effectiveness is indicated, in particular, as the initial cycles alone determined a flow of between 45% and 85% of the optimum for the cases tested.

6. The Flow Rerouting Algorithm

The methodology of this algorithm is to first establish a unit of flow on some path between each vertex pair without regard to capacity constraints on edges. A "toll" is assigned to each edge by a specified function ascribing relatively large tolls to edges with relatively greater total flow. Iteratively, minimum cost paths are found and flows rescheduled off more costly paths with edge tolls dynamically updated. We prescribe a toll function that assigns considerably higher cost to an edge having only slightly higher flow than any other edge. As equilibrium is approached, the flow in those "critical" edges that must in theory be saturated by any maximum concurrent flow [Ma85] approach approximately the same flow (to avoid highly unbalanced costs). Normalization by dividing by the maximum flow observed in any edge yields a feasible concurrent flow where the critical edges should then each be nearly saturated. By appropriate parameter choice for our toll function and employing Theorem 3, it can be shown that a solution with throughput in any arbitrary tolerance of optimality can be determined.

Algorithm Reroute: Concurrent Flow Rerouting

Given a graph G, tolerance ε, and parameter c, this algorithm determines a concurrent flow and a set of edge tariffs such that (for sufficiently large c) the throughput of concurrent flow is within the specified tolerance of an upper bound on the maximum concurrent flow throughput, utilizing the tariff bound of Theorem 3.

1. Assuming edge tolls of unity, allocate a flow of 1 unit on one shortest path $p \in P_{ij}$ between every pair of vertices in the graph, disregarding capacity constraints. Set the flow through each edge, f(e), equal to the number of paths using that edge.

2. Determine the toll placed on each edge $e \in E(G)$ by the following non-linear function:

$$T_e = 2^{c(f(e)-f^*)} \qquad \text{where } f^* = \max_e f(e) \text{ .}$$

3. Proceeding in turn through each pair of vertices $1 \leq i < j \leq |V(G)|$ determine a minimum cost path $p' \in P_{ij}$. If a highest cost path p currently carrying flow from i to j has cost larger than that of p', reroute δ units of flow from p to p', dynamically updating the tolls on the edges affected as per the function of step 2. Determine δ so that the resulting costs as per the adjusted tolls for paths p and p' are identical where p has residual flow greater than zero, or otherwise set δ so as to reroute all flow off p and onto p'.

4. Calculate the concurrent flow throughput realized by this solution: $z = 1/\max_e f(e)$ gives the proper normalization. Calculate the upper bound $z_u = \sum_e T_e / \sum_{i<j} T_{ij}$, and if the difference between z and z_u satisfies the tolerance specified by ε, terminate. Otherwise return to step 3.

The choice of the toll function in step 2 is critical to the success of this procedure. Note that when an amount of flow δ is rerouted from p to an edge disjoint path p', the cost of path p is decreased by the factor $2^{c\delta}$ and p' increased by the factor $2^{c\delta}$. Thus the sum of the tolls on all edges, $\sum_e T_e$ strictly decreases and no cycling is possible. However, if the maximum amount of flow rerouted in any pass through all pairs in step 3 falls below some stipulated minimum, convergence should be considered too slow and the process terminated, or reinitiated with a larger value of the parameter c (e.g. c ← 2c).

The choice of c clearly effects the speed of convergence, but caution is necessary. In theory, for any desired tolerance ε, the algorithm can be shown to terminate with convergence for any sufficiently large c, but large values of c can cause erroneous behavior in finite precision computation. Our approach in the results following applying Algorithm Reroute is to utilize several convenient values of c. Further algorithm refinement incorporating internal determination of c with possible updating in a major cycle is under investigation, as is the application of the algorithm to larger graphs and networks of varying capacities with non uniform pairwise flow demands.

The following table summarizes the results of application of Algorithm Reroute to the test graphs of Figure 2. The data of Table 2 is from an implementation of a version of Algorithm Reroute developed by B. Thompson [TM86]. The version differs only slightly in that in step 3, paths p and p' are determined as indicated for each pair i,j, and the pair with largest difference in cost is chosen for flow rerouting in each iteration. In the table the bound z_u from inequality (6) is termed the elongation (elong.) upper bound, and the cut upper bound is also given for reference corresponding to the cuts illustrated in Figure 2. Note that convergence within 1% of maximum throughput is confirmed for Graphs 1-6, and within 2-1/2% for Graphs 7 and 8.

Acknowledgment: We would like to thank Vipin Kumar and R. V. Helgason for numerous discussions during the preparation of this paper, and Byron Thompson for the version and results of Algorithm Reroute given as Table 2.

Graph	$\lvert V \rvert$	$\lvert E \rvert$	constant c	# flow rerouting iterations	paths per pair	feasible z	elong. upper bound	cut upper bound
1	5	7	4	1	1	.5000	.5384	.500
			16	1	1	.5000	.5384	.500
			256	1	1	.5000	.5384	.500
2	8	14	16	2	1	.2000	.2006	.200
			256	2	1	.2000	.2000	.200
3	8	15	16	20	1.32	.3117	.3424	.3125
			256	25	1.42	.3112	.3417	.3125
			16536	21	1.39	.3101	.3249	.3125
4	15	26	16	32	1.19	.0798	.0844	.080
			256	30	1.14	.0799	.0811	.080
5	15	26	16	29	1.09	.0796	.0815	.080
			256	40	1.17	.0799	.0807	.080
6	12	31	16	101	1.53	.2571	.2823	.2592
			256	97	1.48	.2582	.2826	.2592
			16536	41	1.42	.2586	.2826	.2592
7	15	49	16	85	1.42	.2710	.3193	.2777
			256	152	1.72	.2695	.3231	.2777
8	20	34	2	24	1.02	.0307	.0316	.03125
			4	30	1.04	.0309	.0312	.03125

Table 2: Results of application of Algorithm Reroute to the graphs of Figure 1.

References

[At81] Attar, R., A Distributive Adaptive Multi-path Routing -- Consistent and Conflicting Decision Making, '80/'81 Aiken Computation Laboratory, Harvard Univ., Cambridge, MA 02138.

[FG73] Fratta, L., Gerla, M., and Kleinrock, L., The Flow Deviation Method: An Approach to Store-and-Forward Communication Network Design, Networks 3, 1973, 97-113.

[He84] Helgason, R. V., private communication.

[Ma83] Matula, D. W., Cluster Validity by Concurrent Chaining, in Numerical Taxonomy, Felsenstein, J., ed., NATO ASI Series G No. 1, Springer-Verlag, New York, 1983.

[Ma85] Matula, D. W., Concurrent Flow and Concurrent Connectivity in Graphs, in Graph Theory and its Applications to Algorithms and Computer Science, Alavi, Y., et al., ed., John Wiley, New York, 1985, 543-559.

[MD82] Matula, D. W. and Dolev, D., Path-Regular Graphs, Tech. Report 82-CSE-3, Computer Science Dept., Southern Methodist University, May 1982.

[Pa84] Patty, B. W., The Basis Suppression Method for Linear Programs with Special Structure Excluded by an Objective Side Column, Ph.D. Thesis, Dept. of Operations Research and Engineering Management, Southern Methodist University, 1984.

[PS82] Papadimitriou, H., and Stieglitz, K., Combinatorial Optimization: Algorithms and Complexity, Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[Ri83] Rich, E., Artificial Intelligence, McGraw Hill, New York, 1983, Chapter 2.

[TM86] Thompson, B. J. and Matula, D. W., A Flow Rerouting Algorithm for the Maximum Concurrent Flow Problem with Variable Capacities and Demands, and its Application to Cluster Analysis, Tech. Report 86-CSE-12, Computer Science Dept., Southern Methodist University, March 1986.

A LEAST COST PARTITION ALGORITHM

Thomas J. Marlowe, Jr.

Department of Mathematics and Computer Science

Seton Hall University

The least-cost/greatest-profit integer partition problem have a natural formulations by recursive definition, closely related to the knapsack problem. Beginning with the obvious O (n^2) dynamic programming algorithm for least-cost partition, we present modifications illustrating themes in algorithm and data st-ucture design. One results in a lower average-case complexity of order O $(n^{3/2})$, a second in an efficient print algorithm for the resultant partition, and a third is a prepass particularly effective when unit costs are close to monotone. We show worst-case and average-case complexity for the revised algorithm, and argue that the 'real-world complexity' may be even less. We show each of these as a modification of the underlying system of equations or its data-structure representation. Finally, we show that no algorithm with similar primitives can have better worst-case performance.

The breadth of design and analysis concepts illustrated, their direct connection to data structure modification, and the variety of mathematical tools employed, render this problem an excellent example fc demonstrating these principles in instruction or other contexts.

1. THE LEAST-COST PARTITION PROBLEM

1.1 Definitions

Combinatorial objects such as permutations, selections, and partitions give rise to natural formulations in recursive definitions [15, 21]. Attaching cost functions, such as to edges in a graph, and looking for minima over sets of these objects, leads to interesting problems with significant applications [9, 15]. Such problems frequently have a recursive, dynamic programming formulation [2, 13] as an nXn equation set.

A **partition of an integer n** is a multiset of integers whose elements sum to n. Partitions have significant mathematical interest [7, 21], and can be enumerated in a natural way [17]. The **least-cost partition problem**, closely related to the integer knapsack problem, is clearly susceptible to recursive and/or dynamic programming formulation. (Compare [13, 14].)

We modify the definition of partition slightly, to insist on a particular ordering of its elements:

Definition 1.1.1: A **partition** P of an integer n is a non-decreasing finite sequence of positive integers whose sum is n.

Definition 1.1.2: The set of **summands** of P is the multiset of integers appearing in P; the set of **partisands** of P is the underlying set of *distinct* integers. The **height** of P, h(P), is the number of its summands; the **reduced height**, r(P), is the number of its partisands.

Definition 1.1.3: Let **C** be a cost function (a function from the non-negative integers to the non-negative reals, for which **C**(0) = 0). For a partition P, let the cost of P, **S**(P), be $\sum_{j \in P}$ **C**(*j*), and let **S(k)** = min {**S**(P) | P a partition of k} = **S(LCP(k))**, where **LCP(k)** is the *least-cost partition* of k. (Note that this partition is not necessarily unique. We will, however, give a deterministic algorithm for both **S**(n) and **LCP**(n).)

1.2 A Dynamic Programming Formulation

An obvious worst-case Θ (n^2) algorithm exists for finding **S(n)**, by building **S**(k) from the **S**'s of smaller integers:

Definition 1.2.1: Initial Algorithm for least-cost partitions:

$$S(k) = C(1) \qquad , k=1$$

$$= \min \{C(k), \min \{C(i) + S(k-i)\}_{i=1}^{k-1}\} \qquad , k > 1.$$

For such an integer-programming equation-set, the principal solution techniques are dynamic programming and branch-and-bound. Branch-and-bound is usually faster (Chvatal [3] claims it to be 50% faster for integer knapsack problems), requires less storage space, and proceeds from a reasonable initial approximate solution via heuristically directed search. For knapsack-type problems these heuristics frequently employ the list of candidates sorted in unit-cost order. Although the integer knapsack problem is known to be NP-complete (but pseudo-polynomial) [6], there do exist polynomial-time algorithms for various special cases (compare [4]).

For the least-cost partition problem, however, there are several factors which suggest consideration of dynamic programming: First, the greedy algorithm does not necessarily reach a good approximate solution, and early exact solutions found by branch-and-bound may be arbitrarily bad, so that less pruning will occur; second, the corresponding search space may be exponentially large; and

CH2345-7/86/0000/0637$01.00 © 1986 IEEE

third, dynamic programming lends itself to incremental and parallel application, in particular, the return of least-cost partitions of several distinct integers at essentially no additional cost. This algorithm provides a solution to one special case of the integer knapsack problem for which no good approximate algorithm exists, and can be modified to give an O (p^3) algorithm (polynomial in the O $(2p)$ length of the input)[1] for the problem of least-cost (greatest-profit) partition of n using only the integers {1,2,3, . . . ,p}. [11, 10].

We will, however, use certain features common to branch-and-bound algorithms: We sort the integers in *unit-cost* order, and use a pruning technique to reduce average complexity.

Definition 1.2.2: The **unit cost** of an integer k, relative to a cost function **C**, is given by

$$U(k) = \frac{C(k)}{k}.$$

We show that the solution for **S(n)**, including the print of the resulting partition, is possible in average time substantially less than O (n^2), in fact, in O $(n^{3/2})$. This improvement depends on the elimination (dynamically) of terms from the individual equations. We will show that, on average, fewer than $k^{1/2}$ terms are considered in the equation for S(k). These considerations motivate the following definition:

Definition 1.2.3: Call an integer j <u>simple</u> with respect to the given cost function **C** if its only least-cost partition is itself.

> **Proposition 1.2.1:** 1 is always *simple*, and if j is not *simple*, then
>
> 1. every integer in **LCP(j)** is *simple*, and
> 2. for some integer i in **LCP(j)**, $i \leq \lfloor \frac{j}{2} \rfloor$,
> 3. for some i in **LCP(j)**, $U(i) \leq U(j)$.

Since 1 is always *simple*, we have the algorithm:

Definition 1.2.4: Least-cost partition algorithm using simplicity.

$$S(k) = C(1) \qquad , k=1$$

$$= \min \{C(k), \min \{C(i) + S(k-i) \mid i \textbf{ simple}\}_{i=1}^{\lfloor \frac{k}{2} \rfloor}\} \qquad , k > 1.$$

The restriction to $i \leq \lfloor \frac{k}{2} \rfloor$ is immediate; the restriction to i **simple** justifies elimination of **C(i)** + **S(j)** whenever **C(i)** is not itself the best choice for expressing i.

It is however clear that every integer could be *simple* for a particular **C** [consider **C(k)** = $k^{1/2}$], so that the worst-case cost of the modified algorithm will still be O (n^2).

On the other hand, introducing a prepass 'PREMARK', we can show that, for **U** a monotone (non-increasing or non-decreasing) function of n, worst-case performance also improves significantly. PREMARK also leads to improved worst-case performance under reasonable assumptions related, but not equivalent, to monotonicity. PREMARK may be viewed as the application of an inexpensive prepass to eliminate a number of equations from the equation set; this idea is discussed more fully in section 3.3.

1.3 Description of the algorithm

OVERVIEW OF THE LCP ALGORITHM

0. Input is n and costs for integers 1 to n.
1. Compute *unit costs.* Sort the integers 1 to n in unit-cost order.
2. Apply PREMARK to eliminate some equations.
3. Apply modified definition; condense results.
4. Print resulting least cost and partition.

1.3.1 The LCP Data Structure

We now give an overall description of the basic LCP partition algorithm to compute **S(n)**. We discuss the algorithm as modifying the entries of an array A[1. .n] of the following data structure:

```
| IND | CST | MRK | SET | NXM | FRS | NUM | NXT | ,
```

Figure 1-1: DATA STRUCTURE FOR PARTITION ALGS

where for 1≤k≤n:

- IND = INDEX is simply k itself,
- A[k].CST (COST) = S[k] when generated,
- MRK = MARK is a flag; 1 = {k is *simple*},
- SET is a flag; 1 = {S[k] has been found},
- NXM = NEXTMARK points to the next largest *(simple)* integer MARKed, or to 0,
- FRS = FIRST is the smallest integer in **LCP(k)**,
- NUM = multiplicity of FIRST in **LCP(k)**,
- and NXT = NEXT = k - FIRST*NUM.

We also need the costs **C[k]** and unit costs **U[k]**; we keep the former in A[k].CST until needed.

1.3.2 Statement of the algorithm

Algorithm 1.3.1: The LCP algorithm:

The algorithm consists of five phases:

1. a SORT of U into an array **USORT** of records (INDEX,UNIT_COST), which SORT is stable and
   (1) worst-case O $(n \log n)$, and
   (2) O (n) if **U** is monotone;
2. an INITIALIZE of the data structure;
3. the PREMARK phase discussed below;
4. the PART phase, to invoke dynamic

[1] Counting bit lengths, input has length O(2p log p + log n); running time and output is O$(p^3 \log p + \log n)$.

programming on the recursive definition;

5. and a phase PRINT, to print the resulting least-cost partition in sorted increasing order.

Figure 1-2: OUTLINE OF THE LCP ALGORITHM

1.3.2.1 The algorithm PREMARK

Given the array **USORT**, we consider the permutation of [1. .n] induced by **USORT**.INDEX. It is clear that the integer j_0 = **USORT**[1].INDEX – the integer with the least unit cost – must be *simple*, and that none of its multiples can be. (The **LCP** of each of those multiples must be precisely a set of j_0's.) In fact, if we proceed through the list **USORT** maintaining an integer LEAST = the least integer found to be *simple* so far (initialized at n+1), then

- the first smaller integer, S, encountered in **USORT** must be *simple* (so MARK it),
- each of its multiples less than LEAST has a known least-cost partition (by S's), so can be SET, and finally
- LEAST now has the value S.

The algorithm continues until LEAST = 1.

Ex. 1.3.1: Use of the PREMARK algorithm:
For instance, if the permutation of [1. .23] induced by **USORT** were

17 21 18 5 9 19 13 15 22 7 6 2
10 19 11 4 14 23 20 1 8 12 3,

then j_0 = 17 would be *simple* (independent of other values of **C**), and 17 would be MARKed. Continuing, neither 21 nor 18 are less than 17, but 5 is, so 5 must be *simple*, (since no other partition could have lower average unit cost), and neither 10 nor 15 could be (further, their least-cost partitions are({5,5} and {5,5,5}. Note that we know that 20 cannot be simple, but cannot know whether its least-cost partition is {5,5,5,5} or perhaps {3,17}. Thus 5 would be MARKed, 10 and 15 (but not 20) SET, and LEAST set equal to 5. Continuing, neither 9, 16, 13, 15, 22, nor 7 is less than LEAST = 5, but 2 < 5, so 2 is *simple*, and 4 is not – 2 is MARKed and 4 is SET. Finally, 1 is encountered and MARKed as *simple*, but this time no other integer is SET.

PREMARK uses specific knowledge to solve certain equations before invocation of the recursive procedure (see section 3.3). We note that **our average-case complexity results do not depend on the use of PREMARK**; it has been introduced to guarantee that a unit-cost function close to monotone (more likely in real-world problem contexts) will produce close-to-optimum (linear-complexity) algorithm behavior.

1.3.2.2 PART, the dynamic programming phase of the LCP algorithm

PART proceeds as for definition 1.2.4 with the following modifications: **(1)** we do not expand integers already partitioned in PREMARK, **(2)** if i can be partitioned non-simply, we do so, (e.g., if **C**(8) = **C**(2) + **S**(6) = S(8), then 8 has two least-cost partitions: {8} and {2}∪**LCP**(6); we use the partition containing 2), **(3)** we use the smallest possible new partisand (these two conditions specify a tie-breaking mechanism, always selecting the leftmost expression of minimal cost in each row of the equations set), **(4)** if an integer i is *simple* (whether found in PREMARK or not), we link i into a linked list of *simple* integers. (Insuring that the scan for *simple* integers at level i depends only on the number of *simple* integers.), **(5)** we modify the listed solution (for use by PRINT) in the manner indicated below.

1.3.2.3 Condensation of equations and the PRINT phase

Ordinarily, to print the solution to a set of equations, we would maintain the solution chosen at each level i, e.g.,

$$\mathbf{S}(11) = \mathbf{C}(1) + \mathbf{S}(10)$$
$$\mathbf{S}(10) = \mathbf{C}(2) + \mathbf{S}(8)$$
$$\cdots$$
$$\mathbf{S}(8) = \mathbf{C}(2) + \mathbf{S}(6)$$
$$\cdots$$
$$\mathbf{S}(6) = \mathbf{C}(6),$$

etc.; from which we could expand

$$\mathbf{S}(11) = \mathbf{C}(1) + \mathbf{S}(10) = \mathbf{C}(1) + \mathbf{C}(2) + \mathbf{S}(8)$$
$$= \mathbf{C}(1) + \mathbf{C}(2) + \mathbf{C}(2) + \mathbf{S}(6) = \mathbf{C}(1) + 2\mathbf{C}(2) + \mathbf{C}(6).$$

However, repetition of partisands is characteristic of the partition problem, and the list of these in order a feature of the algorithm. This allows us to reduce the number of calls to other equations: if the j in

$$\mathbf{S}(i) = \mathbf{C}(j) + \mathbf{S}(i-j)$$

is the same as that in the expansion of **S**(i–j), we condense this information (in the FIRST, NUM, and NEXT fields of the data structure). If

$$\mathbf{S}(i-j) = a\mathbf{C}(j) + \mathbf{S}(r),$$

(where r = (i – j) – a•j), then we store at i the equation

$$\mathbf{S}(i) = (a + 1)\mathbf{C}(j) + \mathbf{S}(r);$$

if the j's are different, we will, of course, not attempt condensation.

Descent through this 'condensed' set of equations will make the PRINT routine function efficiently – its complexity is now determined by the *reduced height*, rather than the full *height*, of the partition. We will see in section 2.3 that PRINT actually does return the partition in sorted increasing order.

The next three sections discuss the complexity and effectiveness of the several phases of this algorithm. Section 2.1 covers the initial phases of the algorithm, through invocation of PREMARK. Section 2.2 shows that the average-case complexity of the LCP algorithm is lowered to O $(n^{3/2})$ from O (n^2) by use of simplicity. Finally, section 2.3 shows that the resulting least-cost partitions can be printed efficiently, and that the least-cost partitions of all integers less than n can be generated and printed with essentially no cost beyond that needed for n alone.

1.3.2.4 Assumptions for average-case analysis

Our average-case analysis assumes that all **n!** permutations of {1,2,. . .,n}, induced by sorting unit costs into increasing order, are equally likely. Although this is not entirely satisfactory, it almost certainly underestimates the actual savings by comparison to other reasonable and feasible models. Further, as we shall see, our analysis very likely further overestimates the number of remaining terms in the k^{th} equation, and we can show that the same average complexity would result from a stochastic process in which each term considered in equation k . qually likely to be chosen. Therefore, average-case savings demonstrated will almost certainly remain valid in most other models, or in likely real-world situations.

2. ELEMENTARY ANALYSIS OF THE LCP ALGORITHM

We assume all arithmetic operations are constant-cost, rather than depending on bit length. (We may also assume all operations are additions and multiplications if costs are integral and we substitute comparison of iC(j) with jC(i) for determination and comparison of unit costs.)

2.1 Analysis of PREMARK

The cost of determining the array of unit costs, $U[k] = \frac{C[k]}{k}$, is clearly $O(n)$; the subsequent sorting cost in creating **USORT** is $O(n \log n)$ (by assumption), and remaining initialization cost is $O(n)$. PREMARK now scans USORT.INDEX, MARKing each subsequent LEAST integer as *simple*, and SETting certain of its multiples. PREMARK will see each integer at most twice (once if SET, and once on the scan), so its complexity is also $O(n)$, and the issue of interest is:

How effective is PREMARK? How many integers have **LCP**'s determined during PREMARK, and how many of those will be *simple*? The probability that a particular integer is SET as non-simple in PREMARK is strongly influenced by the number of its divisors; elementary number theory and the divisibility lattice of the integers play a major role in the analysis of this phase.

If $k = j_0$ = **USORT**[1].INDEX, then k itself will be MARKed and SET, and $\lfloor \frac{n}{k} \rfloor$ integers in total will be SET at that point. With respect to the number of integers MARKed or SET, further application of PREMARK is in effect equivalent to the application of PREMARK to the induced permutation of {1,2,3,...,k−1}. This leads directly to recursive definitions for the average and worst-case number of integers MARKed and SET in PREMARK. Using number theory and the theory of recurrence relations, we find:

Proposition 2.1.1:

1. At least one integer larger than $\lfloor \frac{n}{2} \rfloor$ is SET in PREMARK(n).
2. The average number of integers MARKed as *simple* in PREMARK(n) is $\Theta(\log n)$.
3. The average number of integers SET in PREMARK(n) is $\Theta((\log n)^2)$.
4. In the worst case, only the integer 1 is MARKed as *simple* in PREMARK(n).
5. As few as $\Theta(\log n)$ integers could be SET in PREMARK(n)

Proof:

1. j_0 = **USORT**[1].INDEX is MARKed. If $j_0 \le \lfloor \frac{n}{2} \rfloor$, then at least one multiple kj_0 has $\lfloor \frac{n}{2} \rfloor < kj_0 \le n$.

2. Let PA(n) = the average number of integers found to be *simple* in {1,2,3, . . . ,n}. Then

$$PA(n) = \begin{cases} 1 & , n=1 \\ 1 + \frac{1}{n} \cdot \sum_{k=1}^{n-1} PA(k) & , n > 1. \end{cases}$$

whence, reasonably directly,

$$PA(n) = \sum_{k=1}^{n} \frac{1}{k} = O(\log n).$$

3. Let SA = the average number of integers in {1,2,3, . . . ,n} SET during PREMARK. Then

$$SA(n) = \begin{cases} n & , n=0,1 \\ \frac{1}{n} \cdot \sum_{k=1}^{n} \left(\lfloor \frac{n}{k} \rfloor + SA(k-1) \right) & , n > 1. \end{cases}$$

Exchanging double summations, we have

$$SA(n) = \sum_{k=1}^{n} \frac{d(k)}{k},$$

where d(k) = the number of divisors of k. Another double-summation argument shows that

$$\sum_{k=1}^{n} \lfloor \frac{n}{k} \rfloor = \sum_{k=1}^{n} d(k).$$

Finally, from Hardy and Wright [7], we have that

$$\sum_{i=1}^{n} d(i) = n \log n + O(n),$$

whence the average value of d(i), $1 \le i \le n$, is $\log n$. Replacing d(i) by $\log n$ for each i gives $\sum_{i=1}^{n} \frac{\log i}{i} = \Theta((\log n)^2)$.

4. Clear.

5. Let SW(n) = the fewest integers SET in execution of PREMARK on {1,2,3, . . . ,n}. Then

$$SW(n) = \begin{cases} n & , n=0,1 \\ \min \{ \lfloor \frac{n}{k} \rfloor + SW(k-1) \}_{k=1}^{n} & , n > 1 \end{cases}$$

which is minimized for $k = \lfloor \frac{n}{2} \rfloor + 1$. Iteration gives $SW(n) = \Theta(\log n)$.

Ex. 2.1.1: An example of worst-case behavior for PREMARK:
Consider the **USORT**.INDEX permutation for n = 31:

(16 30 29 28 27 26 25 24 23 22 21 20 19 18 17
8 15 14 13 12 11 10 9 4 7 6 5 2 3 1 31)

for which only 16 8 4 2 1 will be MARKed, and no other integer SET.

2.1.0.1 The work done by PREMARK

The total cost of PREMARK is O $(n \log n)$, since **USORT** is not otherwise needed. But how much work does PREMARK save? That is, how much work would have been required in PART to solve equations solved in PREMARK? Assume that evaluation of a term in an equation, and comparison of its value with the current min value of the equation is O (1); the question then reduces to: How many evaluations of terms are rendered unnecessary by PREMARK?

The previous example shows that the worst case is Θ (n), since the number of evaluations saved is $\sum_{k=1}^{\lfloor \log_2 n \rfloor} 2^k$, which is between n and 2n. But what is saved on average? If the index of the first entry of **USORT** is k, then no searching is required for $k, 2k, 3k, \ldots, \lfloor \frac{n}{k} \rfloor * k$, which, compared to original O (n^2) algorithm accounts for

$$k * \sum_{i=1}^{\lfloor \frac{n}{k} \rfloor} i = k * \lfloor \tfrac{n}{k} \rfloor * (\lfloor \tfrac{n}{k} \rfloor + 1) / 2;$$

together with any saved work from the induced permutation on {1,2,3,...,k−1}. Thus:

Definition 2.1.1: Recursive definition for average work done by PREMARK:

$$WORK\,(n) = \begin{cases} 1 & , n=1 \\ \frac{1}{n} \cdot \sum_{k=1}^{n} \frac{k}{2} * \lfloor \frac{n}{k} \rfloor * (\lfloor \frac{n}{k} \rfloor + 1) + WORK\,(k-1) & , n > 1. \end{cases}$$

Proposition 2.1.2: $\mathrm{WORK(n)} = \sum_{k=1}^{n} d\,(k) = \Theta\,(n \log n)$.

Proof: We first show by calculus of sums and differences that

$$\sum_{k=1}^{n} k * \lfloor \tfrac{n}{k} \rfloor * (\lfloor \tfrac{n}{k} \rfloor + 1) = \sum_{k=1}^{n} 2k * d\,(k).$$

(Clearly equal when n = 1, and manipulation of sums shows Δ(left−hand side) = Δ(right−hand side) for n > 1.)

Induction then establishes that $\mathrm{WORK(n)} = \sum_{k=1}^{n} d\,(k)$, which is O $(n \log n)$ by the previously cited result of Hardy and Wright [7].

Since the average work saved by PREMARK is only Θ $(n \log n)$ even in comparison to the original algorithm, and this is also its worst-case cost complexity, what are the reasons for its introduction?
First, it is more efficient and somewhat helpful in constructing an incremental algorithm [11];
second, as we shall see in section 3.1, PREMARK handles monotone and nearly monotone cost functions well. Its own complexity will frequently be less than O $(n \log n)$ (in fact, O (n) if **USORT** is monotone, when PREMARK itself will be more productive);
and, third, analysis of its properties provides an interesting application of certain techniques for determining average complexity.

Some final comments:

- First, if we are only concerned with the generation of **LCP(n)** for some fixed n, then:
  1. If n has been SET in PREMARK, do not invoke procedure PART; this occurs with probability $\frac{d(n)}{n}$.
  2. If **C** is not an increasing function, then any integer i < n whose cost **C(i)** is greater than or equal to **C(n)** can be eliminated from **U** before sorting.
- Second, if **U(i)** ≤ **U(j)** for $i | j$, then j can be partitioned by i's, and cannot be *simple*; a modification of SORT could eliminate j from **USORT**.

2.2 Analysis of PART

For each i not SET during PREMARK, PART selects the best decomposition of i by examining the partition **C(j) + S(i-j)** for each *simple* integer, j, less than or equal to $\frac{i}{2}$. The complexity of the procedure is therefore determined by the number of integers SET in PREMARK and the sum for all integers not SET of the number of *simple* integers less than or equal to half that integer. We have seen that as few as $\log n$ integers may have been SET even though every integer may in fact be *simple* (the partition of 31 in example 2.1.1). Thus worst-case complexity is still O $((n - \log n) * n)$ = O (n^2).

In the given worst-case example, significant improvement may be realized by using the observation that some partisand in the partition of j must have unit cost no greater than j's (otherwise, j would be *simple*.) We could thus use an alternate set of equations:

$$S\,(j) = \min\,\{C\,(i) + S\,(j-i)\},$$

where the min is taken over all i < j preceding j in **USORT**. However, the use of such an approach in general requires traversal of **USORT** for each i, and is clearly O (n^2), not only in worst-case, but on average. We will in fact show that any algorithm using only the **USORT** permutation and pairwise comparison of partition costs must have worst-case complexity O (n^2) (see Section 3.2).

The interesting consideration, therefore, is the average complexity of the algorithm, which relies on the determination of the average number of *simple* integers. We will average on the (equally-likely) permutations of {1,2,3,...,n} induced by **USORT.INDEX**, and we will assume that every integer which could be *simple* based on that information will be, although this need not in general be the case. (Such *combinatorial averaging* techniques [22] have been used for complexity analysis in other, sharper settings; the bibliography by Slominski [20] cites 70 references.)

Ex. 2.2.1: Simplicity not fully determined by **USORT.INDEX**:
If (2 7 3 6 5) are the first five entries of **USORT.INDEX**, then 2 must be *simple*, 6 (= 3·2) and 5 (= 2 + 3) cannot be, but 3 and 7 might or might not be, depending on the particular values of **C**.

Generalizing, for a given **USORT**.INDEX permutation (independent of the particular cost function C), the integers MARKed in PREMARK must be *simple*, an integer j following one of its partisand sets cannot be, and others are undetermined: they may be *simple* for some cost functions for the permutation, and non-simple for others. In fact, it seems probable that for any integer k not MARKed in the PREMARK phase there is a cost function with the given permutation for which k is not *simple*, though this will not be possible for every subset of such integers.

Ex. 2.2.2: Restrictions on non-simple subset:
Suppose 5 has the **USORT.INDEX** permutation (3 5 4 1 2) and 4 is *simple*. Since any non-simple partition of 5 must involve 3, the only alternatives are 3+2 and 3+1+1. But 3+1+1 is clearly less costly than 3+2, and if **LCP**(5) = 3+1+1 then **LCP**(4) = 3+1. CONTRADICTION. Thus 5 must also be *simple*.

Note that for any permutation 1 must be *simple*, and 2 cannot be undetermined. Note also that if there are at least two *simple* integers the second smallest (i.e., other than 1) must also be identified in PREMARK. Otherwise, essentially any behavior can occur: Given two disjoint subsets A and B of {2,3,...,n}, there is a permutation of [1. .n] so that every integer in A and no integer in B is *simple*.

The heart of the improvement in average-case performance lies, as mentioned earlier, in determination of the expected number of *simple* integers less than a given $\frac{k}{2}$, which in turn depends on the probability that each given integer j will be *simple*. Note that we also assume that "any integer which might be *simple* for the given **USORT** permutation will be". We then need to determine an estimate for the probability that k might be *simple*, i.e., that it has not been preceded by a set of partisands which, possibly with repetitions, sums to k.

In particular, we note:

> **Proposition 2.2.1:** k will not be *simple* if preceded in USORT.INDEX by:
>
> 1. any divisor d of k
> 2. any pair of primes m and n $< k^{1/2}$
> 3. any pair of numbers i and k−i.
>
> **Proof:** (2) Given any pair of relatively prime integers m and n, every integer larger than mn − m − n can be written as a positive integral combination of m and n [7, 19].

We model only the third constraint, which appears to dominate in practice. Let f(m) = the probability that a distinguished element * occurs before both elements of any of m pairs (a_j, b_j) in a random permutation of those 2m+1 elements; and let **Simp**(n) be the probability that n may be *simple* for a random permutation (i.e., that there is some cost function **C** with the given **USORT**.INDEX permutation for which n is *simple*). Then:

$$Simp(n) \le f(\lfloor \frac{n}{2} \rfloor).$$

This follows directly if n is odd; for n even, if $\frac{n}{2}$ occurs before n, then n is clearly not *simple*, so that Simp(n) is even less than it would be if $\frac{n}{2}$ had to match a paired element. Our main theorem then follows from the following combinatorial lemma, related to Chernov's inequality:

> **Lemma 2.2.1:** $f(n) \le \frac{c}{n^{1/2}}$.
>
> **Proof:** See [11].
>
> **Theorem 2.2.1:** The PART algorithm has average complexity between O ($n \log n$) and O ($n^{3/2}$).
>
> **Proof:** The average number of *simple* integers less than i is at least $\log i$, since the PREMARK phase MARKs an average of $\log i$ integers, and is certainly no greater than
>
> $$\sum_{j=1}^{i} Simp(j) < \sum_{j=1}^{i} (\lfloor \frac{2\pi}{j} \rfloor)^{1/2} \approx 2\cdot(\pi i)^{1/2}$$
>
> by the integral test.
>
> Thus the average complexity of the algorithm lies between
>
> $$\sum_{i=1}^{n} \log i \text{ and } \sum_{i=1}^{n} 2\cdot(\pi \lfloor \frac{i}{2} \rfloor)^{1/2},$$
>
> which are O ($n \log n$) and O ($n^{3/2}$) respectively.

Also note that each integer i will be *simple* with probability no less than $\frac{1}{i}$ (since that is the probability that, in the induced permutation on [1. .i], **USORT**[1].INDEX = i). Thus, at least

$$\sum_{i=1}^{k} \frac{1}{i} = O(\log k)$$

integers less than k will be *simple* on average, which would also show a minimum average complexity of O ($n \log n$).

Note that this ignores the contribution of the PREMARK algorithm, but section 2.1 shows that PREMARK will not make a significant improvement on average. Nonetheless, the approximation of Simp(k) by $f(\lfloor \frac{k}{2} \rfloor)$ can be seen numerically to be a large overestimate.

2.3 Analysis of PRINT

We first show that **LCP(n)** will be printed in non-decreasing order. Consider the data structure (Fig. 1−1): If the condensed equation for k is **S(k)** = a**C(j)** + **S(r)**, then A[k].FIRST = j, A[k].NUM = a, and A[k].NEXT = r. Since, for j be an element of **LCP(k)** we must have that **LCP(k)** − {j} is a least-cost partition of k − j, we have:

Proposition 2.3.1: At the end of the LCP algorithm, A[k].FIRST is the smallest integer in **LCP(k)**.

Proof: If A[k].NEXT = 0, then only one integer occurs in **LCP(k)**, and the result is clear. (In particular, this occurs if k is *simple* or if k was SET in PREMARK).

So assume at least two distinct j occur in **LCP(k)**. For each distinct j in **LCP(k)**, j is *simple* and **LCP(k)** − {j} is a least-cost partition of k−j. However, the recursive line of the definition of **LCP** must have been invoked for k, and must have selected some integer less than k (since k is not *simple*), and so must certainly have considered the smallest integer actually occurring in **LCP(k)**. But if the smallest was considered, it must have been selcted, since LCP chooses the smallest at each step.

Corollary 2.3.1.1: **LCP(k)** is generated in non-decreasing order. In particular, all occurrences of a given integer are consecutive, and therefore condensed.

In fact, this demonstrates that:

Corollary 2.3.1.2: Among partitions of n with equal and least cost, the LCP algorithm chooses the first in lexicographic order (when listed in sorted increasing order).

However, any partition of n, given in non-decreasing order, whether a least-cost partition or not, can be printed by the PRINT procedure outlined in time proportional to the number of distinct integers (partisands) therein, i.e., to its reduced height. We therefore have:

Proposition 2.3.2:

1. No partition of n, whether a least-cost partition or not, has reduced height $\geq (2n)^{1/2}$.
2. If S and T are two sub-multisets of **LCP(k)**, disjoint as sets of partisands, then $\Sigma_{s \in S} s \neq \Sigma_{t \in T} t$
3. If a_1, a_2, a_3, and a_4 are distinct elements of **LCP(k)**, then $a_1 - a_2 \neq a_3 - a_4$. In particular, differences between consecutive distinct integers in **LCP(k)** cannot take on the same value more than twice.
4. The reduced height of **LCP(k)** is at most $\log_2(k+1)$. Thus the complexity of PRINT(k) is O (log *k*).

Proof:

1. Let P be a partition of n, in which each of k distinct integers a_i, $1 \leq i \leq k$, is repeated $n_i \geq 1$ times. Then

$$n = \sum_{i=1}^{k} n_i a_i \geq \sum_{i=1}^{k} a_i \geq \sum_{i=1}^{k} i = \frac{k(k+1)}{2}.$$

   But then $k < (2n)^{1/2}$.
2. If the total cost of S ≠ the total cost of T, then the cost of **LCP(k)** could be lowered by replacing the more costly with the other. If total costs are equal, then by Proposition 2.3.1, the one with the smaller least entry would have been chosen.
3. Else $a_1 + a_4 = a_2 + a_3$.
4. Let r(**LCP(k)**) = R.

   Then each of the $2^R - 1$ non-empty subsets of $\{a_i\}_{i=1}^{R}$ must sum to distinct integers, of which $k \geq \sum_{i=1}^{R} a_i$ is the largest, whence $k \geq 2^R - 1$.

   Thus the number of distinct integers in **LCP(k)** is at worst O (log *k*), whence likewise the complexity of PRINT(k). On the other hand, it is clear that

$$C(a) = \begin{cases} 1, a = 2^i & \text{for some } i \\ \infty & \text{, otherwise} \end{cases}$$

   gives **LCP**$(2^r - 1) = 1, 2, 4, 8, \ldots, 2^{r-1}$, which has $R = \log_2(k+1)$.

Corollary 2.3.2.1: Determination and printing of the **LCP**'s of all integers from 1 to n is of worst-case complexity $O(n^2)$, and average complexity at worst $O(n^{3/2})$.

Proof: Determination of **LCP(n)** determines in the process the **LCP** of all smaller integers. Printing these **LCP**'s has complexity

$$\sum_{k=1}^{n} O(\log k) = O(n \log n) < O(n^{3/2}).$$

Thus **LCP(n)**, seen as an algorithm determining and printing the **LCP**'s of all integers between 1 and n, determines n 'pieces of information' and outputs them in no worse than $O(n^2)$ time.

3. FURTHER ANALYSIS OF THE ALGORITHM

3.1 The effect of monotonicity assumptions

When unit costs are monotonic or nearly monotonic, PREMARK becomes important, and the order of complexity of the LCP algorithm is lowered significantly. This occurs most dramatically if strict monotonicity holds, in which case this follows directly:

Proposition 3.1.1: If **U** is monotone increasing, then only 1 is MARKed, and everything is SET in PREMARK, and the complexity is thus O (*n*). If **U** is monotone decreasing, then everything is MARKed in PREMARK, and the complexity is likewise O (*n*).

What if U is monotonic except for essentially 'random' fluctuations? We show most of the savings available from monotonicity still apply in many cases.

Definition 3.1.1: A function f is **nearly monotonic** if there exist positive real constants **a** and **b** so that $f(m) \geq f(n)$ for all $m \geq a \cdot n + b$ (or $f(m) \leq f(n)$) for all such pairs m and n. f is **almost monotonic** if a can be taken as 1. We will use terms such as 'almost increasing' and 'nearly decreasing' in the obvious sense.

Proposition 3.1.2: If U is *almost increasing* with constant c, then

1. no more than the first **c** integers can be *simple*,
2. on average, fewer than $c^{1/2}$ will be,
3. at least $\lfloor \frac{n}{c} \rfloor$. integers, and an average of
$$\frac{1}{c} \cdot \sum_{k=1}^{c} \lfloor \frac{n}{c} \rfloor = O\left(\frac{n \log c}{c}\right),$$
integers will be SET during PREMARK.

Proposition 3.1.3: If U is *almost increasing* with constant **c**, then the average complexity of the entire LCP algorithm is at most
$$O\left(\max\left(n \log n, c^{1/2} \cdot n\left(1 - \frac{\log c}{c}\right)\right)\right).$$

Proposition 3.1.4: If **U** is known to be *almost increasing a priori*, and **c** is known, then, then the complexity of the SORT phase can be reduced from $O(n \log n)$ to $O(c \log c)$. Thus the complexity of the LCP algorithm is $O(nc)$ at worst, and $O(nc^{1/2})$ on average.

The situation for *nearly increasing* functions is almost as good:

Proposition 3.1.5: If U is *nearly increasing* with constants **a** and **b**, then **LCP(n)** has average complexity at most $O(\max(n \log n, (a+b)^{1/2} \cdot n))$

Functions which are essentially decreasing, however, are in most cases not so well-behaved: If **U** is *almost decreasing* with constant c, then $U(m) \leq U(1)$ for all $m > c$, and, for any m, at least one of {m,m+1,m+2,...,m+c−1} must be MARKed in PREMARK. However, the only non-simple integers which could possibly be SET in PREMARK are integers less than 2c.

Proposition 3.1.6: If **U** is *almost decreasing* with constant c then:

1. If PREMARK always leaves at least one integer between m and m + c unMARKed for each m ≥ c, then PART(n) has complexity between $O\left(\left(\frac{n}{c}\right)^2\right)$ and $O\left((n)^2\right)$
2. In such a case, PREMARK saves at least $O\left(\left(\frac{n}{c}\right)^2\right)$ work over the efficient implementation of PART without PREMARK.

Consideration of Example 2.1.1, restricted to n = 30, shows that *nearly decreasing* functions with $a \geq 2$ can exhibit very nearly worst-case behavior. However, if **a** and **b** are known *a priori*, and $a < 2$, then the algorithm suggested for that example at the beginning of section 2.1 will be more efficient than PART; a related algorithm may in fact be more efficient for many cases in which all that is known is that **U** is *nearly decreasing*. The corresponding set of equations formulation for an *almost decreasing* example with known **c** would be:

Definition 3.1.2: Least-cost algorithm for *almost decreasing* functions:

$$\begin{aligned} S(k) &= C(k) && , k=1 \\ &= \min\{\min\{C(i)+S(k-i)\}_{i=1}^{\lfloor \frac{k}{2} \rfloor}, C(k)\} && , 2 \leq k \leq 2c \\ &= \min\{\min\{C(k-i)+S(i)\}_{i=1}^{c}, C(k)\} && , 2c < k. \end{aligned}$$

Proposition 3.1.7: Assume **U** is *almost decreasing* with known constant **c**. Then the algorithm of definition 3.1.2 gives a least-cost partition for each integer k and has worst-case complexity $O(O(\text{PREMARK}) + nc)$.

Two final comments:

- First, the initial phases of the algorithm need not be used, if SORT is not required to discover monotonicity;
- Second, we still need not consider non-simple trial partisands.

Further, there exists an elementary $O(n)$ algorithm for determining whether a sorted list of unit costs is *almost monotone* with an arbitrarily specified parameter **c**.

3.2 Minimum complexity of an LCP algorithm

It is clear that any LCP algorithm will have to examine all of the input at least some of the time, since any one integer, and any pair of integers less than $\frac{n}{2}$, can occur in **LCP(n)** for some cost function **C**, whence any LCP algorithm is at least $O(n)$.

Further general analysis is difficult (since, for example, there exist sorting algorithms in restricted domains more efficient than $O(n \log n)$, if operations are not restricted to key comparison and swapping [9]). We can however show a lower bound on minimum complexity for any algorithm which proceeds by computing and comparing the costs of partitions in order to attain least-cost partitions for integers $1 \leq k \leq n$, even if the array **USORT.INDEX** can also be used.

Ex. 3.2.1: $O(n^2)$ best-case partition.
Let n = 4m + 1, and **USORT.INDEX** =

2m, 2m-1, 2m-2,...,3,2,1,2m+1,2m+2,...,4m,4m+1.

Then each integer from 1 to 2m is *simple*, but only 4m is otherwise SET by a PREMARK-type prepass (actually 4m-1 = 2m + (2m-1) is also known, but this is irrelevant, as we will show that no other **LCP** could be).

$$S(2m+1) = \min\{C(i) + C(2m+1-i)\}_{i=1}^{m},$$

where each of the alternatives must be considered. Suppose the partition selected is (1,2m). Then

$$S(2m+2) = \min\{C(i) + C(2m+2-i)\}_{i=2}^{m+1},$$

and again each of the possibilities must be considered. Again suppose (2,2m) is selected. Continuing in this fashion, we must consider

$$m + m + (m-1) + (m-1) + (m-2) + \dots + 2 + 2 + 1$$

$$= m(m+1) - 1 = O\left(\frac{n^2}{16}\right) \text{ different terms}$$

to determine **LCP**'s of integers 2m+1 through 4m.

But determination of **LCP**(4m+1) requires consideration of each possible decomposition

$$C(j) + S(4m+1-j), \text{ for } 1 \le j \le 2m,$$

so that all these $O(m^2)$ possibilities are considered by the algorithm.

To establish the lower bound, the possibility of a cutoff in the evaluation of the **S**(j) must be ruled out. We can however show:

Proposition 3.2.1: Given any pair of integers i and j with $0 < j \le i \le \lfloor\frac{2m+j}{2}\rfloor$, there is a cost function $\mathbf{C}_{i,j}$ with the **USORT**.INDEX permutation of the example above for which the **LCP** of each 2m+k, k ≠ j, is {k,2m}, and for which **LCP**(2m+j) is {i,2m+j−i}.

Theorem 3.2.1: Let **AP** be an algorithm which determines least-cost partitions for each integer from 1 to n, using only ranking of the costs or unit costs of using integers, and comparison of the cost of partitions with each other or with a current minimum. Then the worst-case complexity of **AP** is at least $\Omega(n^2)$

Proof: For n = 4m+1 fixed, each of the cost functions of proposition 3.2.1 has exactly the same **USORT**.INDEX permutation and the same COST permutation. But if any of the $O\left(\frac{n^2}{16}\right)$ terms of the example is not examined by **AP**, say **C**(i) + **C**(2m+j−1), there is a cost function $\mathbf{C}_{i,j}$ for which that would have yielded the least cost partition of 2m+j.

The example appears also to demonstrate a minimum $O(n^2)$ bound on the complexity of an algorithm which finds the **LCP** of n alone.

3.3 The LCP algorithm and the least-cost partition equation set

The techniques used to achieve reduced complexity in the LCP algorithm of definition 1.2.4 are most easily understood through consideration of the dependency graph and equation set [15] of the least-cost partition problem. The direct implementation of the initial algorithm of definition 1.2.1, instantiates the solution of an equation set with acyclic dependency graph [12, 15]. The savings of the LCP algorithm are threefold, namely:

1. the use of PREMARK,
2. the restriction to simplicity, and
3. the condensation of the equations as created for PRINT;

the use of the data structure (Fig. 1−1) makes possible the incorporation of all three improvements.

3.3.0.1 The use of PREMARK

Consider a system of "linear" equations in the form:

$$Q(k) = (R(i_1)+Q(j_1)) \downarrow (R(i_2)+Q(j_2)) \dots \downarrow C(k),$$

in which all R values are non-negative and whose dependency graph is acyclic. We call such a system an **acyclic min-sum equation set**. We can prune the equation set by dynamically determining the minimum constant and setting the corresponding value equal to that constant. The difficulty lies in the cost of updating information. Note, however, that any term in an equation of this type whose coefficient *R* is larger than the current constant (the updated *C*) of that equation can be discarded.

The work done by PREMARK can be seen, in some sense, as related to the above; it too discards equations based on constant and coefficient terms; it is also related to the minimum constant principle [15] and the solution procedure for triangular matrices/acyclic dependency graphs. However, PREMARK does not attempt to solve the entire system, but merely applies an initial step, with some additional dividends: Not only are the equations in which the initial constant dominates eliminated, but also some of those in which substitution of those initial constants will yield solved equations. PREMARK can thus be viewed as a standard technique for solution of equation sets, slightly extended, or as a technique applicable to min-sum equations, significantly restricted.

The resulting simplification of the equation set, though sometimes of small value, is sometimes very significant, at the cost merely of an inexpensive single $O(n \log n)$ sort of the data, without any updating of numerical values. Both this efficiency, and the extension to the integers SET, appear to depend heavily on domain structure, namely, the divisibility structure of the integers. Further, PREMARK is not quite an instance of the minimum-constant solution technique, relying on *unit cost* rather than *cost*. Integers marked *simple* in PREMARK need not have minimum *cost*, and PREMARK does not involve direct application of a min-op equation solution principle. Rather, the operation used by PREMARK is convex combination:

$$U(i+j) = \frac{i}{i+j}U(i) + \frac{j}{i+j}U(j),$$

and simplification relies on the fact that in **S**(k), the term **C**(i) + **S**(j) has j = k− i.

The utility of similar techniques would appear restricted to situations in which domain knowledge makes such a prepass or solution method as cheap to apply as in the least-cost partition problem, or as effective as we have seen it to be in the case of nearly-monotone unit cost functions. In general, we would like the cost of the prepass to be less than the average cost achievable by the algorithm without prepass, and to save on average an amount of work at least comparable to its cost, or else to be of particular use in situations actually likely to occur; three tests we have seen to be met by PREMARK in the least-cost partition problem.

3.3.0.2 The restriction to simplicity

The restriction of terms of the min operator to *simple* values, on the other hand, corresponds to the elimination of columns (corresponding to non-simple integers) in the equation-set 'coefficient' matrix. This is related to the elimination of terms whose coefficient is dominated by the corresponding constant, but does not appear to follow directly from such a procedure. In effect, it proposes elimination of terms whose eventual value will certainly be dominated by the eventual value of some other term, whose identity and value however may still be uncertain; perhaps it is most satisfactorily viewed as a use of *comparability* [15]. The availability of this refinement appears to depend highly on the nature of the partition problem and on properties of the integers; related refinements would also appear to be strongly conditioned on detailed domain knowledge.

3.3.0.3 Condensation of equations

Finally, condensation for PRINT can be viewed as partial solution of equations set aside for back-substitution; the economy arises from the propagation of these partial results and the assurance that a significant amount of such condensation will occur. Similar complexity savings from partially solving equations, though occurring differently and at a different point in the solution procedure, underlie improvements on Gaussian elimination and Allen-Cocke interval analysis in program data-flow analysis [1, 5, 8, 16, 18]. Although such saving too is domain-dependent, its applicability appears fairly wide, and some consideration of the possibility of such condensation of partial solutions may frequently repay investigation.

3.3.0.4 The data structure of the LCP algorithm

The data structure created can be viewed as a multiply-threaded list, linked not only by the integer-successor relation, but with each k linked to some i [with **LCP**(k) = FIRST•NUM + **LCP**(i)], and each *simple* j linked to the *simple* integers preceding and following it.

The INDEX and COST fields are inherent in any dynamic programming algorithm for a knapsack-type problem, although the INDEX field simply repeats address information. PREMARK initializes some portions of this data structure, and the MARK and SET fields allow for the graceful merge of PREMARK information with information being developed by PART. NXTMK allows access to the *simple list* at a cost proprtional to its length, and is needed for average-case improvement. The FIRST, NUM and NEXT fields are used for condensation and thereafter by PRINT (although any one of them could be computed on the fly from the other two), and their presence is responsible for the improvement in the efficiency of the PRINT routine.

References

[1] F.E. Allen, J. Cocke.
A program data flow analysis procedure.
CACM 19:137-147, 1977.

[2] R. Bellman. *Dynamic programming.*
Princeton University Press, Princeton, N.J., 1957.

[3] V. Chvatal. *Linear programming.*
W.H. Freeman, New York, N.Y., 1983.

[4] V. Chvatal. Hard knapsack problems.
Operations Research 28:1402-1411, 1980.

[5] J. Cocke.
Global common subexpression elimination.
In *Proceedings of ACM SIGPLAN Symposium on Compiler Construction*, pages 20-24. July, 1970.

[6] M. Garey, D. Johnson.
Computers and intractability: a guide to the theory of NP-completeness.
Freeman, San Francisco, Ca., 1979.

[7] G. W. Hardy, E.M. Wright.
An introduction to the theory of numbers.
Clarendon, Oxford, England, 1938.

[8] M.S. Hecht. *Flow analysis of computer programs.*
Elsevier North-Holland, , 1977.

[9] E. Horowitz, S. Sahni.
Fundamentals of computer algorithms.
Computer Science Press, Rockville, Md., 1978.

[10] T. Marlowe. Least-Cost Partition Algorithms II.
in preparation.

[11] T. Marlowe, M.C. Paull.
Least-cost partition algorithms.
Department of Computer Science Technical Report DCS-TR-169, Rutgers University, New Brunswick, N.J., 1986.

[12] T. Marlowe, M.C. Paull, B.G. Ryder.
Applicability of incremental iterative algorithms.
Department of Computer Science Technical Report DCS-TR-159, Rutgers University, New Brunswick, N.J., 1985.

[13] G. Nemhauser.
An introduction to probability theory and its applications.
Wiley, New York, N.Y., 1966.

[14] C.H. Papadimitriou, K. Steiglitz.
Combinatorial optimization: algorithms and complexity.
Prentice-Hall, Englewood Cliffs, N.J., 1982.

[15] M.C. Paull.
Algorithm design: A recursion transformation framework.
Wiley, New York, N.Y., 1986.

[16] M.C. Paull, B.G. Ryder.
A unified model of elimination algorithms.
submitted. 1985.

[17] E.M. Reingold, J. Nievergelt, N. Deo.
Combinatorial algorithms: theory and practice.
Prentice-Hall, Englewood Cliffs, N.J., 1977.

[18] B. G. Ryder. Incremental data flow analysis. In *Conference record of the tenth annual ACM symposium on the principles of programming languages*, pages 167-176. Association for Computing Machinery-SIGPLAN, January, 1982.

[19] W. Sierpinski. *Elementary theory of numbers*. Hafner, New York, N.Y., 1964.

[20] L. Slominski. Probabalistic analysis of combinatorial algorithms: A bibliography with selected annotations. *Computing* 28:257-267, 1982.

[21] A. Tucker. *Applied combinatorics*. Wiley, New York, N.Y., 1980.

[22] H. Wilf. Some examples of combinatorial averaging. *American Mathematical Monthly* 92:250-260, 1985.

I. Example of the execution of algorithm LCP

Ex. I.0.1: Let n = 13, and the input **C** array be
(9, 14, 24, 30, 33, 42, 46.9,
64, 67.5, 72, 75.9, 87.6, 92.3).
Computing the **U(k)** = $\frac{C(k)}{k}$, **U** =
(9, 7, 8, 7.5, 6.6, 7, 6.7,
8, 7.5, 7.2, 6.9, 7.3, 7.1),
and the stable SORT procedure gives **USORT.INDEX** =
(5, 7, 11, 2, 6, 13, 10, 12, 4, 9, 3, 8, 1).

Applying PREMARK:
Initially, LEAST = 14, and j_0 = 5. Thus 5 is MARKed and 10 is SET, and LEAST set to 5.
Then neither 7 nor 11 is less than 5, but 2 is, so s is MARKed and 4 SET, and LEAST set to 2.
Finally, none of 6, 13, 10, 12, 4, 9, 3, or 8 are less than 2, but 1 is, so it is MARKed.
See Fig. I-3 for the appearance of the array A at this point, and Fig. I-4 for the appearance of array A at the end of the algorithm.

Figure I-1: Execution of PREMARK

Ex. I.0.2: (Example I.0.1, continued.)
Stepping through the equation set with PART:

```
S(1) = C(1) = 9.                    (simple by PREMARK)
S(2) = C(2) = 14.                   (simple by PREMARK)
S(3) = min {C(3), C(1) + S(2)}
     = C(1) + S(2) = 23.             (3 is not simple)
S(4) = 2·C(2) = 28.            (not simple by PREMARK)
S(5) = C(5) = 33.                   (simple by PREMARK)
S(6) = min {C(6), C(1) + S(5), C(2) + S(4)}
     = C(1) + S(5) = 42.         (note tie-breaking)
S(7) = min {C(7), C(1) + S(6), C(2) + S(5)}
     = C(7) = 46.9.                      (7 is simple)
S(8) = min {C(8), C(1) + S(7), C(2) + S(6)}
     = C(1) + S(7) = 55.9.               (not simple)
S(9) = min {C(9), C(1) + S(8), C(2) + S(7)}
     = C(2) + S(7) = 60.9.               (not simple)
S(10) = 2·C(5) = 66.           (not simple by PREMARK)
S(11) = min {C(11), C(1) + S(10),
       C(2) + S(9),C(5) + S(6)}
     = C(2) + S(9) = 2C(2) + S(7) = 74.9.
                                         (not simple)
S(12) = min {C(12), C(1) + S(11),
       C(2) + S(10), C(5) + S(7)}
     = C(5) + S(7) = 79.9.               (not simple)
S(13) = min {C(13), C(1) + S(12),
       C(2) + S(11), C(5) + S(8)}
     = C(1) + S(12) = 88.9.              (not simple)
```

Figure I-2: Execution of PART

APPEARANCE OF DATA STRUCTURE

IND	CST	MRK	SET	NXM	FRS	NUM	NXT
1	9	1	1	0	1	1	0
2	14	1	1	-	2	1	0
3	-	0	0	-	-	-	-
4	28	0	1	-	2	2	0
5	33	1	1	-	5	1	0
6	-	0	0	-	-	-	-
7	-	0	0	-	-	-	-
8	-	0	0	-	-	-	-
9	-	0	0	-	-	-	-
10	66	0	0	-	5	2	0
11	-	0	0	-	-	-	-
12	-	0	0	-	-	-	-
13	-	0	0	-	-	-	-

Initial values of fields labelled '-' are not used by algorithm.

Figure I-3: After PREMARK in Fig. I-1

IND	CST	MRK	SET	NXM	FRS	NUM	NXT
1	9	1	1	V	1	1	0
2	14	1	1	V	2	1	0
3	23	0	1	|	1	1	2
4	28	0	1	|	2	2	0
5	33	1	1	V	5	1	0
6	42	0	1	|	1	1	5
7	46.9	1	1	V	7	1	0
8	55.9	0	1	0	1	1	7
9	60.9	0	1	0	2	1	7
10	66	0	1	0	5	2	0
11	74.9	0	1	0	2	2	7
12	79.9	0	1	0	2	1	10
13	88.9	0	1	0	1	1	12

The NXTMK field: 1 points to 2, which points to 5, which points to 7, which points to 0. All other NXTMK fields are insignificant since the corresponding integers are not *simple*. LASTMK, the last *simple* integer seen to date is currently set at 7. Current scans for **LCP** trace the simple list up to 5, but would include 7 if S(14) were to be computed.

Figure I-4: After PART in Fig. I-2

A POLYNOMIAL DETERMINATION OF THE MOST RECENT PROPERTY IN PASCAL-LIKE PROGRAMS

Dieter Armbruster

University of Stuttgart, Institut für Informatik
Azenbergstr. 12, D-7000 STUTTGART 1
West-Germany

0. ABSTRACT.

If a compiler knew which procedures (or functions) of a program fulfill the most recent property it could produce more adequate code. This stands in contrast to the prevailing tacit worst case assumption on the non-most-recent behavior of those procedures being passed as parameters (for all other procedures this property holds trivially). This old and well known phenomenon will attract new attention with the development of new computer architectures - such as RISC.

We present a method which is polynomial in the program length for deciding this property in Wirth-Pascal-like programs by exploiting the inherent restriction concerning formal procedures. It is this restriction that enables us to

1. reduce (polynomially) the most-recent problem to a reachabiltiy problem for certain procedures and
2. solve this reachability problem with the polynomial algorithm which we present.

This polynomial result for such programs is rather unexpected since for programs in slightly less restricted languages like ISO-Pascal the problem is still decidable but with a complexity as bad as P-space complete.

1. INTRODUCTION

As new computer architectures emerge, especially those with reduced but faster instruction sets (RISC [PA85]), the use of knowledge about the run time behavior of procedures (and functions) in block structered languages like Pascal attracts new attention. While for the present comfortable (but slow) microprogrammed machine instructions (e.g. for a subroutine call) it does not seem to be promising to distinguish between most recent (mr) and non-mr or recursive and non-recursive procedures, this situation changes for reduced, uncomfortable (but considerably faster) instruction sets, since these call for very sophisticated code optimizers. This means that a general worst case assumption on the run time behaviour of procedures (i.e. being non-mr resp. recursive) can no longer be afforded.

Now, we give (informal) definitions of the necessary notions: We say that a procedure P is actually mr (i.e. it fulfills the most recent property) iff

- P is declared locally within procedure Q (Q is the static predecessor of P, Q=SP(P)) and
- in the run time stack which consists of the frames (=activation records) of the called but not yet terminated procedures the static pointer of P's frame always refers to the downwards closest - i.e. most recent - frame of Q.

A program is said to be mr iff every (nested) procedure in it is so.

The non-mr-ness of the following Pascal-like program fragment is revealed by looking at P's static chain in the snapshot of the run time stack which was taken after the execution of several recursive calls of Q and during the subsequent call z(), (i.e., Some_Cond was false). If P now referenced some variables local to Q, they would be located in the second frame of Q (from top), not in the most recent one.

```
program PROG;    ...
 procedure Q( z: proc() )
  begin
  procedure P()  begin ... end
  if Some_Cond then Q(P);
  z();
  end
 Q(A);  ...
```

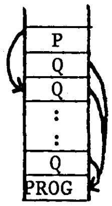


Fig. 1.a: A non-mr program fragment with snapshot of run time stack.

Unfortunately this actual property is in general unsolvable - even for I/O-free programs (because of the ability of such programs to simulate any Turing machine). We thus have to be content with an approximation called formal most recent property which is defined on the formal execution tree of a program schemes [WI82, LA73] rather than on the runtime stack (see below).

In such a scheme (we will call it nevertheless a "program") all data, labels, and the related statements are removed (to provide a better decidability basis) leaving only (arbitrarily nested) procedure declarations with parameters of type procedure and statement parts (body minus local procedure declarations) with zero or more call statements (see Fig.1.b).

If Q = SP(P) (P local in Q) then every call to Q generates a modified copy of the declaration of P by replacing those global formals in the body of P which are in Q's parameter list by the actuals of this call, according to the copy rule of block structured languages. We will distinguish among those copies by appending a superscript s≥0 to P. Now, s can be interpreted as a static pointer to the call of that copy of Q that generated this copy P^s of P.

The static sequence of the call statements within a procedure is made irrelevant by considering them simultaneously, i.e. each call in the statement part of procedure A creates node α_{i+1} on level i+1 as successor of node α_i=A(..) on level i. The label of this successor node is formed by replacing all formals in the call statement by non-formal

CH2345-7/86/0000/0648$01.00 © 1986 IEEE

names of copies of procedures - according to the semantics of Algol or Pascal. Starting this construction with the outermost parameterless procedure α_1=PROG (mainprogram) we end up with a (possibly infinite) formal execution tree E(PROG) of our program(-scheme) PROG:

```
PROG:
{ procedure A()  {  }
  procedure Q(z: proc())
     { procedure P()  {  }
       Q(P)
       z()
     }
  Q(A)
}
```

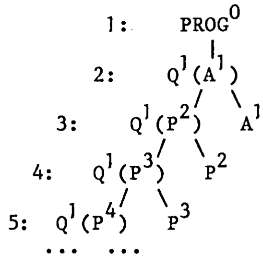
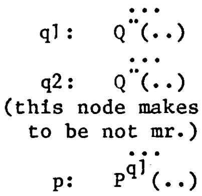


Fig.1b: The non-mr program(-scheme) of Fig.1a and its formal execution tree.

We say further that a procedure P in PROG is formally reachable (recursive) iff there is a branch in E(PROG) which begins at PROG and along which an arbitrary copy $P^{''}$ of P is called at least once (twice). ($^{''}$ denotes a "don't care" static pointer.)

A procedure P formally calls procedure Q iff there exists a subbranch (.., $P^{''}$(..), .., $Q^{''}$(..), ..) in E(PROG).

Now, P is formally mr iff the subbranch between each call to P^{q1} on level p and its static predecessor $Q^{''}$ on level q1 contains no other call to a copy $Q^{''}$. Obviously, P in fig.1b is not mr.



It should be clear by now that formal mr-ness implies the actual but not vice versa (the call that destroys formal mr-ness may never get actually executed!).

By ressorting to this approximation, what do we win as to decidability?

The answer depends on the mode depth MD (type depth) of the procedure names. In fig.1, MD(z)=1, because the call "z()" has no procedure as parameter. If the call would have read "z(P)", then MD(z) would be 2, since MD(P)=1. The specifier of z would then look like "z:proc(proc))". In general:

The mode depth of a formal or non-formal name x which has no parameters when being called is defined to be 1. Then, inductively, the mode depth of a name y which is called as y(..x_i..) is MD(y) = max(MD(x_i))+1, and MD of a program PROG is MD(PROG) = max(MD(P_i)), with P_i being the procedure names.

For finite mode (=FM, e.g. ISO-Pascal [ISO81]) programs MD(P_i) is finite, whereas for FM2 (e.g. Wirth-Pascal) programs MD(P_i) $\leq$ 2.

Note that in FM-programs all formal parameters can therefore be specified completely, as is done, for example, in ISO-Pascal. Thus, a call like P(P) which is legal in an infinite mode allowing language (e.g. Algol60) would be syntactically illegal in a finite mode language, since the formal parameter of P cannot be specified completely (without resorting to recursive types).

We can now give a brief survey of the history of the mr-phenomenon, thereby answering our still open question on the decidability:

In the sixties it was widely (erroneously) believed (at least by Dijkstra) that the static link of an activation record points "to the most recent ... activation of the first block that lexicographically encloses the subroutine" [DI67], i.e., that every block-structured program is mr.

In 1972, McGowan pointed out this "most recent error" and showed the advantages of a mr-based runtime stack organization with all static pointers being redundant. Furthermore he gave a rather coarse sufficient condition for a compiler to determine whether a program is actually mr [MG72].

In [KA74] it was shown that formal mr-ness of a program is decidable - even for infinite mode programs - which is astonishing if contrasted with the negative result for formal reachability of procedures in infinite mode programs [LA73]. (For finite modes formal termination and hence both, formal reachability and recursivity is decidable [DF79,AH85] - though with complexity "complete in deterministic exponential time" [ME85].)

As a consequence, formal mr-ness of a particular procedure P which is based on the reachability of copies of P is also undecidable in infinite mode programs. The positive result for programs is quantified in [WI83] to be unfortunately no better than P-Space complete for both finite and infinite mode programs.

However, we obtain a more encouraging result - namely a polynomial one - as shown in the sequel, if we restrict ourselves to MD $\leq$ 2, e.g. to Wirth-Pascal, where a formal procedure call can no more have actual parameters (remember, the only parameters we deal with are of type procedure) whereas a non-formal call can have formal and non-formal actual parameters - as before.

Now, what is our problem?

If we want to know whether P local to Q is mr we have to check the formal execution tree for a non-mr situation as in the definition. But how deep do we have to search for it? FM2-programs reward us with a pleasant property which we will call "permeability" (section 2), that allows us to call off the search on a branch after having encountered P^q, where q is the level of the first call to a copy of Q on that branch. In section 3 we present a transformation that reduces the question for mr-ness of P to a reachability problem, which is solved in section 4 by an O(n^3) algorithm. (Whenever possible we omit the attribute "formal".)

2. PERMEABLE PROGRAMS

Consider the following figure with MD(EXAMPLE2)=3:

```
EXAMPLE2:
{ procedure Q(x,y: proc(proc(),proc()), z:proc())
    { procedure P()  {  }
      x(P,z)
      Q(y,x,P)
    }
  procedure A(t,u: proc())  { u() }
  procedure B(v,w: proc())  { v() }
  procedure C()  {  }
  Q(A,B,C)
}
```

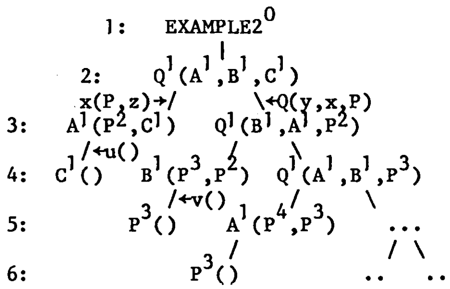


Fig.2: A non-permeable program with MD(EXAMPLE2)=3.

While call node 2: $Q^1(..,C^1)$ has $C^1()$, its third actual, as successor this is not true for 3: $Q^1(...,P^2)$. This possible "constipation" for MD≥3 is the reason for the exorbitant intractability of all interesting calling behavior problems. Therefore we exclude it for permeable programs:

Definition: If the i-th actual of a call to an arbitraray copy of a procedure Q is applied (i.e. either called or passed on as parameter) further down on a branch of the tree, then in a permeable program the i-th actual of any other (possibly different) copy of Q is applied as well.

This – at first glance – exotic and arbitrarily defined property is inherent to all FM2 programs – as can be proven by means of the following lemma which says that if we reach P^t, which has its static link passing via α_q (we write $t{=}{\rightarrow}q$), after a call to an arbitrary copy of Q at level q ($\alpha_q=Q^{\cdot\cdot}(..)$), then we have an analogous situation for any other call $\alpha_r=Q^{\cdot\cdot}(..)$ elsewhere in the whole tree:

Lemma 1: Let P and Q be procedures in program PROG with MD(PROG)≤2. If there exists in E(PROG) a subbranch (omitting parameter lists)

$$B = (\alpha_q=Q^{\cdot\cdot}=A_0{}^{a0}, ..., \alpha_{q+i}=A_i{}^{ai}, ..., \alpha_{q+n}=A_n{}^{an}, \alpha_p=P^{\rightarrow q})$$

with $p=q+(n+1)$, $n\geq 0$, and there exists further a node $\alpha_r=Q^{\cdot\cdot}$, then

(1) there also exists a subbranch

$$B' = (\alpha_r=Q^{\cdot\cdot}=B_0{}^{b0}, ..., \alpha_{r+i}=B_i{}^{bi}, ..., \alpha_{r+n}=B_n{}^{bn}, \alpha_{p'}=P^{\rightarrow r})$$

with $p'=r+(n+1)$,

(2) $A_i = B_i$, i=0..n, and

(3) if $aj{=}{\rightarrow}q$, then $bj{=}{\rightarrow}r$, j=1..n.

Note: The existence of α_p and α_q implies:

q: $Q^{\cdot\cdot}(..) = SP^k(P^{\rightarrow q})$, $k=p-q$, and hence $Q = SP^k(P)$.

The proof is a lengthy induction on the procedures A_1, .., A_n between Q and P, following the construction rules for the tree. (The complete proofs of the lemmata and theorems would not fit into this paper; they should appear elsewhere).

Lemma 1 is used in Theorem 1 and in the next lemma:

Lemma 2: Each program PROG with MD(PROG) ≤ 2 is permeable.

Proof: We are given

$$(\alpha_q=Q^t(..A^a..), ..., \alpha_p = P^s(..), \alpha_{p+1} =..A^a..), p\geq q,$$
and $\alpha_r = Q^{t'}(..B^b..)$ in E(PROG), MD(PROG) ≤ 2.

We show that

$$(\alpha_r=Q^{t'}(..B^b..), .., \alpha_{p'}= P^{s'}(..), \alpha_{p'+1}=..B^b..)$$ exists.

Let w be the formal being replaced by A^a and B^b in α_q and α_p resp. Then α_{p+1} and $\alpha_{p'+1}$ are created by the statement ..w.. in P according the two cases:

- If w is locally formal then P=Q, and w is simply replaced by B^b to build α_{p+1}.
- If w is globally formal then $s{=}{\rightarrow}q$ together with α_r form the hypothesis of lemma 1 giving us $\alpha_{p'+1}$ = ..B^b.. .

We will make extensive use of this lemma in the correctness proof of our algorithm in section 4 (of the complete paper).

Now, let's look back to EXAMPLE2 in fig.2 which shows us that for programs with MD=3 the non-mr-ness of a procedure (here P) is not necessarily revealed by the first call to Q (in fact, here it is the second on level 3, the fourth on level 5, etc. on the subbranches ..–Q–Q–A–P. Of course, by enlarging the rotating parameter list of Q this event could be placed anywhere down the tree).

This messy situation is cleared by the following theorem which assures that in FM2 programs non-mr-ness is detected using the first call to Q as reference:

Theorem 1: Let P and Q be procedures of program PROG with MD(PROG)≤2 and Q=SP(P). P is not mr iff there exists a subbranch in E(PROG)

(1) $B = (..., \alpha_i=A_i{}^{\cdot\cdot}, ..., \alpha_{q1}=Q^{\cdot\cdot}, ..., \alpha_{q2}=Q^{\cdot\cdot}, ..., \alpha_p=P^{q1}, ...)$,

(2) and $A_i\neq Q$, $0\leq i<q1<q2$, A_0 = PROG.

Proof: The right-to-left direction is trivial (definition of mr-ness). For the other direction we only have to show (2) which is an application of lemma 1.

It is this theorem that crunches the intractable (P-space-complete) complexity of the decision problem and makes it managable by means of the following sections.

3. TRANSFORMATIONS

As a result of the previous section we can concentrate on the first call to a copy of Q in, say, $\alpha_q = Q^{\cdot\cdot}(..)$ in each branch during our nondeterministic search for a non-mr situation. From α_q on we look for a second (recursive) call to $Q^{\cdot\cdot}$. It's only after such subsequent calls to $Q^{\cdot\cdot}$ that we are interested in the reachability of P^{q1} – a copy of P the static predecessor of which is α_{q1}.

Now, P is mr iff such a P^{q1} is unreachable.

3.1 Reducing calling to reachability

The key tool to achieve this is a reduction of the problem "does A call B?" to "is B' reachable?", since for reachability we have a polynomial algorithm.

The reduction idea is simple:

After having encountered a call to a copy of procedure A in a branch during a nondeterministic search through the tree we have to change some "state" in order to memorize this event and to be ready for any subsequent call to a copy of B (which would have beeen ignored up to now). To implement this state of change we duplicate each declaration of a procedure Q(..x..) to get the pair: procedure Q1(..x1.., ..x2..) and Q2(..x1.., ..x2..) on the same nesting level as Q. The call ids in the statement parts of Q1 and Q2 receive the suffix 1 and 2 resp., except for A1 which has only call ids with suffix 2. The actual parameterlists get duplicated analogously to the formal ones. Note that the mode depths are not changed by this transformation.

Now, let's follow a branch BR and its transformed BR1, starting at PROG resp. PROG1: On our way to reach the first $A^{\cdot\cdot}$ and $A1^{\cdot\cdot}$ all ids in BR1 have suffix 1. It is directly after $A1^{\cdot\cdot}$ that the suffix switches to 2 and remains 2 for all successors of $A1^{\cdot\cdot}$. So, iff there is a call to $B^{\cdot\cdot}$ in BR1 we must reach $B2^{\cdot\cdot}$. Now the function of the duplicated parameterlists becomes clear: they make the ..2-procedures statically available when needed. This transformation T_A is specified as an attributed grammar in the final paper; here we demonstrate it by applying it to procedure P (i.e. T_P) in a didactical modification of the program in fig.1b:

```
PROG:                           PROG1:
 { procedure A(x)                { procedure A1(x1,x2)
     { A() }                          { A1() }
                                   procedure A2(x2)
                                      { A2() }

   procedure Q(z)                  procedure Q1(z1,z2)
     { procedure P()                  { procedure P1()
          { P(z) }                         { P2(z2) }
                                        procedure P2()
                                           { P2(z2) }

        Q(P)                            Q1(P1,P2)
        z()                             z1()
                                      }
                                   procedure Q2(z2)
                                      { procedure P2()
                                           { P2(z2) }
                                        Q2(P2)
                                        z2()
     }                                }
   Q(A)                            Q1(A1,A2)
 }                               }
```

Fig.3: PROG1 = T_P(PROG)

Note: 1. We realize that within ..2-procedures we do not need any ..1-names since they can never be used. Therefore we simplified the transformation which now yields a worst case (all n procedures in PROG nested in each other) of $O(n^2)$ procedures in PROG1 – as opposed to $O(2^n)$.

2. Sometimes we want to distinguish (for didactical reasons) between procedures bearing the same name (as a result of the transformation). We do this by appending to them – seperated by dots – the names of their surrounding procedures (inside out) until they become unique.

We are now ready to make a theorem out of it:

Theorem 2: Procedure A in program PROG (indirectly) calls procedure B iff
B2 in PROG1 = T_A(PROG) is reachable.

Proof: The proof is a formalization of the mechanism: it takes an arbitrary branch BR and analogously constructs its transformed BR1 by induction on the different cases that may arise: a successor may be called by a nonformal, local formal, or global formal statement. Then, if we are about to construct a node in BR with a call to a copy of B – after having previously constructed a node with a call to a copy of A, we are in BR1 within a ..2-procedure and about to construct the node $B2^{\cdot\cdot}(..)$.

For the other direction, a branch BR1 containing $B2^{\cdot\cdot}(..)$ is transformed back to BR with the inverse transformation and, again by induction, it is shown that BR then must contain both, a call to a copy of A and B.

3.2 Reducing mr-ness to reachability

In order to look for the first occurence of some copy of Q=SP(P), we apply T_Q to the program to be analyzed: If then Q2 is reachable, we know there exists (at least) a second (recursive) call to Q on some branch in the original tree.

Now, we want to know whether this Q2 (indirectly) calls P2.Q1, i.e. SP(P2)=Q1, which would violate the mr-ness of P. This is accomplished by a second application of the transformation with respect to Q2, i.e. T_{Q2}, which takes P2.Q1 into P22.Q11 – if it shows up (the second "2" in the suffix stems from the fact that P2 is called after the suffix switching Q2 whereas the call to Q11=SP(P22) occurs before Q2 which explains the second "1").

It should be clear by now, that and why the following theorem holds (s. fig.5):

Theorem 3: Let P and Q be procedures in PROG, with MD(PROG)≤2 and Q=SP(P).
P is mr iff P22.Q11 in PROG11 = $T_{Q2} \circ T_Q$(PROG) is unreachable.

	q1:		q2:		q3:	p:	p1:	p2:	
	.. Q	.. A	.. Q	.. B	.. Q	.. P^{q1}	.. P^{q2}	.. P^{q3}	..
T_Q:	.. Q1	.. A2	.. Q2	.. B2	.. Q2	.. $P2.Q1^{q1}$	.. $P2.Q2^{q2}$	.. $P2.Q2^{q3}$	..
T_{Q2}:	.. Q11	.. A21	.. Q21	.. B22	.. Q22	.. $P22.Q11^{q1}$	.. $P22.Q21^{q2}$	.. $P22.Q22^{q3}$	..

Fig.5: Transformation of a non-mr situation in an execution tree (to be read from left to right).

4. THE ALGORITHM FOR CALCULATING REACHABILITY

We now need an efficient method to calculate the reachability of a specific procedure, preferably only by inspecting the program text - without constructing the execution tree. Such a method is well known and it computes what is sometimes called the "potentially" reachable procedures. But unfortunately they form only a (less exact) superset of the formal reachable ones [*KL*74, *WA*76].

The method keeps a set of actuals for each formal parameter x, being continuously updated during (in general several) passes through the program: wherever a new actual is found on the position of x it is simply added to the set.

Since no static information is used it is little surprising that this method is less accurate than one operating on the tree. However, the following algorithm is based on the above described method, and yet it yields exactly the formal reachable procedures if applied to FM2-(Wirth-Pascal-)programs (in fact it is only applicable to such programs).

This means: for FM2-programs formal and potential reachability coincide!

Let P be the set of procedures in program PROG, and
X the set of all formal parameters in PROG,

then $I \subseteq P \times X$ is the <u>insertion relation:</u>
A I x <==> A is inserted for x.

$R \subseteq P$ is the <u>reachability relation:</u>
R(A) <==> A is reachable.

```
Step 1: Initialization ($I$ and $R$ viewed as boolean
        arrays): $I$, $R$ := "false"; $R$(PROG) := "true".
Step 2: For all reachable procedures A, i.e. those for
        which $R$(A)="true", do:
  Step 3: For all statements stm in the statementpart of
          A do one of the following three cases depending
          on the form of stm:
    Step 4a: stm = B(..D..), with the nonformal actual D
             on the position of y:
             $R$(B) := D$I$y := "true".
         4b: stm = B(..z..), with the formal actual z on
             the position of y:
             $R$(B) := "true".
             Q$I$y := Q$I$y $\vee$ Q$I$z, for all Q $\in$ $P$.
         4c: stm = y():
             $R$(Q) := $R$(Q) $\vee$ Q$I$y, for all Q $\in$ $P$.
Step 5: If $R$ or $I$ has been changed in step 2 then go back
        to step 2.
Step 6: Halt.
```

Complexity considerations:

If n is the number of procedures in the program (= $|P|$),
s the number of statements of the longest statement part, and
f the number of formal names in the program (= $|X|$),
This yields an overall complexity of $O(s*f^2*n^3)$.

The correctness of this algorithm is stated in our last theorem:

<u>Theorem 4:</u> P is formally reachable iff R(P) holds.

<u>Proof:</u> The left-to-right direction is shown with an induction over a branch containing P''(..). While constructing this branch, we observe the changes that occur to I and R. The other direction uses an induction over the passes through step 4, where R and I are affected.

Now we are ready for our final result combining theorem 3 and 4:

<u>Theorem 5:</u> Let P and Q be procedures of a (FM2-)program PROG, with MD(PROG)≤2, Q=SP(P) and PROG11 = $T_{Q2} \circ T_Q$(PROG). Then
P is not mr iff R_{PROG11}(P22.Q11) holds.

Note: If an original program has n procedures, then PROG11 has at worst $O(n^{2*2})$ procedures. Since the transformation itself is essentially a parsing mechanism with output (the transformation is implemented by means of solely synthesized attributes) it can certainly be done in O(n*f*s). Then R is calculated in $O(s*f^2*n^{4*3})$, i.e. polynomial.

However, if we restrict the nesting depth of procedures (to an arbitrarily high limit), our transformation will then yield only O(n) new procedures; a further restriction on the length of parameter lists and statement parts then allows to compute R in $O(n^3)$.

5. CONCLUSION

We presented a polynomial method that determines whether or not a procedure P behaves most-recently and - applied to all nested procedures - whether the whole program behaves so. Strictly speaking, we were dealing with the weak mr-property; an extension to the strong one (where mr-ness is required not only for the static predecessor but rather for all procedures in the whole static chain) is straightforward but didactically less useful.

We do hope that these results stimulate a reconsideration of an old phenomenon.

6. REFERENCES

[*AR*85] D. Armbruster: On the Decidability of Weak/Strong Recusivity of Procedures in Pascal-like

[*DF*78] W. Damm, E. Fehr: On the Power of Selfapplication and Higher Type Recursion.
In G. Aussiello / C. Böhm Eds., Automata, Languages and Programming, Udine, 177 - 191, 1978.

[*DI*67] E. Dijkstra: Recursive programming, in Programming Systems and Languages, S. Rosen, McGraw-Hill, New York, 1967.

[*ISO*81] ISO/TC79/SC5N: Specification for Computer Language Pascal, Third draft proposal, 1981-11-04.

[*KA*74] P. Kandzia: On the most recent property of Algollike programs. Lecture Notes in Computer Science, Vol. 14, 97 - 111, Springer Verlag, 1974.

[*KL*74] P. Kandzia, H. Langmaack: On a Theorem of McGowan concerning the most recent property of programs.
Fachbereich Informatik, Universität Saarbrücken, 19 S., A74/07, May 1974.

[*LA*73] H. Langmaack: On Correct Procedure Parameter Transmission in Higher Programming Languages.
Acta Informatica 2, 110 - 142, 1973

[*ME*85] A. R. Meyer: Complexity of Program Flow Analysis for Strictness ...
Yet unpublished extended abstract, August 1985.

MODELING AND MEASUREMENT ARENA

Performance Modeling and Measurement

TRACK CHAIR: Dr. Stephen Lavenberg
IBM T. J. Watson Research Center

The State of the Art of Capacity Management in MVS Systems

TRACK CHAIR: Mr. Kenneth Kolence
Kolence Associates

FRAME CACHING IN MENU-DRIVEN VIDEOTEX SYSTEMS

Seetha Lakshmi, Seraphin Calo and Piyush Gupta

IBM T.J. Watson Research Center
Yorktown Heights, N.Y. 10598

Abstract

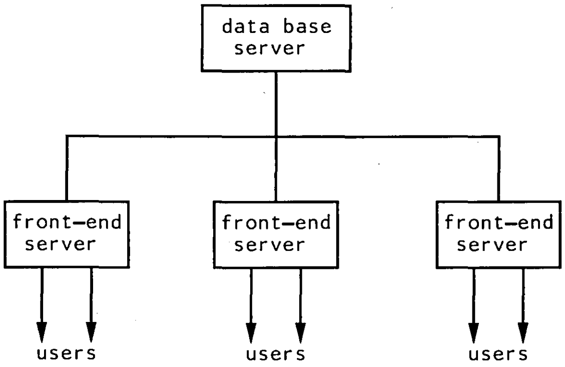
The concept of frame caching in Videotex systems is explored. Caching of frames is expected to improve the response time for retrieval requests. The level of improvement, however, is a function of the cache hit ratio. In this paper, by modelling the user activities on the data base frames, we obtain an analytic expression for cache hit ratio. The analytic solution provides exact results for simple user models, and a tight upper bound for complex models of user behavior. In addition, we use a simulation methodology to study the impact of different parameters on the hit ratio. Our studies reveal that a considerable amount of locality of reference exists among menu-driven videotex users.

1. Introduction

A Videotex system is a medium for delivering information in an effective, user friendly, and relatively inexpensive manner to a large user population. It combines color, graphics, and text to present information in an attractive manner. It provides a unified interface between multiple data bases and a wide variety of user terminals. The interface is simple enough to enable naive users (non-dp professionals) to easily use the services offered. These services range from retrieval services for information such as general news, entertainment listings, financial news, airline schedules, etc., to transactional services such as banking and home shopping.

The back bone of a Videotex system is a powerful data base management system (DBMS). The users access the DBMS through a high level, user friendly interface rather than a traditional data base query language[2,16,18]. An important factor, besides user interface, that must be considered in designing a successful Videotex system is its ability to provide very fast responses to user requests, and to support a large number of users. One of the design strategies that is expected to improve the response time is frame caching.

A Videotex frame is defined as a logical unit of information that can be displayed on a user terminal. A Videotex data base is a collection of frames pertaining to a specific topic. The Videotex system maintains several such data bases. Frame caching is a mechanism by which a subset of the most frequently accessed frames (across all users) is stored in a cache external to the DBMS. For caching to be effective, the cost of retrieving the frame from the cache should be less than the cost of retrieving it from the DBMS.



A Videotex System

Figure 1

Figure 1 represents a conceptual architecture for a Videotex system that supports a large number of users. It consists of a data base server which maintains the Videotex data bases, and a set of front-end communication/distribution servers, which connect the user terminals and the data base server. The front-ends act as terminal concentrators, provide session and terminal management functions, and perform simple user commands which do not require access to the data base server.

Typically, when a front-end receives a frame retrieval command from a user, it performs some preprocessing (to resolve the frame id, data base id, etc., if possible) and then sends a message to the data base server to retrieve the appropriate frame. When frame caching is in effect, frames retrieved from the data base server are retained in a cache maintained by the front-end. Retrieval commands from the users are sent to the data base server only if the required frame is not already present in the cache. Since, for some requests, this scheme avoids the communication delay and processing delay involved in accessing the data base server, it is expected to enhance the average response time perceived by the users. Also, by sending only some of the requests to the data base server, the load on the data base server is reduced. This, in turn, enables the data base server to support more users, thus increasing the overall

CH2345-7/86/0000/0655$01.00 © 1986 IEEE

capacity of the system. The expected performance of a Videotex system with frame caching is evaluated in [7].

The performance improvement is proportional to the cache hit ratio (ratio of the number of requests that found the required frame in the cache to the total number of requests served by the front-end processor). This paper attempts to determine the range of hit ratios that is achievable in real Videotex systems.

In Section 2, we discuss the related work on caching in conventional programming environments and data base transaction processing environments. In Section 3, we describe the access mechanisms prevalent in Videotex systems; we also elaborate on possible organizations of frames in the data base, and the user behavior. In Section 4, we develop an analytical model to evaluate the cache hit ratio. Numerical results based on the analytical model are compared with simulation results. The simulation methodology used to study the impact of various parameters on cache hit ratio is described in Section 5. The results are presented in Section 6.

2. Previous Work

No previous work has been reported on cache studies in Videotex systems. Exhaustive studies of cache memory performance in conventional programming environments can be found in [3,14,15]. Program behavior, which influences the effectiveness of cache memory, has been well studied and understood. It has been established that programs exhibit strong locality, i.e., over a short period of time most accesses tend to reference only a few pages, and these pages constitute a very small subset of the program's total set of pages. Such strong locality enables one to achieve very high hit ratios with very small cache memory sizes. Techniques such as structured programming and loop constructs help achieve this strong locality.

The analog of program behavior in the realm of conventional DBMS is transaction behavior. There has been very little work done in modelling the behavior of data base transactions. Authors [4,9] have studied data reference traces from specific IMS (a hierarchical DBMS) applications and characterized the transaction behavior. An interesting observation made from these studies is that within a single transaction there is very little rereferencing of data; however, data referenced by one transaction is often referenced by other transactions as well. It appears, therefore, that in a DBMS such as IMS, locality is mostly due to inter-transaction rereferencing. Whether inter-transaction locality will improve cache hit ratio will depend, among other things, on the cache management policy, and the scheduling of the transactions. Discussions on alternate cache management schemes for DBMS are found in [10].

3. Videotex Data Bases and Users

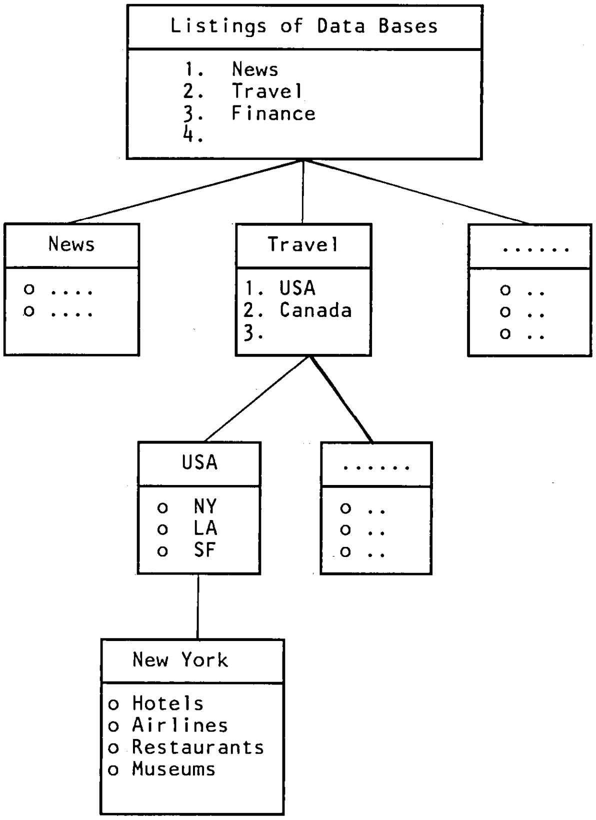
In this section, we characterize the organizational structure of frames and the behavior of users in Videotex systems. Such a characterization enables us to systematically study cache performance.

3.1 Data Base

As mentioned in Section 1, Videotex data bases consist of a collection of frames. The organization of these frames within the data bases and the types of retrieval commands to be supported are interdependent. These are key issues in designing a successful Videotex system. Studies by Videotex researchers are underway to establish the types of retrieval requests that are efficient and powerful, yet easy to learn/use by non-dp professionals [11,17]. Currently, two types of retrieval commands, menu-driven and key word based, are widely supported.

In a menu-driven system, the data base is organized as a tree with one frame at each node. The frames which guide the users are known as *menu frames*. The frames which contain the information sought by the users are called *information frames*. Information frames are located at the leaf nodes of the tree while menu frames are located at the intermediate nodes. Users traverse the tree to view the various frames. They are presented with a series of menu frames which guide them to the information frame of interest. Let us consider an example where a user is interested in some travel related information. The first frame displayed to the user will be a menu frame that resides at the root of the tree. This frame lists the various categories of information available on the system (see Figure 2). Suppose the user selects a menu choice corresponding to travel service. In response, another menu frame containing the information providers logo, categories like domestic travel, foreign travel, etc., will appear on the user's terminal. The user can make further selections based on city, hotel, airlines, etc. Thus, with very few key strokes, the user is guided towards the information of interest.

In a key word based retrieval system, tree traversal by the user is avoided. The users specify key words which succinctly describe the information required. The system searches the whole data base and displays the frames containing the specified key words. While key word based systems may be convenient to use (for experienced users) they have some inherent drawbacks with today's technology. First, a list of key words has to be distributed to the users (perhaps akin to telephone directory distribution). When the list gets frequently updated, distributing most recent versions could become a problem. Second, although it is a simple matter to search the data base for the occurrences of the key words, as the number of data bases and their sizes get larger, the search can consume most of the processor's capacity. A poor choice of key words will also result in wasted processor capacity. On the other hand, the processing requirement in a menu-driven system, which lists only a limited number of selections, will be significantly less. Therefore, while the experienced users may profit from the use of key word



A Tree Structured Data Base

Figure 2

based retrieval commands, inexperienced users may actually achieve a higher level of productivity with menu-driven commands. Efficient methods of structuring the data bases and providing both types of commands are still open research problems[5,13].

In this paper, we study frame caching in the context of menu-driven retrievals. We further limit our attention to tree structured data bases. In reality, the data bases may not have a pure tree structure; there may be multiple links to a given frame. We conjecture that data base structures containing multiple links to a frame will provide better hit ratio than pure tree structures because the probability of several requests fetching the same frame increases. Hence, compared to other structures, our results based on tree structure can be interpreted as a lower bound on cache performance.

In our model, each data base is characterized by two parameters: N, the number of levels in the data base tree and K, the number of choices in each menu frame. The information frames are located at the leaf nodes (level N) and the menu frames are located at the intermediate nodes (levels 1 to $N-1$). The level 0 frame corresponds to the root node which lists the available data bases in the system. Each node representing a menu frame has K children. The parameters N and K completely describe the data base organization. The total number of frames in the system is M.

3.2 User Behavior

Users are assumed to have independent identical behavior, i.e., a user's entry into the system and subsequent frame selections do not depend on the behavior of other users; however, the frame retrieval patterns of all the users are statistically identical. By statistically identical we mean: if several users retrieve a large number of frames, the probability p_{ij} of selecting frame j while viewing frame i (for all i and j) has the same distribution for all users. (It is possible to have models of user behavior where different groups of users have different p_{ij} values. Such models are not considered here.) Users traverse the Videotex data base tree as discussed in Section 3.1. While viewing any of the frames, a user leaves the system with probability p_0 , and views another frame with probability $(1 - p_o)$. The next frame chosen depends on the probability matrix p_{ij}. In Sections 4 and 5 we consider different p_{ij} matrices.

4. Analytical Model

In this section, we develop analytical models to evaluate cache performance. For simple user behavior patterns, the analytical model provides closed form expressions for cache hit ratio. As the user behavior becomes more complex, a closed form expression is no longer possible; instead, it requires an algorithmic evaluation. One of the limitations of the analytical model is that it does not explicitly consider the dynamics of cache management policies. It assumes the existence of a perfect replacement policy that retains the optimal set of frames in the cache. i.e., the replacement policy is considered to have absolute knowledge of future frame references. In reality, replacement decisions are not necessarily optimal since they are based on the past history of user references. Hence, the results from the analytical model should be considered an upper bound on the cache performance. Comparison with simulation models (which capture the dynamics of cache behavior) shows that this is indeed true, and that the analytical model can be confidently used for establishing bounds on cache performance. In the remainder of this section a general formula for hit ratio is obtained and the hit ratios for different user behavior patterns are evaluated. The analytical results are validated through a simulation.

To obtain an analytic expression for cache hit ratio, we model each frame in the data base as a separate service center. Thus, there are M service centers labelled $1, \ldots, M$. Users enter the system according to a Poisson process with rate Λ. Upon entering the system, the users first visit service center i, $(i = 1,2,..,M)$, with probability

p_{oi}. Note that $\sum_{i=1}^{M} p_{oi} = 1$. The residence time at each center is assumed to have a general distribution with a mean of $1/\mu$. The sequence of frames referenced by the users is governed by a first order Markov chain $\{p_{ij}\}$, $1 \leq i,j \leq M$. p_{ij} is the probability that the next frame to be referenced is j, given that the frame being viewed is i. Let p_{i0} denote the probability of a user departing from the system after referencing frame i. The Markov chain whose states are labeled by i is shown in figure 3.

Let λ_i be the aggregate rate at which frame i is requested by the users. The following set of linear equations hold good for the system.

$$\lambda_i = \Lambda\, p_{oi} + \sum_{j=1}^{M} p_{ji}\, \lambda_j \qquad i = 1, \ldots ,M \quad (4.1)$$

If the probability of leaving the system is independent of the current frame being viewed, i.e., $p_{io} = p_o$, for all i, then $\sum_{i=1}^{M} \lambda_i = \Lambda / p_o$. In the following discussions, unless otherwise stated, we assume that $p_{io} = p_o$.

In the Videotex system, it is possible for several users to view the same frame simultaneously. Hence, the occupancy process at each service center is that of a center with infinite number of servers. Such a network of M service centers clearly belongs to the class of product form networks [1]. Let us define the aggregate state of the system by the vector $X = [x_1, x_2, \ldots , x_M]$ where x_i represents the number of references to frame i. Let $P[X]$ denote the steady state probability of the system being in state X. Knowing the rate at which frames are being referenced (λ_i), and the mean viewing time of each frame ($1/\mu$), we can compute $P[X]$ using the closed form expression given in references [1,6]. The equilibrium state probabilities are given by

$$P[X = x_1, x_2, \ldots , x_M] \quad = \quad \prod_{i=1}^{M} e^{-\rho_i} \frac{\rho_i^{x_i}}{x_i!}$$

where $\rho_i = \lambda_i / \mu$

The marginal probability distribution $P[x_i]$ is obtained by summing $P[X]$ over all feasible states with fixed x_i, i.e.,

$P[x_i]$, Prob. that x_i references are made to frame i

$$= \quad \frac{e^{-\rho_i}}{x_i!} \left(\frac{\lambda_i}{\mu}\right)^{x_i}$$

The expected number of references to frame i is given by

$$\sum_{x_i=0}^{\infty} x_i P[x_i] \quad = \quad \rho_i$$

And, the expected number of overall references in the system is given by

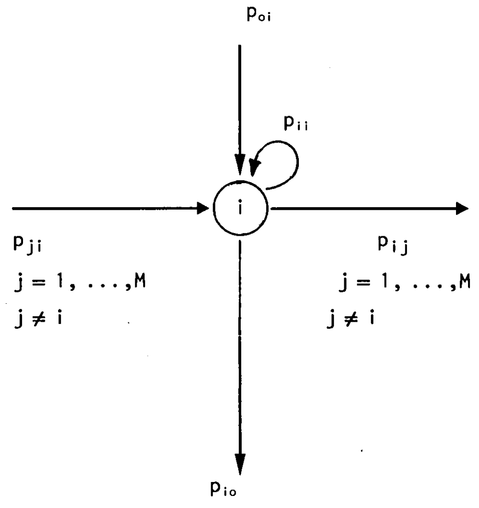


Figure 3

$$\sum_{i=1}^{M} \sum_{x_i=0}^{\infty} x_i P[x_i] \quad = \quad \frac{\rho}{p_o}$$

where $\rho = \Lambda / \mu$

In a cached system, a subset ζ of the M frames is captured in the cache. The subset ζ retained in the cache is determined by the cache management policy. Efficient cache management policies attempt to retain the most frequently accessed frames in the cache. If the next frame referenced is already present in the cache, then a cache hit is said to occur. The steady state probability of cache hit (also known as hit ratio) is defined as the ratio of the number of references to frames in ζ over the total number of references to the system.

Expected value of number of references to frames in ζ

$$= \quad E\left[\sum_{i\in\zeta} \rho_i\right]$$

Expected value of number of references to all M frames

$$= \quad E\left[\sum_{i=1}^{M} \rho_i\right]$$

$$\mathcal{H}\text{, cache hit ratio} \quad = \quad p_o \sum_{i\in\zeta} \lambda_i \,/\, \Lambda \qquad (4.2)$$

Let $|\zeta|$ be the cardinality of the set ζ. Since the cache retains the $|\zeta|$ most frequently referenced frames, the summation in the expression (4.2) is over those frames which have the highest request arrival rate (i.e., the λ_i's are sorted in ascending order, and the first $|\zeta|$ elements are summed).

In the following sections, we compute the cache hit ratio for different user behavior patterns. Each pattern leads to a different Markov chain.

Case 1: Random Entry and Uniform Transitions

Users, upon entering the system, choose to view any of the M frames in the system with equal probability; and after viewing frame i, choose any frame j with equal probability. i.e.,

$$p_{ij} = \begin{cases} 1/M & i = 0,\ j = 1, \ldots, M \\ p_o & j = 0,\ i = 1, \ldots, M \\ (1 - p_o)/M & i,\ j = 1, \ldots, M \end{cases}$$

Substituting these in equation (4.1) the arrival rate for each frame is given by

$$\lambda_i = \frac{\Lambda}{M p_o} \qquad i = 1, \ldots, M \qquad (4.3)$$

Since the request arrival rates are equal, the first $|\zeta|$ frames can be considered to be in the cache. From equations (4.2) and (4.3), the cache hit ratio is derived as

$$\mathcal{H} = p_o\, |\zeta|\, \lambda_i/\Lambda \qquad = \frac{|\zeta|}{M}$$

Case 2: Unique Entry and Uniform Transitions

There is a designated frame f that is viewed by all the users upon entering the system. After viewing frame f, the subsequent frames viewed by the users are as described under case 1. Thus, the arrival rate to each frame is given as

$$\lambda_f = \Lambda + \frac{1 - p_o}{M} \sum_{j=1}^{M} \lambda_j$$

$$\lambda_i = \frac{1 - p_o}{M} \sum_{j=1}^{M} \lambda_j \qquad i = 1, \ldots, M; \quad i \neq f$$

i.e.,

$$\lambda_f = \Lambda \left(1 + \frac{1 - p_o}{p_o M}\right)$$

$$\lambda_i = \frac{(1 - p_o)\, \Lambda}{p_o M} \qquad i = 1, \ldots, M, \quad i \neq f$$

Substituting the above expressions in equations in (4.2), the cache hit ratio is derived as

$$\mathcal{H} = p_o + (1 - p_o) \frac{|\zeta|}{M}$$

When p_o (the probability of leaving the system) is small, case 2 degenerates to case 1 and the hit ratio approaches $|\zeta|/M$. (i.e., unique entry does not have any significance in the long run.) On the other hand, when p_o approaches 1, users reference only a few frames before leaving the system; and these frames are found in the cache. This results in a high hit ratio (approaching 1).

Case 3: Strict Tree Traversal

Here, users enter the Videotex database tree at the root and traverse the tree by selecting the menu choices available at each level in the tree. When they reach the leaf nodes, they return to the root node and traverse the tree again.

For this analysis, a balanced tree with N levels (level 0 corresponds to the root node) and degree K is considered. Let $\{S_\nu\}$ be the set of frames located in the tree at levels 0 through ν. Now,

M, the total number of frames in the system

$$= \frac{K^{N+1} - 1}{K - 1}$$

$|S_\nu|$, the sum of all the frames up to level ν

$$= \frac{K^{\nu+1} - 1}{K - 1}$$

The number of frames cached $|\zeta|$ is such that, $|S_\nu| < |\zeta| \leq |S_{\nu+1}|$ for some νin the range $0 \ldots N - 1$ Let λ_n denote the rate of arrival to any frame at level n in the tree and let $q_o = (1 - p_o)$. Then,

$$\lambda_o = \Lambda + K^N \lambda_N q_o$$

$$\lambda_n = \frac{q_o}{K} \lambda_{n-1} = \left(\frac{q_o}{K}\right)^n \lambda_o \qquad n = 1, \ldots, N$$

Solving for λ_o from the above equations

$$\lambda_o = \frac{\Lambda}{1 - q_o^{N+1}}$$

Let R_ν denote the expected number of references to frames in the set $\{S_\nu\}$.

$$R_\nu = \sum_{n=0}^{\nu} K^n \frac{\lambda_n}{\mu} = \frac{\rho}{p_o} \frac{1 - q_o^{\nu+1}}{1 - q_o^{N+1}}$$

The cache hit ratio is given by

$$\mathcal{H} = \frac{R_\nu + \#\text{ of references to frames in } \{\zeta - S_\nu\}}{R_N} \qquad (4.4)$$

Equation (4.4) will lead to the following expressions, based on the cache size.

$$\text{when } |\zeta| = 1, \qquad \mathcal{H} = \frac{p_o}{1 - q_o^{N+1}}$$

And, when $|S_\nu| < |\zeta| \leq |S_{\nu+1}|$, $\nu = 0, \ldots, N - 1$

$$\mathcal{H} = \frac{p_o}{1 - q_o^{N+1}} \left[\frac{1 - q_o^{\nu+1}}{p_o} + \left(\frac{q_o}{K}\right)^{\nu+1} \left(|\zeta| - |S_\nu| \right) \right] \qquad (4.5)$$

4.1 Validation

In this section, the cache hit ratio is computed using the expression (4.5) and compared with that obtained from a simulation study (described in Section 5). Two different trees are being considered. The users are assumed to perform a strict tree traversal as discussed under Case 3. The probability of leaving the system, p_0 is assumed to be 0.0125. The results are presented in Table I. One can observe, as expected, that the simulation results are always lower than the analytical results.

Table I

Analytic Vs Simulation Comparison of Hit Ratios

cache size	M = 66,430
	K = 9, N = 5
			M = 9331
			K = 6, N = 5	

	anal	sim	anal	sim
2K frames	0.704	0.690	0.84	0.79
4K	0.754	0.760	0.89	0.88
6K	0.804	0.802	0.93	0.93
8K	0.840	0.803	0.97	0.95
10K	0.846	0.804	1.00	1.00

The analytic solution is also used to obtain a general understanding of the effectiveness of frame caching. Table II illustrates the cache hit ratio as a function of cache size for various data base sizes. Interestingly enough, it appears that hit ratios higher than 60% can be achieved with cache sizes which are merely 1% of the data base size, irrespective of the depth or degree of the tree. The actual hit ratio in a real system may, however, be slightly less. These results are corroborated by the simulation study presented in Section 5.

5. The Simulator

While the analytic model developed in the last section can be further extended to model more complex user behavior, it has the limitation of providing only a bound on the performance. In this section, the results from the analytical model are augmented with a simulation study. For this purpose, we constructed a two part simulator. The first part, a reference string generator, generates a sequence of frame ids referenced by the active users. These synthetic frame reference strings are processed by the second part, the cache simulator. While processing the reference strings the cache simulator also collects statistics on the cache performance and displays them at the end of the run. The design of the traffic generator and the cache simulator allows the data base related, user related, and cache related parameters to be easily varied, and their effects observed. We first describe the different parameters that are being varied.

Table II

Cache Hit Ratio for Strict Traversal in a Balanced Tree

cache size
(as % of
db size M)
	K = 3
	N = 10
	M =
	88,573
		K = 5
		N = 7
		M =
		97656
			K = 10
			N = 5
			M =
			11111
				K = 18
				N = 4
				M =
				11151

1 %	0.63	0.64	0.67	0.63
5 %	0.76	0.77	0.75	0.78
10 %	0.82	0.80	0.83	0.81
20 %	0.86	0.88	0.86	0.84
50 %	0.94	0.93	0.91	0.90
90 %	0.98	0.98	0.98	0.98

5.1 DB-Related

It is assumed that there are 9 data bases; i.e., there are 9 choices at level 0. The depth N and the degree K of the tree depend on which of the 9 data bases is selected at level 0. For example, if data base i is chosen, the depth along that path will be N_i and the degree will be K_i. Thus, the Videotex tree used in the simulator need not be a balanced tree. N_i and K_i are randomly chosen (from a uniform distribution between 5 and 9), such that the total number of frames M in the system ranges from 10 thousand to 10 million. From a ***human factors*** view point, traversing more than 9 levels of a tree or choosing among more than 9 choices may not be tolerated by the users, hence, the choice of values for N_i and K_i.

5.2 User-Related

User behavior is characterized by the following probability measures:

p_0 probability of leaving the system

$p_r(l)$ probability of retrace, i.e., probability of backing up to level $l - 1$ while viewing a level l frame $(0 < l < N)$; $p_r(0) = 0$

$p_d(l)$ probability of descending to level $l + 1$ while viewing a level l frame, $(0 \leq l < N)$; $p_d(N) = 0$

$p_b(n)$ probability of backing up to level n while viewing a level N (leaf) frame, $(0 \leq n < N)$

A typical user enters the system through the level 0 frame (the root node) which lists the data bases on available topics. While at level 0, he chooses to view any of the 9 data bases with equal probability. When the user is viewing a level l frame, $(0 < l < N)$, he decides to retrace (back up one level on the tree) with probability $p_r(l)$; or decides to select a menu choice and traverse down the tree with probability $p_d(l)$. When the user is viewing any of the leaf nodes (level N), he backs up to level n, $(0 \leq n < N)$, with probability $p_b(n)$. The probabilities defined above satisfy the following invariants:

$$p_0 + p_r(l) + p_d(l) = 1; \qquad (0 \leq l < N)$$

$$p_0 + \sum_{n=0}^{N-1} p_b(n) = 1$$

The various parameter values considered in the simulation study are listed below:

1. $p_0 = 0$; i.e., the users do not leave the system. In a real system, $p_0 = 0$ implies that when an active user leaves the system, he is immediately replaced by another user entering the system. This is true when the total user population is much larger than the number of active users. The fixed number of active users model has been successfully used in evaluating the performance of time sharing computer systems [8,12]. In the simulation, number of active users = 1, 2, 50, 100.
2. $p_r(l) = 0, 0.1,$ or 0.2. A higher value for $p_r(l)$ implies that the user will be rereferencing a frame more frequently.
3. $p_d(l) = 1 - p_r(l)$, since the invariant has to be maintained.
4. Menu choice selection: Let j be a menu frame at some level l, $(0 \leq l < N)$, containing K choices. If a user decides to traverse down the tree, one of these K choices has to be chosen. Let p_{jk} be the probability of selecting menu choice k,$(1 \leq k \leq K)$. Two different probability distributions for menu choice selection are used in the simulation.

   Case 1: The menu selection is based on a uniform probability distribution, i.e., each menu choice is equally likely to be selected. $p_{jk} = \frac{1}{K}$, for all k.

   Case2: The menu selection is based on a geometric probability distribution. Here, the probability of selecting choice k is less than the probability of selecting choice k-1. With a geometric distribution it is possible to model the fact that some choices have a higher probability of being selected than others. In a geometric distribution with parameter g, $(0 < g \leq 1)$

$$p_{jk} = \begin{cases} g(1-g)^{k-1} & 1 \leq k < K \\ (1-g)^{K-1} & k = K \end{cases}$$

   By assigning different values to g we can model different degrees of locality in frame references. Results for g =0.3 and 0.5 are reported in Section 6.

5. The probability of backing up to level n after viewing a leaf node is assumed to be uniformly distributed. i.e.,

   $p_b(n) = \frac{1}{N}$ for $0 \leq n < N$

5.3 Cache-Related

The following cache related parameters are varied:

1. cache size, also referred to as the number of frames cached (.125K, .25K, .5K, 1K, 2K, 4K, 8K, and 10K)
2. replacement policy (Least Recently Used vs First In First Out)
3. caching policy (any frame can be cached vs only menu frames can be cached)

6. Results

The base parameters for the following experiments are assumed to be

DB size = 100K frames
Number of users = 100
Uniform distribution for menu choice selection
Retrace prob. = 0.1;
Replacement policy = LRU; (menu frames and information frames cached)

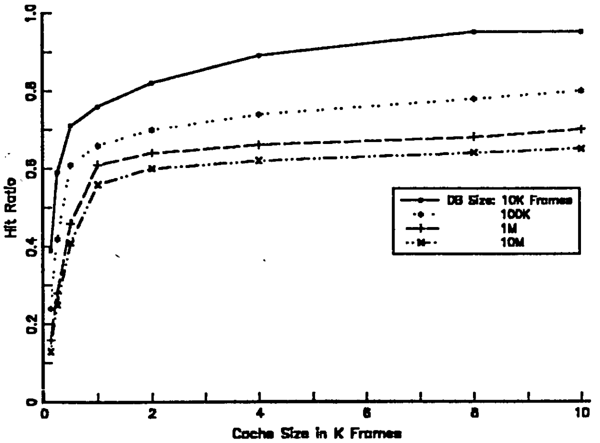
In each experiment, one of these parameters and the cache size are varied, and the effect observed.

6.1 Impact of Data Base Size and Cache Size

In this experiment, the dependence of the hit ratio on data base size (total number of frames in the system) and cache size are studied. The data base size is varied from 10K to 10M frames. For each data base size setting the cache size is varied from 125 to 10K frames. Plots of hit ratio against the cache size, as a function of data base size, are given in Figure 4. These plots show that it is possible to achieve a high hit ratio with a relatively small cache size. In fact, a cache size of 2K frames, which can only hold a very small fraction of the (10M frames) data bases, is able to provide a hit ratio as high as 0.6. This supports the analytical results.

6.2. Impact of Number of Active Users

In order to investigate how the number of concurrently active users affects the cache hit ratio, the cache behavior is studied while the number of users is varied from 1 to 100. The total number of frames in the system is held constant at 100K frames. The menu choice selection probability is assumed to be uniformly distributed; the probability of retrace is assumed to be 0.1. The hit ratios achieved for different cache sizes are listed in Table III. Interesting observations can be made from Table III. The high hit ratios observed in the single user experiments suggest that there is strong intra-user locality of reference. Even when the number of active users in the system increases, the hit ratio remains high. This suggests that there is also considerable inter-user locality of reference.



Hit Ratio vs Cache size and Data Base size

Figure 4

Based on the 10K cache size experiment, one can conclude that a single user is interested in only 10% of the frames available in the data base. Also, when there are multiple users active in the system, 80% of the references (across all users) are for only 10% of the data base. These observations suggest that more detailed studies should be undertaken to establish some rules of thumb for locality of reference in Videotex systems (similar to the 80-20 or 90-10 rules in conventional programming environments).

Table III

Hit Ratio Vs. Active Users

cache size	active users			
	1	2	50	100
.125K frames	0.58	0.58	0.34	0.24
.25K	0.60	0.60	0.55	0.42
.5K	0.63	0.62	0.63	0.61
1K	0.66	0.66	0.61	0.66
2K	0.70	0.70	0.70	0.70
4K	0.74	0.74	0.74	0.74
8K	0.78	0.78	0.78	0.78
10K	1.00	0.80	0.80	0.80

6.3. Impact of Menu Selection Probability Distribution

In this and the next experiment, the effect of changing user related parameters is studied. As discussed in section 5.2, two different probability distributions (uniform and geometric) are considered for menu choice selection. In the geometric distribution, two different values are assigned for the parameter g ($g = 0.5$ and 0.3). The parameter g is the probability of selecting the first choice in the menu frame. A high value for g implies a high degree of commonality of interest among all the users. Clearly, it should be possible to achieve better hit ratios with a geometric distribution rather than a uniform distribution. The simulation results, shown in Table IV strongly support the above conjecture. For a given cache size and data base size, uniform distribution provides the lowest hit ratio while geometric distribution (with g approaching 1) provides hit ratios very close to one.

Table IV

Hit Ratios with Different Distributions of Menu Selection Probability

cache size	Prob. Distribution		
	uniform	geometric	
		g = 0.3	g = 0.5
.125K frames	0.24	0.26	0.34
.25K	0.42	0.44	0.56
.5K	0.61	0.63	0.72
1K	0.66	0.68	0.78
2K	0.70	0.72	0.83
4K	0.74	0.76	0.89
8K	0.78	0.82	0.92
10K	0.80	0.84	0.94

6.4 Impact of Probability of Retrace

In this experiment, $p_r(i)$, the probability of retrace (backing up one level in the tree) is varied from 0 to 0.2, and the hit ratio is observed. This models the situation where the users tend to rereference a frame within a short period of time. The hit ratios achieved for different retrace probabilities and cache sizes are listed in Table V. As expected, when users retrace their path often, higher hit ratios are observed.

Table. V

Impact of Probability of Retrace

cache size	Prob. of Retrace		
	0	0.1	0.2
.125K frames	0.23	0.24	0.28
.25K	0.39	0.42	0.45
.5K	0.59	0.61	0.63
1K	0.65	0.66	0.68
2K	0.69	0.70	0.72
4K	0.72	0.74	0.76
8K	0.77	0.78	0.80
10K	0.79	0.80	0.82

6.5 Impact of Caching Only Menu Frames

The locality of reference observed in all the above experiments can be attributed to the tree structure of the data bases. Since the users are constrained to traverse the tree, the frames in the top few levels of the tree are more likely to be rereferenced. In this experiment, the cache hit ratios are determined when only the menu frames (non-leaf nodes) are cached. The hit ratios from this experiment along with those obtained when any frame is allowed to be cached are listed in Table VI. As expected, for the same cache size, higher hit ratios can be achieved by caching menu frames alone. However, the improvement is not that appreciable. An alternate way to interpret this improvement in hit ratio is as follows: if only the menu frames are cached, a given hit ratio can be achieved with a smaller cache.

Table VI

Hit Ratio When Only Menu Frames Are Cached

cache size	any frame is cached	only menu frame is cached
.125K frames	0.24	0.27
.25K	0.42	0.47
.5K	0.61	0.64
1K	0.66	0.67
2K	0.70	0.71
4K	0.74	0.76
8K	0.78	0.81
10K	0.80	0.82

6.6 *Impact of Replacement Policy*

The choice of cache replacement policy may have an impact on the cache performance. Here, hit ratio is studied under two different cache replacement policies, LRU (least recently used) and FIFO (first in first out). When a frame is to be inserted in the cache and the cache is already full, it replaces another frame. In the case of LRU, the frame that has been used least recently is replaced while in the case of FIFO, the frame that has been in the cache for the longest time is replaced. The hit ratios are determined for different cache sizes. The results are given in Table VII. Although there is no dramatic difference in performance, LRU replacement policy tends to provide better performance than FIFO policy.

7. Conclusion

In this paper, we have considered a caching mechanism for Videotex systems, wherein, a dynamic subset of the information is moved closer to the users. The subset that is moved is dependent on the users' behavior. Since the data base functions and the front-end functions are not implemented within the same processor, moving a subset of the information closer to the users reduces the communication and processing delays involved in accessing the data base server. Therefore, users perceive better response times and the data base server is able to support more users. The effectiveness of this scheme, measured by cache hit ratio, is a function of the locality of references.

Table VII

Hit Ratio Vs. Active Users

cache size	replacement policy	
	LRU	FIFO
cache any frame		
2K frames	0.70	0.67
4K	0.74	0.71
8K	0.78	0.76
10K	0.80	0.77
cache only menu		
2K frames	0.71	0.69
4K	0.76	0.74
8K	0.81	0.80

Through an analytic model and a simulation method, we have explored the performance of frame caching in Videotex systems. We find that when the frames are organized as a tree structure, and the users traverse the tree (as in a menu driven retrieval service), it is possible to achieve hit ratios in the 60% to 80% range, with very small cache size. Because of the tree structure, a single user tends to rereference the frames often and exhibits strong locality (only 10% of the data base is referenced). When there are multiple users, there are several frames that are shared by them, resulting in a high degree of inter-user locality (80% of references across all users are observed to be for only 10% of the data base). We feel that more detailed studies should be conducted to establish rules of thumb for locality of reference in Videotex systems. Varying user related parameters like menu selection probability distribution and retrace probability influences the hit ratio in an expected manner. Cache replacement policy does not have a major impact on the hit ratio.

The simulation methodology discussed in this paper is very general and can be used for detailed studies. However, it is computationally expensive. On the other hand, the analytic model developed in this paper is computationally inexpensive; but it is useful only in establishing a tight upper bound. We believe the two approaches complement each other.

References

[1] F.Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *JACM,* 22(2) (April 1975)

[2] D.D. Chamberlin, et al. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control," *IBM J. Res. Dev.* 20(6), (Nov. 1976)

[3] E.G. Coffman, and P.J. Denning, *Operating Systems Theory,*, Prentice Hall, Englewood Cliffs, N.J. (1973)

[4] M.C. Easton, "Model for Interactive Data Base Reference String," *IBM J. Res. Dev.* 19(6) (Nov. 1975)

[5] D. Gangopadhyay, "Conceptual Model for Videotex Data Bases," Unpublished work, 1983,

[6] H.Kobayashi, *Modeling and Analysis,* Addison-Wesley, Mass. (1978)

[7] M.S. Lakshmi, "Performance Engineering of a Videotex System," IBM Research Internal Report (1985).

[8] S.S. Lavenberg, (editor) *Computer Performance Modeling Handbook*, Academic Press, 1983

[9] J. Rodriguez-Rosell, "Empirical Data Reference Behavior in Data Base Systems," *Computer,* (Nov. 1976)

[10] G.M. Sacco, and M. Schkolnick, "Mechanism for Managing the Buffer Pool in a Relational Data Base System Using the Hot Set Model," *IBM Research Report,* RJ3354, (Dec. 1981)

[11] S. Santo, "An Eye on the CBS Database: a Case Study of Information Access Methods", in *Proceedings of Videotex'83,* London Online Inc., 1983

[12] C.H. Sauer and K.M. Chandy, *Computer Systems Performance Modeling,* Prentice-Hall, Englewood Cliffs, New Jersey, 1982

[13] D. Shasha, "Netbook - Knowledge Based Support for Readers and Writers," *Proceedings of VLDB,* 1985

[14] A.J. Smith, "Cache Memories," *ACM Computing Surveys,* 14(3) (Sept. 1982)

[15] J.R. Spirn, *Program Behavior: Models and Measurements,* Elsevier-North Holland, N.Y., 1977

[16] M. Stonebraker et. al., "The Design and Implementation of INGRES," *ACM Transactions on Data Base Systems,* 1(3) (Sept. 1976)

[17] F. Tompa and A. Schabas, "Trees & Forests: User Reactions to Two-Page Access Structures," in *Proceedings of Videotex'83,* London Online Inc., 1983

[18] J. Ullman, *Principles of Data Base Systems,* Computer Science Press Inc., Maryland (1982)

THE CONTRIBUTION TO PERFORMANCE OF INSTRUCTION SET USAGE IN SYSTEM/370

O. R. LaMaire and W. W. White

IBM Thomas J. Watson Research Center
Yorktown Hts., New York 10598

Abstract

The actual usage of instructions in the different environments in which a processor must run reflects both architectural and workload considerations, while the achieved overall internal processor performance reflects these as well as processor implementation considerations. The most frequently used instructions are not necessarily those which the processor spends the most time executing, either totally or on a per instruction basis. Furthermore, focusing on the effects of instruction usage in one environment may provide an inadequate basis for projecting the effects in other environments. This is illustrated by measurements of production systems executing on large IBM mainframe processors. A significant aspect is the differentiation between supervisor and problem program instruction usage, and the extent of their commonality across environments.

1. Introduction

The internal performance of modern, high end computers is heavily affected by both architectural and implementation considerations. However, the nature of the workload actually being executed on the machines is itself a major factor. It is not hard to find cases where one workload running on a machine executes up to twice as fast as a different workload running on the same machine. In an earlier paper [7], we examined the effect of a major implementation factor, the cache (high speed buffer memory), on processor performance in different workload environments. In this paper we will explore a possibly more direct workload effect, the instruction mix of the workload itself. However, implementation considerations (including the cache itself) come into play when considering overall performance, as it is the time spent executing the instructions which is then important. Examples drawn from real processing environments will be used to illustrate these points.

One of the major aspects of instruction usage is the amount of system code being executed. In some sense, as we shall see, the system code provides a degree of commonality between different workloads, while the problem code may be more different since it is more specialized towards solving a particular problem or performing a particular function. Thus, environments which have a higher supervisor content will tend to look more alike than environments in which the supervisor content is low. From a user's perspective, this is important for two reasons: (1) neglecting the system code may be misleading in terms of processor evaluation, and (2) system code is usually not under the control of the user. From a designer's perspective, this is important because it provides a "core" set of instructions which should occur with some degree of regularity, and then another set of instructions, more workload specific, which will provide the variability which must be addressed in order to design an effective, general purpose machine. With special purpose machines, of course, one can be much more specific as to one's design points, with a corresponding loss of performance if the machine is not used for what it was designed.

In any case, however, instruction set usage should be tracked in real environments. Often this will provide the necessary feedback to designers on what the real life problems are, which should be driving systems design. And this information can be used to calibrate benchmarks which can then be used to provide deeper, laboratory studies of processor performance. For these purposes, it is necessary to look at all the instructions executed in order to see the effects on processor performance, and not just on application characteristics.

The processors with which we will be concerned will be the large mainframes on which businesses run their major production work, including commercial and scientific programs, large corporate data bases, program development, etc. While the implications of much of the material presented below is applicable to such machines offered by many different manufacturers, our focus is on machines using the IBM System 370 instruction set (see [6]), or its variants, and the examples will come from IBM processors such as the 3081K.

In Section 2 below we will discuss internal processor performance, noting the dependence on workload, and how instruction set usage contributes to the overall internal performance. A measurement technology for obtaining instruction mixes using a hardware monitor is discussed in Section 3, and in Section 4 we go into more detail on the measurements themselves. In Section 5 we explore some of the supervisor and problem program content considerations, and a general discussion of the implications, as well as some possible extensions, is given in Section 6. Section 7 presents a short summary of the major points made.

2. Internal Processor Performance

As noted above, the actual use of instructions delineates the work done from an internal processor viewpoint. If all instructions executed in the same amount of time, for example in one machine cycle, a RISC architecture-like feature (see Colwell et. al., [4]), then one could evaluate the internal speed of the processor directly from knowing the total number of instructions executed. If, however, even just a subset of the instructions executes in a different number of machine cycles, then one must know the number of cycles each instruction will take to execute in order to evaluate performance, as well as the instruction mix itself.

CH2345-7/86/0000/0665$01.00 © 1986 IEEE

This is generally more complicated than looking up some nominal value supplied by instruction timing formulas. While such an approach can be workable, e.g., for base timing of particular instructions dependent on operand length, it may have little connection with actual execution as experienced within a full system context. In this case, the interaction of the instructions with other aspects of the workload behavior may dominate the 'kernel-like' characteristics. In particular, many machine implementation considerations will have an effect, especially in high-performance processors. These include pipeline breakage due to branching, holding off of instruction execution due to lack of operands (or of the instruction itself) being present, memory interference, and so forth.

To do a full performance evaluation from first principles, one needs to get at more information than just instruction mix and instruction timing formulas (see, for example, Peuto and Shustek [10]). However, one can do a characterization of the performance impact of instruction usage by measurement 'after the fact', i.e., by actually measuring code execution, as in Clark and Levy [2], although the focus there is more specific rather than at system level behavior across environments. Depending on what data one collects, one can gain insight into how the instructions of a particular workload drive performance.

To provide a somewhat more specific basis for our discussions, let us call f_i the fraction of occurrence, i.e., the normalized frequency, of the i^{th} instruction, so that

$$1 = \sum f_i$$

In the following, we will also sometimes refer to f_i as a percent instead of a fraction, but the context will make it clear which is which.

It is generally convenient to assume an order to the instructions. In particular, one that we shall use will be the frequency of occurrence itself with respect to a given workload, so that, for that workload, $f_i \geq f_{i+1}$ for all i. For other workloads ordered according to the given workload, this relationship need not hold, of course. With this ordering, then, we can define the partial sum

$$F_p = \sum_{i=1}^{i=p} f_i$$

of the first p instructions, ordered according to the given workload. As a percent, F_p is then the percent of the total instructions for this workload executed by the top p instructions of the given workload. This metric, a cumulative frequency distribution, is of interest as it is indicative of the "working set" of instructions for this workload. The differences between these values for different workloads, keeping the same instruction ordering, shows the amount of variation which can occur between environments, with implications as to the necessity of robust design. One notes that the F_p for the given workload itself must be at least as great as any F_p for this workload generated by the order of some other workload.

If we then look at the total execution time for all instructions, i.e., the total processor busy time, we can try to allocate it to individual instructions, as follows. Let t_i be the fraction of the total busy time attributable to the execution of the i^{th} instruction, in such a fashion that the *whole of the busy time can then be accounted for.* Using this definition, it is then true that

$$1 = \sum t_i$$

It should be noted that this is a definitional statement, where we are apportioning the total amount of processor busy time between all instructions, including the time spent on all the other machine dependent factors noted above.

A simple way to achieve this is to attribute all the time spent between the start of execution of one instruction and the start of execution of a second instruction to the first instruction. In a strictly sequential machine, the applicability of this simple approach is clear, while in a pipelined machine, this amounts to placing emphasis on the execution element, and ignoring some instruction fetch and other overlapped activity. An alternative simple approach, employed in Clark and Levy [2], is to attribute the time between successive *decodes* to the first instruction: in a sequential machine this is largely equivalent to the first approach (i.e., between starts of execution) but in a more pipelined machine, this places more emphasis on instruction element processing (missing some overlapped execution). While some care must be taken in interpretation in either case*, we use the first approach here (between successive starts of execution) as we prefer an execution element emphasis as providing more insight into processor performance behavior.

Other ways of attributing time may be more meaningful depending on machine implementation, and, in fact, one may gain additional insight into machine behavior by not apportioning all the busy time to actual instruction execution, but splitting some of the time off for other factors (e.g., for cache effects as suggested by LaMaire and White [8], or for more detailed performance factors as delineated in MacDougall [9]). A very good example of this is in Emer and Clark [5], where this is done on groups of opcodes. In practice, much of this depends on the signals available from the processor under study. One could also go further and develop expressions relating to the average execution time per instruction, but we will not do so here, as it is less important to the development of this paper.

As with instruction frequencies, one can also assume an ordering for instruction times. The one which we shall generally employ here will be the same as for instruction frequencies, i.e., we shall assume an ordering of the t_i according to the frequency, *not* time. Thus T_p as defined by the relation

$$T_p = \sum_{i=1}^{i=p} t_i$$

for the first p instructions ordered according to the frequency of occurrence in the given workload, is the contribution to the total busy time of these instructions. The degree to which T_p is less than F_p is indicative of the performance impact of infrequently used but "long" instructions. In words, one can talk about p instructions accounting for F_p of the instruction use, but only for T_p of the time (with some implicit ordering assumed).

We shall focus on these two metrics F_p and T_p as the items of interest, in that they summarize, in the first case, the usage of the instruction set, and in the second case, the impact of this usage on

* We note that either of these approaches can be used for other processor organizations, including vector processors which execute instructions synchronously, although they both become less and less applicable the more overlapped the operations become, e.g., for multiple execution units fed from a single instruction processing element. Overlapping in multiprocessor systems can be handled by examination of each processor in the configuration separately.

processor performance, i.e., on where the processor spends its time. A more detailed analysis could delve into individual instructions and their characteristics (as noted in Section 6), but our main purpose is to develop some overall characterizations of instruction set usage. Such overall system-level characterizations will serve to illustrate our principal points regarding instruction set skew, environmental differences and similarities, and the importance of considering supervisor content in such analyses.

3. The Measurement Approach

There are many different ways to get instruction usage data, often depending on, or limited by, the nature of the workload environment being examined. In laboratory environments one can instrument processors, both by hardware and software monitors, to gather a wealth of information. In such cases, one can often repeat measurements (in fact, repeatability is a major feature of laboratory environments), so as to extract more data in multiple passes. This might be necessary, for instance, when the amount of information gathered per pass must be limited, in order to minimize processor interference induced by the monitor, or because of a limited collection capability in the monitor.

There is a difficulty in focusing on workloads in laboratory environments, though, in that it means one is dealing with closed, largely known entities, which may or may not be representative of how real production workloads execute on these processors. Both this representativity, and the question of how many other environments may exist, indicate the need to address oneself to real world behavior. In the very least, one can then develop a feel for how representative are the laboratory benchmark workloads. In other cases, one can get input from workloads not easily created in the laboratory, such as large data base systems, and see if such workloads lie in the design space of the processors themselves.

Gathering data from real production workloads implies many operational restrictions on the collection process. First, there must be minimal impact on the real system. Second, one has to get all the data 'on the fly', as there is no controlled environment -- repeating an experiment by re-running is not possible. This points to the use of hardware monitors, as software monitors tend to cause processor degradation (although for some kernel analyses, as in Bonnes [1], they can be of use). An additional problem with software monitors in general environments is that, to be efficient, they must be sampling monitors, and it is difficult to handle the bias introduced when the operating system inhibits getting a sample (e.g., when interrupts are inhibited). The fact that one is interested in all instructions, including those executed while in supervisor state (meaning that the software monitor cannot simply be an application code) compounds the problem.

For our measurement activity we have employed a specially designed hardware monitor to collect such data as may be available from the large IBM processors in which we have an interest. This monitor operates transparently to the processor being measured, that is, there is no processor degradation, and is readily transported to different real production locations. These sites are chosen as leading edge systems, both in terms of the compute power required for the applications, and for the importance of the applications themselves. Effort is made to look at a variety of different production workloads. Typically, a measurement takes place over a week, 24 hours a day, to gather data on the overall measured system, with data being written to a recording device at a frequency sufficient to profile the system at, for example, 15 minute intervals. During periods of greater interest, however, such as during peak hour operation, snapshots of 5 to 15 minutes are taken at a high resolution, i.e., a high recording rate, to allow for more detailed study. There is sufficient commonality in the data being collected so that, from the overall profile, one can identify if anomalous behavior existed during one of the snapshot periods (e.g., extensive error processing) and therefore ignore the data gathered during that unrepresentative interval.

The hardware monitor employed has some special features which allow the collection of instruction information. In addition to a set of accumulators generally available in these monitors, it contains an 8 bit decoder. Since the primary instruction code, the operation or *op code*, for System/370 instructions (see the architecture documents [6], [7]), is eight bits long*, decoding these eight bits and then incrementing one of 256 registers addressed by this decoding effectively counts the occurrence of that instruction, at the time when the decoding takes place. In order to make use of this facility, one must guarantee a valid strobe for telling when the particular 8 bits are to be decoded (and, of course, one must know that the 8 bits do in fact represent a valid op code at that time). Typically, one can generate a strobe signal corresponding to the first machine cycle of the execution of an instruction. Assuming the 8 bits present at that time do in fact represent the instruction being executed, this facility will then collect in its 256 registers the instruction mix being executed. Alternatively, if one 'latches up' the 8 bits so that they are valid until the next instruction is executed, then by strobing on every machine cycle one can get a time distribution of the instructions, i.e., each register then contains the number of machine cycles spent executing the corresponding instruction**. By performing logical combinations on the strobe signal, for example, 'ANDing' it with the supervisor state signal, one can get instruction information about what's happening in supervisor state. During a week's measurement, several snapshots are in fact taken with different such 'conditionings' being done.

Another practical aspect is that to gather both instruction mix and instruction time distributions requires two such facilities, and if one is measuring a two-processor multiprocessor system such as the IBM 3081K, getting all the information desired would require four such facilities. This then becomes a substantial hardware and software issue, as well as an operational one. To ease these difficulties, we multiplex the use of a single facility, as follows: during one recording interval (say of one second's duration) we gather the data one way, and, after writing it out, we gather the information a second way during the next interval, thus alternating among the possibilities. For a single processor machine such as the IBM 3083J this amounts to alternatively collecting the data so that half the time we are collecting instruction mix data, and half the time instruction time data, interleaving the collection periods. By interleaving many small (relative to the run length) periods, and performing some calculations as to the stability of the mean of the

* Technically, for those instructions with an extended op code, the secondary op code, also 8 bits, is also of interest. To some extent, since not all 256 possibilities of the primary or secondary op codes are present, one might 'fold' these together into one set of 256 registers, thus collecting a full set of information. In this paper, however, we will restrict our attention to the primary op code only, without differentiating on the basis of different secondary op codes.

** In practice, looking at every machine cycle requires a very fast monitoring capability, particularly on modern high speed processors with cycle times of under 25 nanoseconds. Consequently, a sampling procedure is usually employed, looking every *k* machine cycles and decoding at that time. By taking *k* to be a small, prime number, say, less than 32, the sampling error for runs the length considered here is minimal.

collected data, one can collect both kinds of instruction data during one measurement, thus providing most of the information needed with an economy of operation and expense.

4. Measurement Experience

Table 1 gives a high level description of several different measured real life production systems. While these are all IBM 308x processor family environments, one should note that the 9083 is a uniprocessor, the 3081's are dyadic (2-way multiprocessors), and the 3084 is a 4-way multiprocessor (although when run in partitioned mode, it runs as two separate 2-way processors). The 3081D and 3081G processors each have a 32K cache per processing unit (i.e., 2 32K caches total), while the other processors have 64K caches per processing unit (plus other enhancements), so that one would expect different ranges for the performance parameters of interest, particularly those dependent on cache activity (see LaMaire and White [8] for more detail on this). Also, the 3081D and 3081G processors differ somewhat in their internal organizations from each other and from the 3081K and Q machines which will further change the average cycles per instruction. Additionally, the cycle times on the machines are not all the same.

Three different operating systems are present, and the MVS operating system itself has two distinct versions, SP1 for System/370 mode operation, and SP2 for System/370 Extended Architecture (370-XA) mode. The workload environments range from scientific batch, to VM/370 and TSO time-sharing for interactive program development and execution, to three different data base data communication environments, IMS and CICS operating under MVS, and the TPF high transaction rate environment for airline reservations. The VM/370 environments also contain a fair amount of 'personal computing': mail, text processing, and the like.

ID	Processor	Operating System	Description
A	3081D	VM/370 HPO2	Program Development, Text, Scientific Processing
B	9083	TPF2	Airlines reservations
C	3081G	MVS/SP2.1	CICS production data base, background batch
D	3081G	MVS/SP2.1	CICS production data base (dedicated)
E	3081K	MVS/SP1.3	Scientific batch
F	3081K	MVS/SP1.3	Scientific interactive
G	3081K	MVS/SP1.3	Scientific interactive with background batch
H	3081K	VM/370 HPO2	Program Development, Text, Scientific Processing
I	3081K	MVS/SP1.3	Scientific batch
J	3081K	MVS/SP1.3	Scientific TSO, interactive graphics, background batch
K	3081K	MVS/SP1.3	Scientific TSO, interactive graphics, background batch
L	3081K	MVS/SP1.3	Scientific TSO, batch
M	3084Q	MVS/SP2.1	IMS production data base (2 of 4 CPUs measured)
N	3084Q	MVS/SP2.1	IMS production data base (Partitioned mode, performance enhanced)
O	3084QX	MVS/SP2.1	TSO, background batch (2 of 4 CPUs measured)

Table 1: Workload Summary

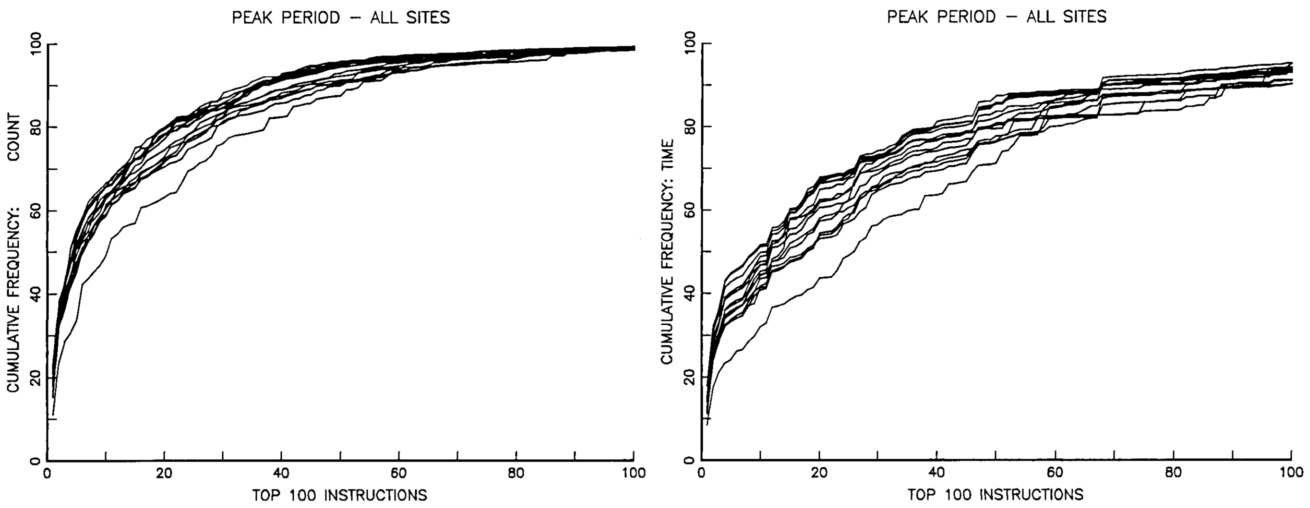
There is a mix of languages present across workloads, and most of the workload environments themselves include many different languages. The operating systems tend to use assembler or related low level languages, as does the code in the major subsystems such as IMS and CICS. The scientific applications tend to use FORTRAN in various flavors, and PL/I in some instances, while the more commercial applications can be COBOL or PL/I; some special constructs are also used for graphics processing. In general, it is not possible to pick out individual language characteristics from such a melange: more controlled studies would be necessary, which is not possible in these production environments (it is usually difficult enough just to identify which applications are being run, e.g., to what extent sorts or other utilities in general are part of the workload mix). From other experience, we share with Clark and Levy [2] the conclusion that it is how the application uses the language, not so much the language itself, that is important. As such, we have characterized the environments in broad terms indicative of their principal activity.

The measurement data represents peak period operation of these workloads. Peak period is taken to be a two-hour period, which can vary between sites, during the day time when the processing load is the heaviest, e.g., in terms of processor utilization*. Often this is in the afternoon, such as between 2 and 4 PM. The data comes from the 5 to 15 minute snapshots taken during this period, for as many days of the week as possible, usually comprising on the order of 2 hours total.

For each of the measured environments, Figure 1 presents instruction frequency data in terms of the cumulative frequency, F_p for $p \leq 100$. The instruction ordering used here is that of the average instruction frequencies for all the environments, so that the figures represent the top 100 instructions of this 'average', respectively. As there are about 170+ primary op codes in the S/370 and S/370-XA instruction sets, slightly under 60 percent of them are represented in Figure 1. The 'outlier' is a scientific batch workload, Workload I, probably the most heavily scientific of the set (it is, however, a real workload -- there is no basis for discarding its data as anomalous). The other workloads separate, of sorts, into two pieces, as can be noted by looking at the gap between the sets of curves at the top 20 or top 40 instructions: the lower group is predominantly more scientific in nature than the upper group.

If we consider the top 17 instructions (10 percent of all instructions), we see that these account for from slightly more than 60 to somewhat under 80 percent of the instructions executed, depending on workload; the average is slightly more than 70 percent. For the top 35 instructions (20 percent of all instructions), we see that these account for from somewhat under 80 to slightly more than 90 percent of the instructions executed, again depending on workload; the average here is in the high 80's.

* This 'peak period' view is oriented toward workloads for which response time is a criteria, since much of the work done at that time has an interactive content. As such, the data is indicative of those sites' stress environments which, for example, one might use as a basis for characterizing high daytime load. Not included here is overnight processing, which more often has a 'turn-around' criteria, e.g., the work must finish by the next morning. This kind of work often consists of longer running jobs, utilities, and the like, and may have more problem program activity. Scientific work, in particular, may have a higher computational content. This means that one may well expect more variability in instruction use data were one to include nighttime activity in the set of measured data.



Figures 1 and 2: Cumulative Instruction Distribution for All Sites
by Count and Time for the Top 100 Instructions Ordered by Average Frequency

From another perspective, to achieve 70 percent of the instructions executed requires anywhere from 13 to 25 of the top 100 instructions, varying according to workload (13 to 20 disregarding the 'outlier'). This 50+ percent variation of the maximum over the minimum is not atypical for other percentiles, and gives some indication of the range one must design for, in terms of 'instruction working sets', when addressing these environments.

We note that instruction mix data should not be as sensitive to machine implementation as instruction timing data should be, since the instruction mix is principally driven by the kind of work done on the machine. Instruction timing data, however, is influenced both by the work done, and how (in terms of processor implementation) it is done, thus leading us to guess a larger variation in the cumulative instruction times across workloads.

Figure 2 in fact shows what the instruction time distribution is for the measured sites. Again, the instructions are the top 100 instructions executed, ordered by frequency, *not* time, for the overall average frequencies. In other words, we have plotted the cumulative instruction times T_p for $p \leq 100$ for the same instructions p as we did above. The variation is indeed larger. Again, Workload I still stands out, but the other workloads do not separate as obviously. However, the more scientific workloads tend to be the lower curves, as before. The differing machine implementations (cache size, cycle time) do not stand out significantly; the workload effects are substantially more dominant. The plotted curves are much more 'wavy' than those in Figure 1, indicating the effect of 'long' (in terms of machine cycles per instruction) ops, and the extent of their commonality across the environments.

When considering where processors spend their time, then, we observe that now the top 17 (10 percent) of the instructions executed account for from about 40 to somewhat over 60 percent of the time of execution, and the top 35 (20 percent) account for from about 60 to somewhat under 80 percent of the time of execution. In the extreme, in some of the environments, 10 percent of the time has not been accounted for by the top 100 instructions, indicating that some very infrequently used ops may account for significant portions of the execution time.

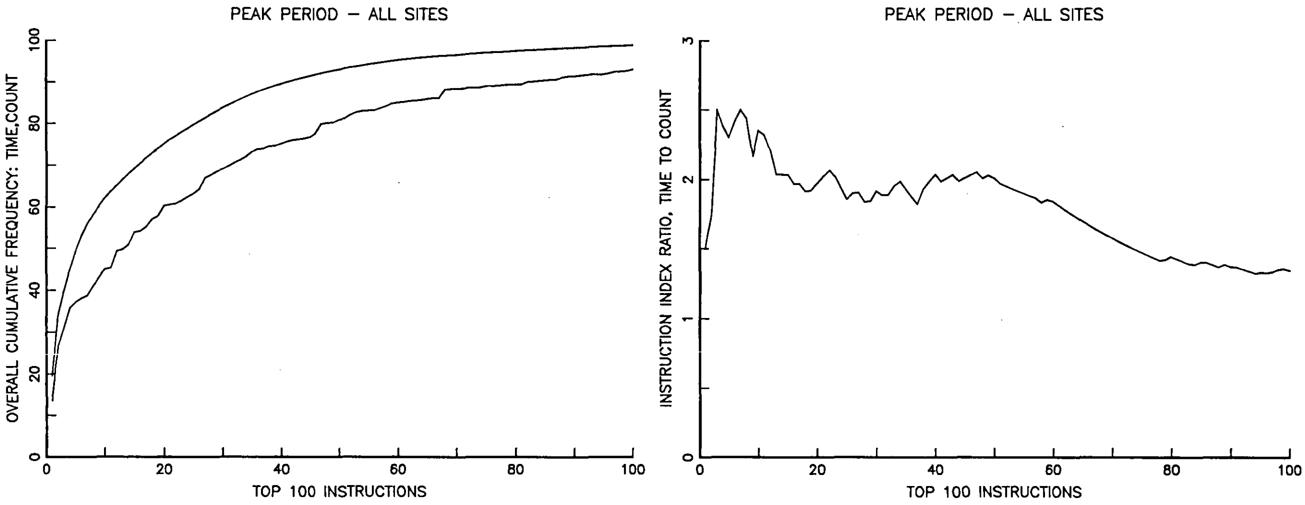
From our other perspective, to achieve 70 percent of the instruction time being accounted for requires anywhere from 26 to 47 of the most frequently used (on the average) instructions, according to workload (26 to 42 disregarding the 'outlier'). The 50+ percent variation of the maximum over the minimum is still generally true for other percentiles, and again shows the range one must design for, from an instruction execution point of view.

One can put this data, both count and time frequencies, together to see how much coverage of execution time is given by instruction usage. To simplify things, we just do this for the overall averages in Figure 3. The two curves then represent the cumulative average instruction use F_p and time T_p frequencies, with the former being the upper curve. The instruction order is, as before, that of decreasing average instruction use frequency, so that, as one would expect, the upper curve is the smoothest presented so far.

From this figure we can note the on-the-average 'coverage' behavior. The top 17 (10 percent) of the instructions account for slightly more than 70 percent of the instruction usage, but only around 55 percent of the time, assuming the given instruction ordering. In order to account for, say, eighty percent of the execution time, one has to consider those instructions accounting for over 90 percent of usage. And while it is true that a small number (12-14) of instructions account for two thirds of the use, they account for under half of the time.

Figure 4 presents another way of looking at the data of Figure 3. Here is plotted the ratio of the number of instructions needed to get roughly equal percents, timing to counting. That is, if one draws a horizontal line on Figure 3 and notes the two instruction indices of the places where the line intersects (one for the cumulative instruction count distribution, and one for the cumulative time), the ratio is that index for the time curve over the index for the count curve*. The ratio then gives the factor increase for the number of instructions needed to obtain the same 'coverage' of

* Actually, since the distributions are discrete, the index for the time distribution is the average of two indices, one being the maximum index i for which $T_i \leq F_p$ and the other being the minimum index j for which $T_j \geq F_p$, for the same given ordering.

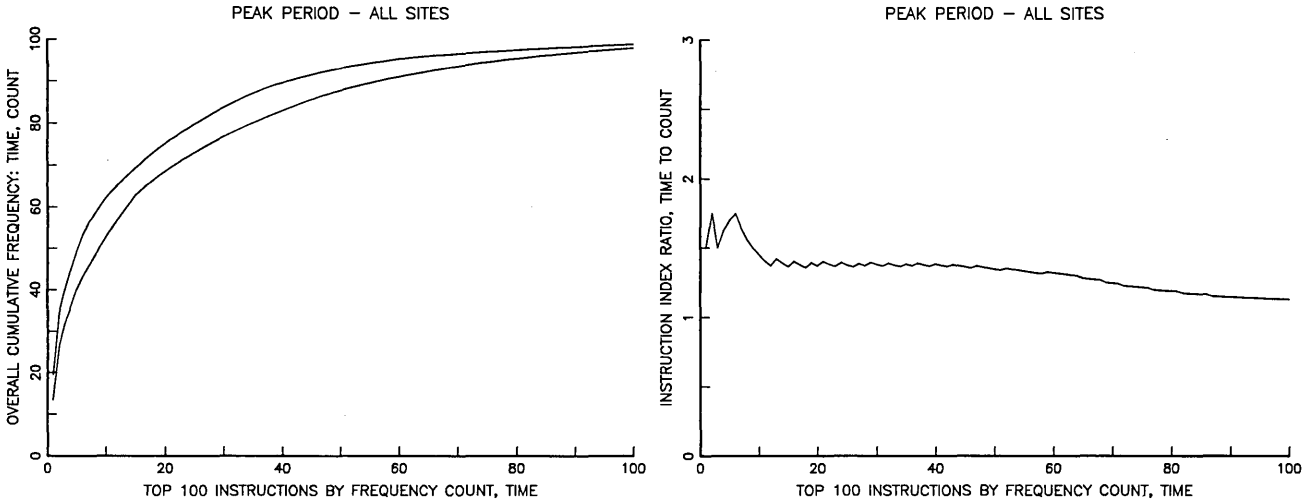


Figures 3 and 4: Cumulative Instruction Count and Time Distribution and Ratio of Equivalent Instruction Indices for the Average of All Sites for the Top 100 Instructions Ordered by Average Frequency

execution time as the given number of instructions 'cover' instruction use. One can note that, over a wide range, one needs 1.5 to 2 times as many of the top most frequently used instructions to get the same percentile coverage of execution time as of instruction use.

It is clear that in order to gain maximum performance, one should concentrate on a somewhat different set of instructions than would be dictated by strictly frequency-of-use considerations. For a general architecture, however, this can lead to difficulties, as one would expect that over time implementations will differ as new processor models are introduced, and what was stressed at one time may not be as important later. The general frequency of use should be a more important consideration long term, as this is more closely associated with the work which the processors must execute (although seeing how this evolves over time is interesting, and is important both for architectural considerations, and to see where design emphasis should be placed). This would give one a more solid base, and the time distribution of instructions would then be indicative of how 'all-encompassing' and robust the most frequently used instructions should be implemented.

One can compare the distributions, instructions ordered by frequency to those ordered by time, to get a rough estimate of what the 'best case' might be in terms of 'coverage'. Figure 5 shows these two cumulative distributions for the top 100 instructions of each ordering. The top curve is still the frequency curve, as in Figure 3, while the bottom curve is the time curve. It is evident that even if the most frequently used instructions were the same as the ones to which the most execution time were attributed, it



Figures 5 and 6: Cumulative Instruction Count and Time Distribution and Ratio of Equivalent Instruction Indices for the Average of All Sites for the Top 100 Instructions Ordered by Average Frequency for Count and Execution Time for Time would still take more instructions to 'cover' the same percent for time than for frequency. In fact, the plot of the instruction index ratios for equal coverage, Figure 6, shows that, for a wide range of interest, one still needs about 1.4 times as many instructions to achieve the same percent coverage by time as by count, even in this 'best case'. This same property holds on a workload by workload basis as well, although the 'equal coverage factor' does vary.

5. Workload Content Considerations

In the Introduction it was mentioned that the supervisor component of workloads, because it is present to some extent in all workloads, might provide a thread of commonality across environments. And to the extent that it may dominate some workloads, it could be quite important in and of itself. We can see how important this might be by borrowing some ideas presented in LaMaire and White [8]. If we let N be the average instruction time (in, say, nanoseconds per instruction), then we can write

$$N = \lambda N_s + (1 - \lambda) N_p$$

where N_s is the average instruction time when in supervisor state, and N_p is the average instruction time when in problem state. Thus λ is the fraction of the time in supervisor state. Clearly, as λ approaches 1, the supervisor state workload component dominates the overall execution time.

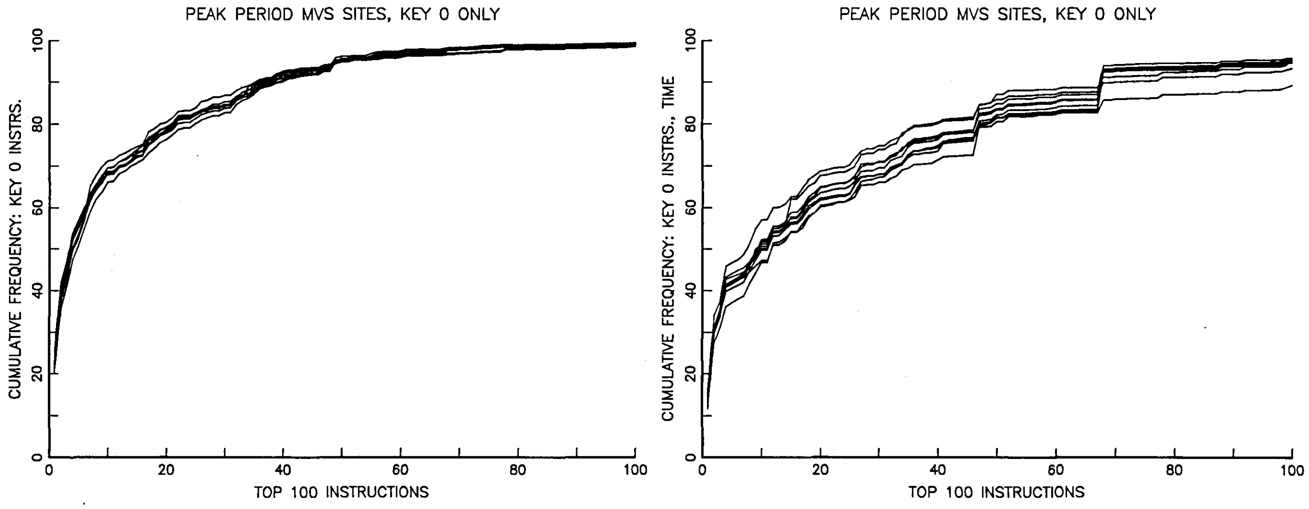
To see how important this may be, we can examine the measured workloads. But here again, we run into practicalities. Some workloads treat supervisor time differently than others. For example, the IMS data base control program executes largely in supervisor state, but is not present unless one is running an IMS data base; other major application subsystems have similar characteristics. But let's consider that what we are really looking for in terms of commonality is the functions which are accomplished in the operating system itself, in what is called the system control program. For the MVS operating system, this is code which executes predominantly in Storage Protect Key 0. As a further simplification, we will just look at the measurements from sites using the MVS operating system. Thus, as a surrogate for a general supervisor state time, we shall consider MVS Key 0 time. The fraction of the total busy time spent in Key 0 for nine of the MVS environments appears in Table 2*. This Key 0 ('supervisor') fraction ranges from .2 to .7, so that for some of these environments the Key 0 component is indeed a major contributor to performance.

Id	Description	Key 0 Fract.
E	Sci. Batch	.2
F	Int. Sci.	.6
G	Int. Sci with Batch	.3
I	Sci. Batch	.2
J	Sci. Graph with Batch	.6
K	Sci. Graph with Batch	.5
M	IMS data base	.5
N	IMS data base	.6
O	Prog. Devt with Batch	.7

Table 2: Fraction of Busy Time in Key 0 for the MVS Sites

Let us now consider the instruction usage of our pseudo-supervisor code, i.e., the code executing in MVS Key 0. This Key 0 instruction data for the sites in Table 2 was gathered similarly to the way the overall data for all the sites was gathered. In this case there were also 5-15 minute snapshots taken during the 'peak periods', although there were fewer snapshots, and, of course, not at the same time. Thus while the data for these Key 0 periods is still representative of peak period behavior, one would expect some small differences between this data and the overall data in some of the principal performance metrics, the source of which comes from the natural variability of the workload. These differences are not significant in any major way.

* Data is not included for the 3081G (Workloads C and D), as related instruction data was not readily available, nor for Workload L, one of the 3081K sites (for which the related instruction data was not available).



Figures 7 and 8: Cumulative Instruction Frequency and Time For Selected Sites for Key 0 Instructions Ordered by Frequency of Top Instructions Averaged over All Sites

Figure 7 presents the Key 0 instruction mix cumulative frequency for the measured sites. The ordering of instructions is the same as before: the decreasing average frequency of all instructions over all sites. Note now the high degree of similarity between the Key 0 instruction usage between these sites. The 'waves' indicate differences between instruction usage between the Key 0 ('supervisor') component and the overall general instruction usage., i.e., show where some instructions occur more frequently in Key 0 than in the general case. By comparing Figure 7 with Figure 1, one notes that the curves in Figure 7 lie more along the upper curves of Figure 1, which as we had noted, were the workloads which have a higher interactive content (and as Table 2 shows, have a higher fraction of Key 0 time). One sees that indeed a fair commonality exists across the workloads coming from the Key 0 component, and that it is dominant in workloads with a high fraction of Key 0 time.

Figure 8 shows the corresponding Key 0 cumulative instruction time distribution. Again there is less variation across the sites than in the general case, though more so than when considering just instruction frequencies -- while Key 0 uses similar instructions, some of them may be taking more time in some of these workloads than in others.

This commonality of Key 0 usage across the sites seems indeed to reflect the commmonality of supervisor function. All environments require I/O and interrupt handling, task scheduling and switching, and various other supervisor services, albeit in possibly somewhat different proportions. If one accepts the paradigm of an operating system providing its service at the request of a user program or to handle an interrupt, the similarity across environments seems intuitively likely, even though these services are invoked less frequently in some environments than in others. For the sites given here, for example, there is no readily apparent differentiation between the curves for more compute bound as opposed to more I/O bound environments.

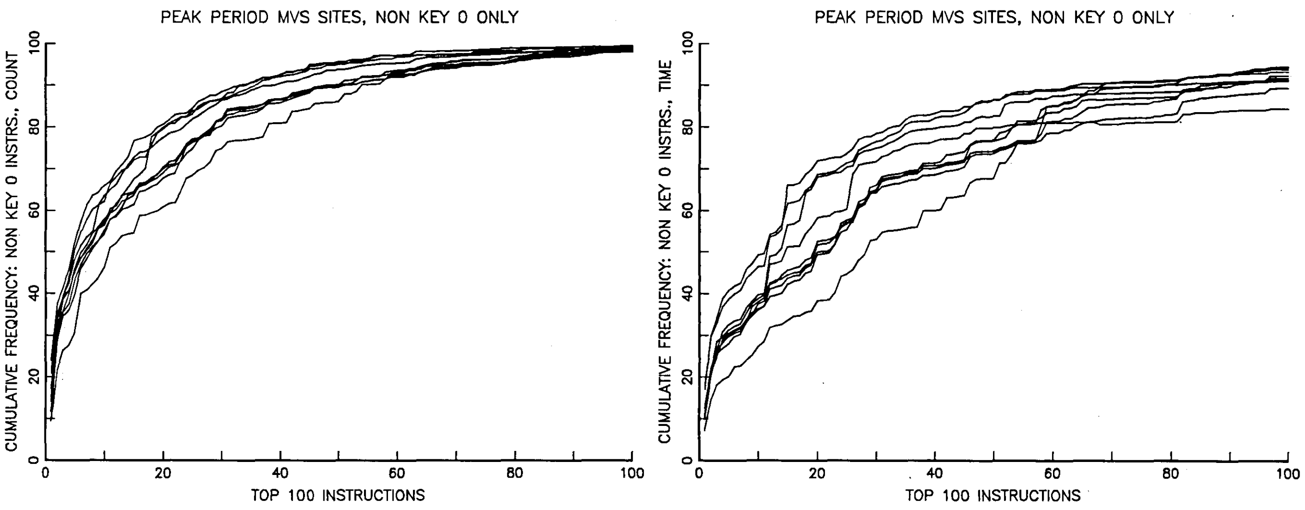
We can also examine the pseudo non-system code instruction usage, or, as we use it here, the 'non-Key 0' instruction usage. This non-Key 0 component corresponds to application code and major subsystem (e.g., IMS) code. The data is derived by subtracting the Key 0 data from the overall data, with appropriate weightings. As such, it is somewhat artificial, since the overall data and the Key 0 data came from two different sets of snapshots, but it is a reasonable representation of 'non-Key 0' activity.

Figures 9 and 10 present cumulative instruction mix and instruction time distributions for the non-Key 0 instruction usage; the instruction ordering is our standard decreasing overall average frequency for all sites. There is substantially more variation between sites than seen earlier, indicating that the variation in the overall instruction use (and time) comes from non-Key 0 usage differences. In addition to the 'outlier', Workload I, showing up, one can also see a similarity within the four 'scientific' workloads, and, above them, the four 'interactive' workloads, particularly for instruction use frequency (Figure 9).

6. Discussion

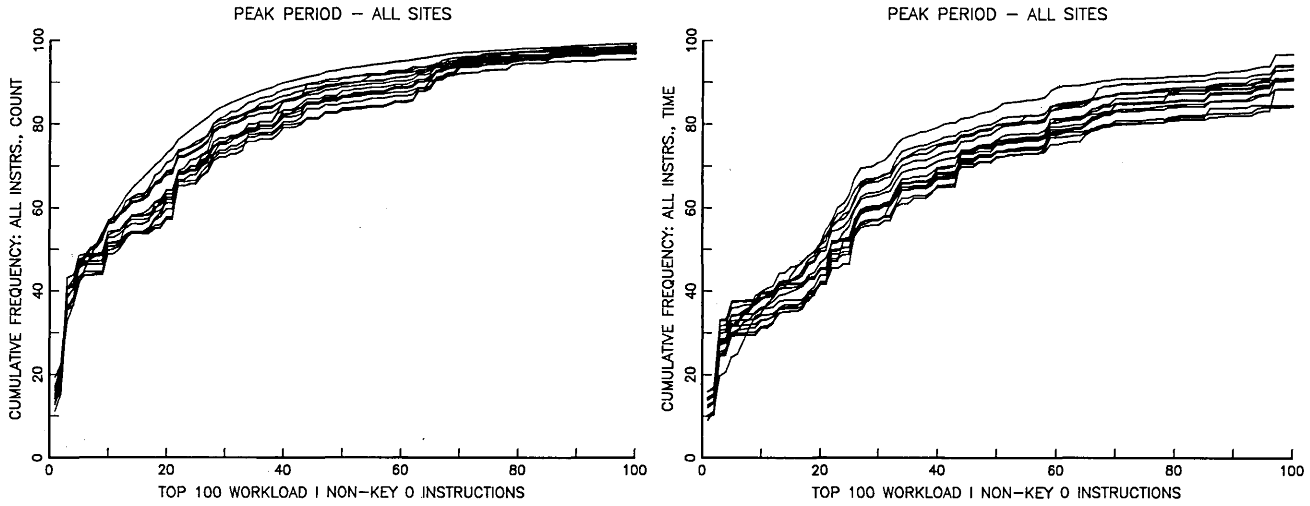
The foregoing shows that there is a certain amount of variability in instruction usage across different environments, with some commonality based on the presence of supervisor code in each environment. The way processors are implemented affects instruction execution time, and accentuates the variability. Because of this variability, it is important for robust design to consider different possible instruction mix contexts when evaluating potential system performance.

As an example of this, suppose that instead of the decreasing overall average instruction frequency as an ordering, we select an ordering based on the decreasing average instruction frequency for one particular workload. Even more, let us use the decreasing average non-Key 0 frequency. This would be somewhat akin to basing design decisions on a particular sort of application code. To illustrate this, we shall use Workload I.



Figures 9 and 10: Cumulative Instruction Frequency and Time For Selected Sites for non-Key 0 Instructions Ordered by Frequency of Top Instructions Averaged over All Sites

* The curve in Figure 11 is not as smooth as, for example, the top curve in Figure 3, since the curve represents the cumulative use of *all* instructions, but the ordering is based on *non-Key 0* instruction frequency.



Figures 11 and 12: Cumulative Instruction Frequency and Time For All Sites for All Instructions Ordered by Frequency of Top non-Key 0 Instructions of Workload I

Figures 11 and 12 present cumulative instruction mix and instruction time distributions for instruction usage for all sites; the instruction ordering is now the decreasing average frequency for the non-Key 0 instructions of Workload I.*. In both figures, the curve for Workload I starts out below the curves for the rest of the workloads, but becomes the top-most curve after 12 to 20 or so instructions, for counting and timing respectively. This shows the extent of mismatch with general instruction usage, on the average, were one to use Workload I application code as a design point: while for the first few instructions, one would be safe (since their use in other workloads is higher than in Workload I), one would be exposed after that. And comparing Figures 1 and 11 and Figures 2 and 12 shows that the extent of coverage is less when using Workload I application code as the base, since the general curves in Figures 1 and 2 are generally higher than their counterparts, particularly in Figure 2 in the range of 10 to 30 top instructions.

Akin to the problem of just using application code as a base for instruction use analysis (unless, of course, one only wants to characterize that particular code) is just using a subset of the instructions as a base for analysis. It was noted in the discussion following Figure 3 in Section 4 that while a given set of the top instructions may 'cover' a fair portion of the total instruction use, they usually 'cover' less of a portion of the total execution time. From a design point of view, this means that analyses based on the top so many instructions may not generalize well, i.e., there could be performance exposures in not evaluating substantially more of the instructions. In Figure 3, 90 percent of the instruction use accounted for about 75 percent of the execution time. If, for example, one did not look closely at the instructions accounting for the last 10 percent of the usage, resulting in an implementation which was slower by, say, a factor of 2 for these instructions, then 40 percent of the total execution time would not have been accounted for, a 25 percent increase in overall execution time.

There are several extensions that would be of interest, both from an instruction use and from a performance analysis orientation. For instruction use, one could do more in terms of instruction categories, such as statistics on instruction type usage (e.g., register-to-register, register-to-storage, etc.), or instruction function usage (e.g., loads vs. stores vs. branching, etc.). A number of these would have implications as to addressing hardware, storage bus, computational (e.g., floating point) element design, and so forth. Clements and Kolence [3] provide three general categorizations for software-sampled data, and Peuto and Shustek [10] give several in-depth examples of such categorizations, albeit for problem programs, along with a good discussion of why one looks at them. Emer and Clark [5] present results for groups of instructions where the data is obtained directly, not by summarization from individual instructions; the groups are meaningful for their analyses. What would be harder here would be to look at sequences of instructions and operand length distributions, for which data is not directly gathered (although some of this can be backed out, e.g., BC instructions generally follow TM instructions, and one might estimate operand length by looking at the cycles per instruction for particular instructions, such as LM or MVCL).

A more interesting extension is the integration of the approach presented here with more general processor performance models. Conceptually, one would like to incorporate implementation considerations, such as pipeline breakage, cache effects, and so forth, as noted in Section 2 above, and as presented in MacDougall [9]. The extent of the model presented in that paper is generally beyond the ability of a single performance tool to produce the data for, particularly in real environments. But there may be ways to extend the measurement capability, combining the approach presented here with that in LaMaire and White [8]. In that paper a fairly simple model of internal processor performance is given, which separates performance into a component having to do with execution time when all instructions and data are found in the cache, and a component relating the effect of delay when information is not found in the cache. The attempt here would be to try to apportion the in-cache effect on an instruction basis, similar to that presented in this paper.

7. Conclusion

One can conjecture about how wide the application of the foregoing is to different architectures and implementations. It does provide some insight into current instruction usage and its performance impact for large IBM mainframe processors. However, the actual numbers should be less important than some of the general observations that have been noted.

The first of these is another confirmation of the skew (with long tails) in instruction usage, as has been noted by others (Clark and Levy [2], Peuto and Shustek [10], etc.), and that this holds at the system-wide level as well as for problem code. Related to this is the general tendency in these cases of instruction mix 'coverage' to be higher than instruction execution time 'coverage', for the same most frequently used instructions. Presumably, the frequency of use data for the top instructions should always dominate the time data for these instructions for most of the environments for which a processor is designed, because the biggest performance gains come from speeding them up, and hence the most attention should be paid to their implementation. Considering these two points, the implication here is that analyses based on a certain number of frequently used instructions may be overly optimistic when considering the performance impact.

Another observation relates to the commonality of supervisor code across environments. This does, in effect, provide a "core" set of instructions which should occur with some degree of regularity, even though this core may appear in different proportions in different environments.

Concomitantly, the particularity of problem program code is what will give an environment its own flavor. These more workload specific instructions provide the variability which must be addressed in order to design an effective, general purpose machine. Consequently, there is a need to address a variety of mixes (especially for general purpose machines) to get a feel for the need for robust processor design.

Finally, one should analyze production systems; after all, the final assay of value lies in the crucible of the real world.

References

[1] Bonnes, A. H. J., "Instruction Execution Time Measurements in MVS Systems", Computer Performance, September 1983, pp. 167-175

[2] Clark, D. W., and H. M. Levy, "Measurement and Analysis of Instruction Use in the VAX-11/780", Proceedings of the 9th Annual Symposium on Computer Architecture, April 26-29, 1982, pp. 9-17

[3] Clements, R. and K. W. Kolence, "Building Workload Profiles to Estimate Practical CPU Power", Proceedings of CMG '85, The Computer Measurement Group, 1985, pp. 392-399

[4] Colwell, R. P., C. Y. Hitchcock III, E. D. Jensen, H. M. B. Sprunt, and C. P. Kollar, "Computers, Complexity, and Controversy", Computer, September 1985, pp. 8-19

[5] Emer, J. S., and D. W. Clark, "A Characterization of Processor Performance in the VAX-11/780", Proceedings of the 11th Annual Symposium on Computer Architecture, June 5-7, 1984, pp. 301-310

[6] IBM System/370 Principles of Operation, GA22-7000, IBM Corp.

[7] IBM System/370 Extended Architecture Principles of Operation, SA22-7085, IBM Corp.

[8] LaMaire, O. R., and W. W. White, "Processor-Level Workload Characterization", Proceedings of the International Workshop on Workload Characterization of Computer Systems and Computer Networks, (North Holland, to appear), Pavia, Italy, October 1985. Also as IBM Research Report RC11385, IBM Corp., September 1985.

[9] MacDougall, "Instruction-level Program and Processor Modeling", Computer, July 1984, pp. 14-24

[10] Peuto, B. L., and L. J. Shustek, "An Instruction Timing Model of CPU Performance", Proceedings of the 4th Annual Symposium on Computer Architecture, March 23-35, 1977, pp. 165-178

Dynamic Load Sharing in Distributed Database Systems

Philip S. Yu
Simonetta Balsamo
Yann-Hang Lee

IBM Thomas J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Abstract

In this paper, we investigate dynamic load sharing strategies for locally distributed database systems in which the database is partitioned and distributed among multiple transaction processing systems and a common front-end processor is employed for transaction routing. In this environment, if a transaction issues a database request referencing a non-local database partition, the request has to be shipped to the system owning the referenced partition for processing. A total of four different dynamic strategies have been studied. Their performances are compared with that of the optimal static strategy. A dynamic load sharing strategy which takes into account previous transaction routing decisions is found to provide a substantial improvement over the optimal static strategy. The robustness of this strategy is further studied through sensitivity analysis over a wide range of conditions, including transaction load, communication overhead and database reference distribution.

1.0 Introduction

When processing power is distributed over multiple computer systems, load sharing is critical in achieving higher aggregate throughput and better response time. To attain appropriate sharing, arriving tasks are allocated according to some strategy. Strategies are static in nature if allocation decisions are based solely on the static characteristics of arriving tasks and the processing systems. The other category, in which allocation decisions depend upon not only the static characteristics but also the current system state, is referred to as dynamic strategies.

Numerous load sharing strategies, both static and dynamic, have been studied for models of distributed systems. Queueing network analysis, mathematical programming, and other techniques can be used to obtain performance estimates and then to derive optimal static strategies; for example, the optimal deterministic allocation of tasks in [Ston77, Ston78], and optimal probabilistic assignments in [Ni85,Tant85]. On the other hand, certain optimalities in the dynamic approach have also been discovered. For instance, the send-to-shortest queue strategy is found to be the best for the case of Poisson arrivals and identical exponential servers [Wins77] and the round-robin strategy becomes optimal when the queue length at each server can not be observed, provided that all servers have the same initial state [Ephr80]. A survey on load sharing strategies in distributed systems can be found in [Wang85]. For other more complex cases, like heterogeneous servers, multiple classes of tasks and different arrival and service time distributions, simulations have been adopted to study the performance of various strategies [Agra82, Care84, Chow79].

The dynamic load sharing approach takes into account the changing condition of the system state and would appear to be more advantageous than the static approach. However, if a significant amount of overhead is needed to collect the required system state information to make routing decisions, the dynamic strategy can become impractical. The collection, the usages and the impacts of different levels of information are critical issues in the study of dynamic strategies. In [Eagr84, Eagr85], a probe limit scheme based on queue length threshold is proposed to make a compromise between the overhead of collecting information and the amount of information collected. Their results show that using a small amount of state information can improve performance considerably.

In these previous studies, it is assumed that incoming tasks can be serviced completely at any processing system. This implies that either all tasks are purely computational or requested resources, e.g. database or files, are shared or replicated among all processing systems. This assumption can be quite restrictive. Consider a locally distributed database environment as shown in Figure 1.1. By locally we mean so close together that communication delay are negligible, e.g., the entire system is located in the same machine room. The database is partitioned among the various processing systems and a common front-end system is employed for routing transactions to one of the processing systems. If a transaction issues a database request referencing a remote database partition, the request has to be shipped to the system owning the referenced partition for processing. This is referred to as a remote database request. Thus, a new dimension, the reference pattern or the reference locality, has to be considered in load sharing strategies.

In a locally distributed database complex, the processing associated with each transaction can be divided into two categories. The first category denoted as routing dependent processing, is to be executed at the processing system where a transaction is routed to. The application process belongs to this category. The other is partition dependent processing which is a service request against a particular database partition, e.g. the

CH2345-7/86/0000/0675$01.00 © 1986 IEEE

database requests, and can only be executed at the processing system owning the partition. To balance the loads among processing systems, only the routing dependent processing can be used as leverage. In addition, different transaction routing strategies may affect the number of remote database requests and, thus, incur different communication load. Load sharing has become more complicated compared to the case without request shipping between processing systems.

In this paper, we consider four different dynamic strategies for load sharing in a locally distributed database environment and evaluate their impacts on system performance. These strategies require different levels of information on the system state. The usefulness of different levels of state information is quantified through performance comparisons. From simulation results, we notice that good dynamic load sharing strategies should strike a compromise between balancing the processing load and reducing remote database requests. Neglecting either the load condition or database reference information during routing decision making can lead to less than satisfactory results. Also, detailed information on instantaneous system state, e.g. queue length at a particular moment, is found not to be beneficial to improve performance.

In the next section, the locally distributed database environment and transaction characteristics are described. The model of the system complex is also described. In section 3, various load sharing strategies are discussed. Section 4 presents simulation results on response time under different load sharing strategies. Detailed sensitivity analyses are also provided. We summarize the results in Section 5. Note that the model studied is not limited to the database system. It can be applied to any system with distributed resources and requests.

2.0 Model Description and Formulation

We consider a locally distributed transaction processing system as shown in Figure 1.1. The system consists of N transaction processing systems and a front-end system, connected by an interconnection network. The transaction processing systems execute transaction application processes and handle database requests. The whole database is partitioned into N databases which are denoted as $DB_1, ..., DB_N$, where DB_i is attached to the transaction processing system, P_i . All database requests to DB_i are assumed to be handled by the processing system P_i. The processor speeds and the I/O service times at the different processing system are assumed to be identical.

Transactions submitted by users enter the system through the front-end system where transactions get formatted and are routed to one of the processing systems. After a transaction processing is completed, output messages will be mapped into the user's screen format and delivered back to the user via the front-end system. A load sharing strategy is employed at the front-end system to determine the assignment of an incoming transaction to a processing system.

At the assigned processing system, a transaction invokes an application process which may issue a number of database requests. The application process of a transaction will be executed completely at the assigned processing system, whereas database requests will be executed at the processing system owning the database partition. During the execution of database request, I/O devices will be accessed if the required data is not in the processing systems. The flow of transaction processing is shown in Figure 2.1, where a transaction will be completed after several iterations of application processing segments and database requests. Transactions, then, can be characterized into different classes by (1) the processing service demand of each application processing segment, (2) the number and reference distribution of database requests, and (3) the processing and I/O service demands of each database request. For simplicity, we assume that these service demands are exponentially distributed. Also, at the end of each application processing segment, fixed probabilities of issuing a database request to a particular database or terminating the transaction processing are assumed where the probability can depend on the transaction class.

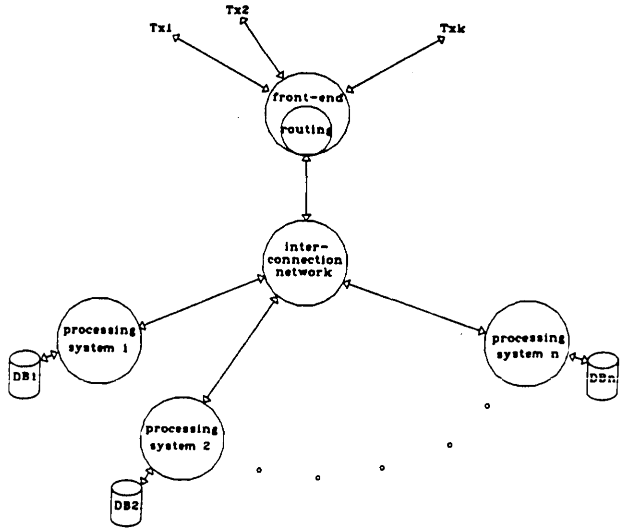


Figure 1.1 The Configuration of a Locally Distributed Database System

Based on the sequence of transaction processing, we construct the model of transaction processing as shown in Figure 2.2. Let there be K transaction classes in the system and let tx_k denote a class k transaction, $k = 1,...,K$. For each k tx_k 's arrive according to a time-invariant Poisson process with rate λ_k, where $k = 1,2,,...,K$. The mean processing service demands of an application processing segment and a database request of tx_k are a_k and b_k, respectively. Both a_k and b_k can be estimated by measuring pathlength of application processing and database request. For each database requests issued by tx_k, we assume that, with a fixed probability p_k^{IO} , an I/O device will be accessed and the service time of each I/O access is exponentially distributed with mean d_k . When the execution of an application processing segment is completed, transaction tx_k may issue a database request to database DB_i with probability p_{ki} , or may terminate with probability p_{k0}. The p_{ki} is referred to as the database request probability of transaction k to DB_i. Thus, the total processing load of incoming transactions per unit of time to the whole system becomes

$$S = \sum_{k=1}^{K} \frac{\lambda_k}{p_{k0}} (a_k + (1 - p_{k0}) b_k)$$

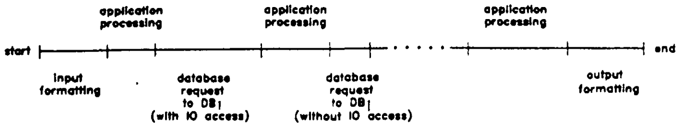


Figure 2.1. The Sequence of Transaction Processing

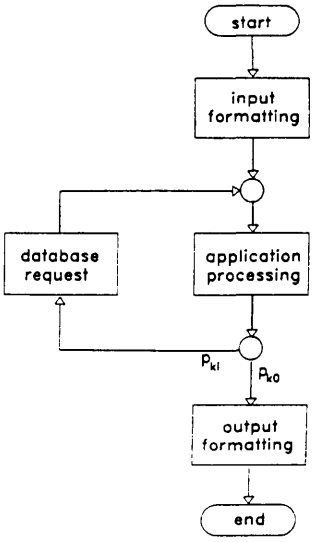


Figure 2.2. Model of Transaction Processing

Among this total processing load, there is a portion associated with the processing of database requests. We denote the processing load of database requests per unit time at P_i as follows

$$S_i^b = \sum_{k=1}^{K} \frac{\lambda_k p_{ki} b_k}{p_{k0}}$$

Notice that S and S_i^b only depend upon the characteristics of transactions and are independent of the transaction routing decisions. When a database request is issued, it must be shipped from processing system P_i to P_j if a transaction being executed at P_i issues a database request to DB_j, where $i \neq j$. After the request gets processed, the result will be sent back. Both P_i and P_j have to perform sending and receiving services in this case. The service demands of sending and receiving a database request or the results of a request are referred to as communication overhead and are assumed to be exponentially distributed with mean c.

The system model is illustrated in Figure 2.3. Each processing system is modeled by a single server processor sharing queue to represent the processor and an an infinite server queue to represent the I/O. After a transaction is routed to a processing system, its various tasks may be executed sequentially at different processing systems. The transmission delay of shipping database requests in the network and the delay due to the decision process at the front-end system are assumed negligible. The former assumption while reasonable in a locally distributed system would not be in a geographically distributed system.

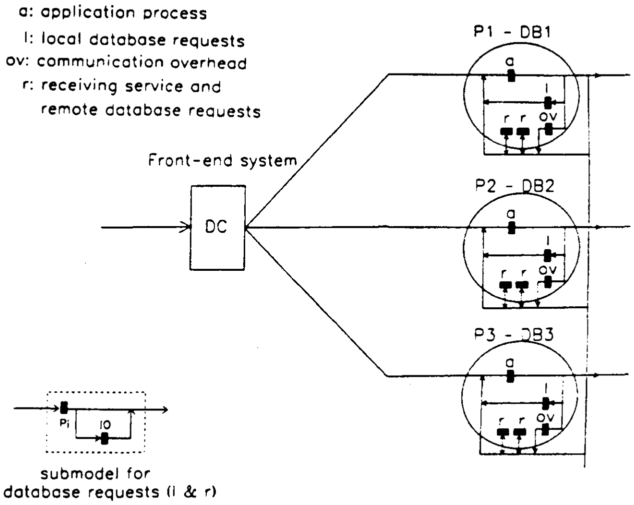

a: application process
l: local database requests
ov: communication overhead
r: receiving service and remote database requests
Front-end system
DC
P1 - DB1
P2 - DB2
P3 - DB3
a
l
r r ov
P_i
IO
submodel for database requests (l & r)

Figure 2.3. Model for Locally Distributed Database System.

3.0 Load Sharing Strategies

We now consider four different dynamic routing strategies which can be employed at the front-end system. Different system static and state information are considered in making the routing decision under each strategy. We call the number of tasks being executed at processing system P_i the queue length at i, where a task is either application processing, database request processing or communication overhead. The number of tasks receiving IO service at system i is not included in the queue length at i.

3.1 Minimum Instantaneous Queue Length

The minimum instantaneous queue length strategy (denoted as MQL) routes each arriving transaction to the processing system that has the least number of tasks being executed. Let $\underline{n} = (n_1,...,n_N)$ where n_i is the instantaneous queue length of the processing system P_i, for $i = 1,2,..,N$. The minimum queue length strategy selects the processor element P_i such that $n_i = \min_{1 \leq j \leq N}\{n_j\}$. If the minimum is not unique, a processing system, among the ones that achieve the minimum, is randomly chosen. This strategy is only based on the state information $\underline{n}$ and does not require any static information.

The minimum instantaneous queue length strategy, also known as the send-to-shortest queue policy, is optimal for a system with identical exponential queues, a single Poisson arriving transaction stream and where each transaction executes on only one processor and can executed on any processor. For more complex systems with multiple classes and/or transactions which cannot execute on certain processor and/or transactions which must execute on more than one processor, the minimum queue length strategy is not in general optimal.

3.2 Minimum Expected Response Time Strategies

p. Consider a set of dynamic strategies where the routing decision process consists of two steps. First, the expected response time of the incoming transaction given the transaction is routed to the processing system P_i, is estimated. Then, the incoming transaction is routed to the processing system P_i which provides the minimum expected response time.

Assuming the system state on arrival is known, the expected response time of the incoming transaction should be estimated using a transient analysis. For simplicity, a steady-state analysis is applied. Consider the system model shown in Fig. 2.3. Let us assume that the class k transactions are routed, according to a certain probability distribution, to the processing system P_i. Then the system can be represented by a product form open queueing network [Bask75] with K chains corresponding to the transaction classes and N processor sharing queues representing the processors, each connected with an infinite server queue representing the I/O servers. If the steady state queue length $\bar{L} = (L_1,...,L_N)$, where L_i denotes the mean queue length of P_i, for $i = 1,...,N$, is known, then the mean response time of an arriving type k transaction, denoted as $R_{ki}(\bar{L})$, provided that it will be routed to P_i, can be approximated as follows.

Let $R^a_{ki}(L_i)$ denote the expected response time for completing the application process for a class k transaction, tx_k, which is routed to a processing system P_i. Consider the processing time of a database request at a processing system given there is no I/O access. The expected response time of serving a local database request to DB_i at P_i, for a transaction tx_k routed to P_i, is denoted by $R^l_{ki}(L_i)$. For a database request to DB_j issued from P_i, where $i \neq j$, a database request must be shipped from processing system P_i to P_j. Then, in this case, there are two components in the expected response time: the first, $R^{ov}_{ki}(L_i)$, is due to the communication overhead in P_i of shipping the request and receiving the result, and the second, $R^r_{kj}(L_j)$, is due to the servicing a remote database request in P_j, which includes receiving the request, processing the request and sending the result back. Hence, the total expected response time $R_{ki}(\bar{L})$, for an arriving type k transaction assigned to processing system P_i can be written as

$$R_{ki}(\bar{L}) = \frac{1}{p_{k0}}[R^a_{ki}(L_i) + p_{ki}\, R^l_{ki}(L_i) + \sum_{\substack{j=1 \\ j\neq i}}^{N} p_{kj}\, R^{ov}_{ki}(L_i) + \sum_{\substack{j=1 \\ j\neq i}}^{N} p_{kj} R^r_{kj}(L_j) + p^{IO}_k d_k] \tag{1}$$

where $p^{IO}_k d_k$ is the expected I/O service time for each database request and p_{kj} is the database request probability of a class k transaction to DB_j.

On the other hand, each component in the expected response time in (1) can be approximated by

$$R^x_{ki}(L_i) = S^x_{ki}\ (L_i + 1) \tag{2}$$

for $k = 1,...,K$, $i = 1,....,N$, where $x = a,l,ov,r$, S^x_{ki} is the service time of tx_k on P_i for service x. Given the mean service demand for tx_k of application processing segments, database requests and sending or receiving requests, i.e. a_k, b_k and c, respectively, one can write

$$S^a_{ki} = \frac{a_k}{\mu}\ ,\quad S^l_{ki} = \frac{b_k}{\mu}\ ,\quad S^{ov}_{ki} = \frac{c}{\mu}\ ,\quad S^r_{ki} = \frac{b_k + c}{\mu} \tag{3}$$

where μ is the processing speed of each processor. Hence, by (2) and (3), the total expected response time given by (1) for transaction tx_k routed to P_i can be rewritten as

$$R_{ki}(\bar{L}) = \frac{1}{p_{k0}}\{(L_i + 1)[S^a_{ki} + S^l_{ki}\, p_{ki} + S^{ov}_{ki} \sum_{\substack{j=1 \\ j\neq i}}^{N} p_{kj}] + \sum_{\substack{j=1 \\ j\neq i}}^{N} (L_j + 1) S^r_{kj}\, p_{kj} + p^{IO}_k d_k\} \tag{4}$$

Note that the above equation is a steady-state expression of the expected response time of the arriving transaction as a function of mean queue length $\bar{L}$. We now consider three different ways to estimate $\bar{L}$ which result in three different routing strategies.

3.2.1 Instantaneous Queue Length Based Strategy

We assume that the instantaneous queue length $\bar{n} = (n_1, ..., n_N)$ is known on arrival and we estimate $\bar{L}$ by $\bar{n}$. i.e., the expected response time is estimated by $R_{ki}(\bar{n})$. Thus, this strategy requires state information $\bar{n}$ and static information $\{a_k, b_k, c, p_{k0}, p_{ki}, \mu\}$. We refer to this strategy as MRT.IQL for minimum response time based on instantaneous queue length. Note that the instantaneous queue length information required can be costly to collect. In addition, the instantaneous queue length may not be representative of the mean queue length.

3.2.2 Service Time Based Strategy

Consider the moment that a new transaction arrives. Let m_{ki} be the number of class k transactions assigned to the processing system P_i, and not yet completed, i.e., m_{ki} is determined by the previous routing decisions for all transactions still in the complex. Let $prob_k(i,j)$ denote the probability that a class k transaction assigned to processing system P_i is waiting for or receiving processing service at P_j. Hence, we can express the expected queue length L_j of P_j as

$$L_j = \sum_{i=1}^{N} \sum_{k=1}^{K} m_{ki}\, prob_k(i,j) \tag{5}$$

for $j = 1,...,N$. Assuming that the fraction of time that a type k transaction is either (a) waiting or receiving processing service at P_j, or (b) receiving I/O service is proportional to the corresponding overall service time, one can approximate the unknown probabilities in (5) as follows

$$\begin{aligned} prob_k(i,j) &= S^r_{kj}\, p_{kj} \frac{1}{C_{ki}} \qquad i \neq j \\ prob_k(i,i) &= [S^a_{ki} + S^l_{ki}\, p_{ki} + S^{ov}_{ki} \sum_{\substack{j=1 \\ j\neq i}}^{N} p_{kj}] \frac{1}{C_{ki}} \end{aligned} \tag{6}$$

where the normalizing constant C_{ki} is given by

$$C_{ki} = S^a_{ki} + S^l_{ki}\, p_{ki} + S^{ov}_{ki} \sum_{\substack{j=1 \\ j\neq i}}^{N} p_{kj} + \sum_{\substack{j=1 \\ j\neq i}}^{N} S^r_{kj}\, p_{kj} + p^{IO}_k d_k \sum_{j=1}^{N} p_{kj}$$

Notice that the probabilities $prob_k(i,j)$ are normalized by considering the transactions assigned to P_i and receiving either processing or I/O service. the I/O devices.

Hence, given $\vec{m} = (m_{ki}: k = 1,...,K;\ i = 1,...,N)$ and the approximate probabilities $prob_k(i,j)$ based on the service time proportions, one can derive estimates from Eq. (5) of the mean queue length $\vec{L}$. Each incoming transaction is then routed to the processing system P_i such that $R_{ki}(\vec{L}) = \min_{1 \le j \le N}\{R_{kj}(\vec{L})\}$. This strategy is called MRT.ST which requires state-dependent information $\vec{m}$ and static information $\{a_k, b_k, c, p_{k0}, p_{ki}, \mu\}$.

3.2.3 Residence Time Based Strategy

The residence time based strategy, which will be called MRT.RT, is different from MRT.ST only in the computation of probabilities $prob_k(i,j)$. With MRT.RT strategy, the probability $prob_k(i,j)$ is proportional to the residence time at the processing system P_j, instead of the service time. An iterative approach based on the MVA equations [Reis80] is applied in order to derive the probability $prob_k(i,j)$ and the expected queue length, $\vec{L}$.

The residence time $T_k(i,j)$ in P_j of a transaction of class k initially assigned to P_i can be written as

$$T_k(i,j) = S_k(i,j)\ (N_k(i,j) + 1)$$

where $N_k(i,j)$ is the mean number of tasks that a type k transaction assigned to P_i finds in P_j and $S_k(i,j)$ is its total service time at P_j. Values of $S_k(i,j)$ can be simply derived from the service times expressed in (3). $N_k(i,j)$ can be approximated by $N_k(i,j) = L_j - prob_k(i,j)$ which is similar to the well known Bard-Shweitzer's algorithm [Schw79, Bard80]. Thus, the probability $prob_k(i,j)$ which is proportional to the residence time $T_k(i,j)$ is given as

$$prob_k(i,j) = S^r_{kj} p_{kj} (L_j - prob_k(i,j) + 1)\frac{1}{C_{ki}} \qquad (7)$$

$$prob_k(i,i) = [S^a_{ki} + S^l_{ki} p_{ki} + S^{ov}_{ki} \sum_{\substack{j=1 \\ j \ne i}}^{N} p_{kj}]\ (L_i - prob_k(i,i) + 1)\frac{1}{C_{ki}}$$

where the normalizing constant C_{ki} is given by

$$C_{ki} = [S^a_{ki} + S^l_{ki} p_{ki} + S^{ov}_{ki} \sum_{\substack{j=1 \\ j \ne i}}^{N} p_{kj}](L_i - prob_k(i,i) + 1) + \sum_{\substack{j=1 \\ j \ne i}}^{N} S^r_{kj} p_{kj} (L_j - prob_k(i,j) + 1) + p_k^{IO} d_k \sum_{j=1}^{N} p_{kj}$$

Hence, given a system state $\vec{m}$, the mean queue lengths are computed by iterating (7) and (5), starting with zero values for both queue lengths L_j and probabilities $prob_k(i,j)$, for $k = 1,...,K$, $i,j = 1,...,N$. The MRT.RT strategy uses the same static and state-dependent information as the MRT.ST strategy. The iteration between (7) and (5) should not impose any substantial overhead during decision making.

4.0 Simulation Study and Performance Comparisons

In the following, we use simulation to investigate the effectiveness of the proposed load sharing strategies. Mean transaction response time is the main concern and is used to indicate system performance under the different load sharing strategies. To compare the performance of tne above strategies, we also consider the optimal static load sharing strategy. Under a static load sharing strategy, an incoming transaction is routed to a processing system according to a predefined routing probability. The optimal static load sharing strategy takes account of all static information, i.e. $\{\lambda_k, a_k, b_k, c, p_{k0} p_{ki}, \mu\}$, to determine the routing probabilities such that the mean transaction response time is minimized. The details of solving this optimization problem have been given in [Yu85a], where a simplex reflection method is used to find the optimal routing probabilities.

4.1 Description of the Simulations

In order to evaluate dynamic load sharing strategies, we simulated the model for a locally distributed database system illustrated in Figure 2.3. The simulation was implemented using RESQ [Saue82]. The routing decision is implemented as a separate function and is invoked upon transaction arrival at the front-end system. In addition, for all simulation runs, 95% confidence level was obtained. The simulations were run until the relative width of the confidence interval (width divided by midpoint) was less than 0.1.

In the experiments reported in the following, we assume that there are three transaction processing systems ($N = 3$) and three transaction classes ($K = 3$). Based on data from some IBM IMS systems [Corn85, Yu85b], the average number of database requests per transaction is set to 15 for all transaction classes, i.e., $p_{k0} = 0.0625$ for $k = 1,2,3$. The matrices $\frac{1}{1-p_{k0}}[p_{ki}]$, which indicate the distribution of database requests, are given in Table 4.1 to reflect low, middle, and high localities of database requests.

To study the impact of processing load on database requests and request shippings, various service demands and processing powers are assigned with relative values. We regard $a_k + b_k$ as a unit of service demand for all k. The mean service time of this unit is assumed to be 0.04, i.e., $\mu = 25$. The service demand of shipping a request, c, is defined in terms of this unit. The IO access time, d_k, and the probability of having IO access, p_k^{IO}, are assumed to be 0.4 and 0.7 for all transaction classes, respectively. Also, we denote the ratio of b_k to $a_k + b_k$ as r_k. The complexity of database requests can be represented through this ratio.

The load of a processing system could be due to the services of transaction application processing, database request processing, and communication overhead processing. Let $\rho_p = \frac{S}{N\mu}$, where S is given in Section 2, which indicates the average processing load per system due to application processing and database requests and is independent of the routing decisions. By changing the arrival rates λ_k, we can study the transaction response time under different ρ_p. The load of a processing system P_i due to processing database requests, de-

	low locality			middle locality			high locality		
Database	1	2	3	1	2	3	1	2	3
Tx type 1	0.65	0.20	0.15	0.75	0.11	0.14	0.90	0.10	0.0
Tx type 2	0.17	0.52	0.31	0.07	0.82	0.11	0.07	0.87	0.06
Tx type 3	0.21	0.21	0.58	0.11	0.06	0.83	0.11	0.03	0.86

Table 4.1 The Distribution of Database Requests Used in Experiments

noted by $\rho_b(i)$, is routing independent and is equal to $\frac{S_i^b}{\mu}$, where S_i^b is given in Section 2. By changing the arrival rates subject to a fixed ρ_p, we can vary the relative value of the $\rho_b(i)$ which represent the partition depending processing loads.

4.2 Performance Comparisons

First, we study the effectiveness of load sharing strategies under different processing loads, ρ_p. The incoming transactions are assumed to have middle locality in regard to the distribution of database requests and $r_k = 0.3$ for all transaction classes. The partition dependent load on each processing system is assumed to be equal, i.e., $\rho_b(1):\rho_b(2):\rho_b(3) = 1:1:1$. Figures 4.1 and 4.2 show the transaction response times versus ρ_p for $c = 0.05$ and $c = 0.25$, respectively.

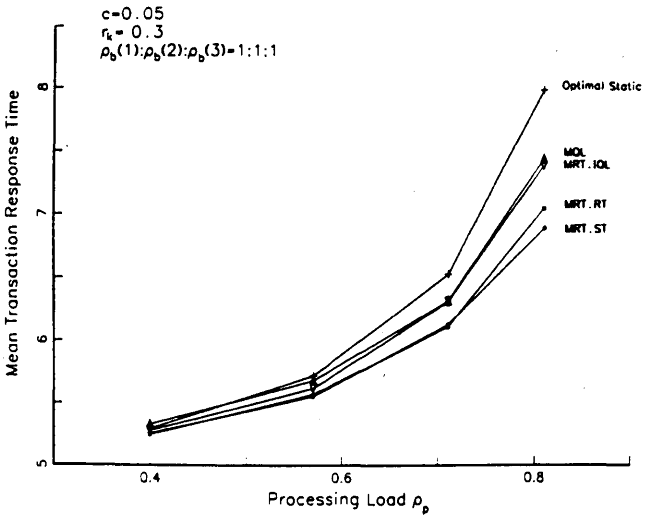


Figure 4.1 Mean transaction response time vs. processing load ($c = 0.05$)

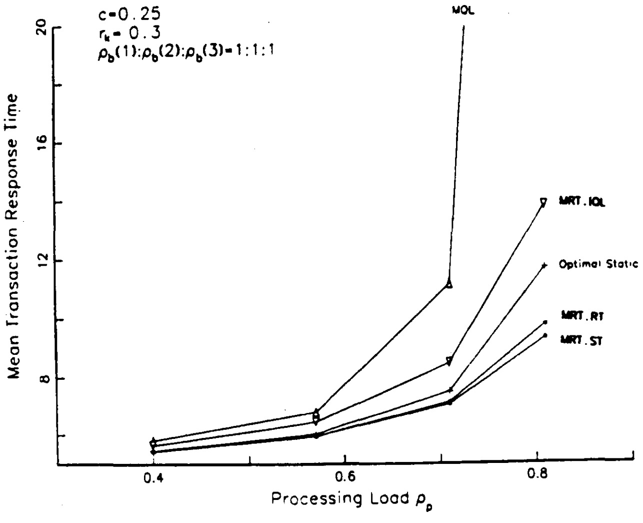


Figure 4.2 Mean transaction response time vs. processing load ($c = 0.25$)

Notice that when the communication overhead of shipping remote requests is low, all dynamic load sharing strategies have better performance than the optimal static strategy. When the communication overhead is high, optimal static strategy becomes better than MQL and MRT.IQL, strategies. Over all strategies considered, MRT.ST and MRT.RT strategies yield the best performance. Neither of these two strategies needs instantaneous state information. If we keep a count at the front-end system of the previous transaction routings and completions, both strategies can be implemented without the extra communication between the front-end system and the transaction processing systems that would be needed to collect instantaneous state information.

Both MQL and MRT.IQL strategies use instantaneous queue lengths to make transaction routing decisions. The queue length at each processing system contains the tasks of application processing, database request processing and communication overhead. The instantaneous queue length may fluctuate frequently due to the issuing of database requests. As a consequence, the decision processes using MQL and MRT.IQL are based on more instance where the system has unbalanced load than under the other strategies. Thus, more transactions are routed to a non-preferred system where only small percentage of database requests are designated to and more remote database requests are introduced. When the communication overhead is high, these additional remote database requests lead to an increase of system utilization and transaction response time. Both the MQL and MRT.IQL strategies end up with inferior performance to the optimal static strategy, which does not consider the system state in making the routing decision.

4.3 Sensitivity Analyses

In Figures 4.1 and 4.2, it has been shown that both MRT.ST and MRT.RT are superior to the other load sharing strategies considered in this paper. We shall proceed further to investigate the robustness of these two strategies through sensitivity analyses on various system and workload parameters. Specifically, we vary communication overhead, distribution of database requests and degree of balance of the partition dependent loads.

1. Sensitivity to communication overhead

The relationship between transaction response time and communication overhead is illustrated in Figure 4.3 where simulation results for various communication overheads are presented for MRT.ST, MRT.RT, and the optimal static strategies. Two groups of curves are shown; one is for $\rho_p = 0.57$, the other is for $\rho_p = 0.71$. The response times increase more than linearly in the second group which is under higher processing load. Figure 4.4 shows the increase in mean communication load at each processing system when the communication overhead of shipping database requests and results becomes large.

Although both MRT.ST and MRT.RT are superior to the optimal static strategy, the mean communication load under the optimal static strategy is less than that under MRT.ST or MRT.RT. This is shown in Figure 4.4. Compared with the optimal static strategy, the larger shipping loads under MRT.ST and MRT.RT indicate that more transactions are routed to a non-preferred processing system to eliminate temporary load unbalancing. Despite the increase in communication load, the dynamic routing balances the loads among processing systems, eliminates possible bottlenecks, and thus reduces the response time. The other interesting observation in Figure 4.4 is that the number of non-preferred routings under MRT.RT is greater than that under MRT.ST. In contrast to MRT.ST, the MRT.RT's approximation captures queueing effects. Thus, the difference of estimated queue lengths between processing systems under MRT.RT tends to be larger than that under MRT.ST. As a consequence, more transactions are routed to a non-preferred processing system under MRT.RT.

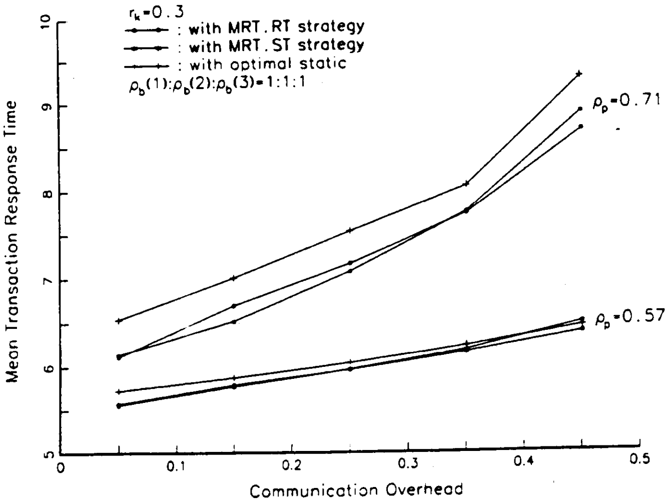


Figure 4.3. Mean transaction response time vs. communication overhead

2. Sensitivity to the degree of balance of database request processing

In Figure 4.5, the performance of MRT.ST, MRT.RT and the optimal static strategy is studied for different proportions of $\rho_b(1):\rho_b(2):\rho_b(3)$ which is changed from 0.5:1:1 to 2.5:1:1 . This is achieved by changing the arrival rates of different transaction classes. Since the application processing load can be allocated to any processing system, the changes of arrival rates only vary the load of serving database requests. The load of serving application processing and communication at each processing system is determined by the load sharing strategy. However, this underlying unbalancing certainly increases the difficulty in trading off sharing transaction processing load and reducing communication overhead load. As shown in Figure 4.5, when the $\rho_b(i)$'s are unbalanced, the response time increases apparently because (1) more contention for one or two processing systems when database requests are issued, and (2) more communication load is introduced during balancing the load at each processing system. In addition, the MRT.ST strategy is much more sensitive to the degree of balancing of $\rho_b(i)$.

The unbalanced loads in the processing systems can also be used to differentiate the performances under MRT.ST and MRT.RT. Since the routing decision is based on the difference of estimated loads between processing systems, the accuracy in estimation only becomes clear when the loads are quite different. As shown in Figure 4.5, the performance under MRT.RT can be a lot better than that under MRT.ST for highly unbalanced cases. When $\rho_b(1):\rho_b(2):\rho_b(3)$ = 2.5:1:1, the response time under the MRT.ST gets even worse than that under the optimal static strategy.

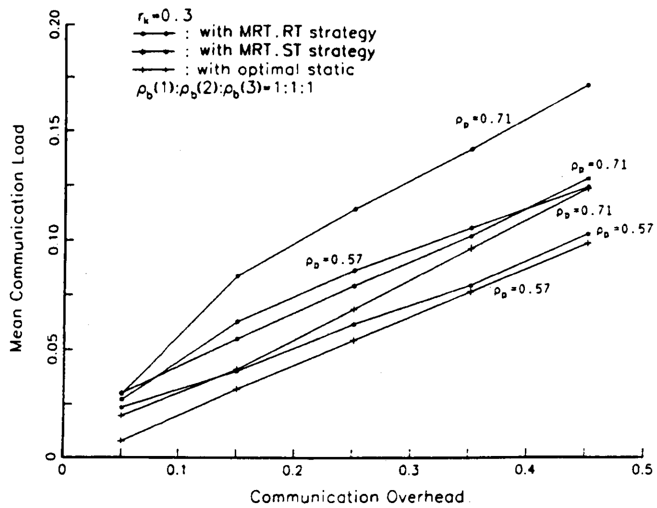


Figure 4.4. Average communication load vs. communication overhead

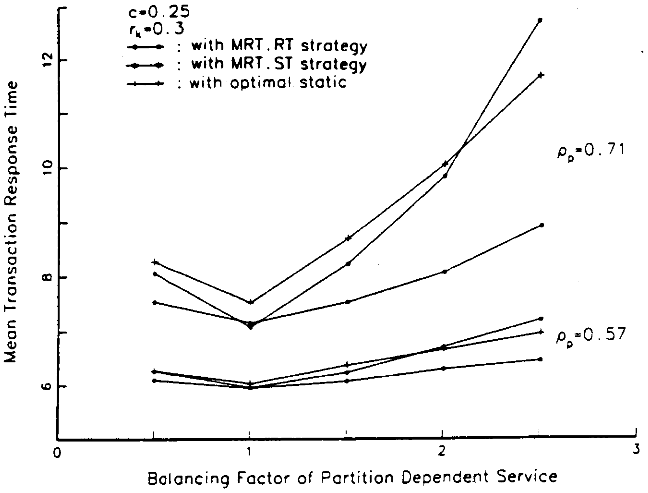


Figure 4.5 Mean transaction response time vs. degree of balancing on partition dependent load

3. Sensitivity to the database reference pattern

Under the MRT.RT strategy, Figure 4.6 shows the mean response time versus processing load for the different reference distributions of database requests defined in Table 4.1. As expected, the cases with less locality have longer response times. When ρ_p = 0.81 and c = 0.25, the processor utilization of the low locality case can reach 0.99. The decrease in the locality leads to higher communication load. When the locality of database request decreases, the benefit of routing transactions to the preferred processing system decreases. Thus, more transactions get routed to a non-preferred processing system to balance the load. Figure 4.7 shows the percentage of transactions routed to non-preferred processing systems versus processing load. Clearly, the percentage of non-preferred routing in the case with low locality is larger than that of the cases with middle or high locality. The changes in transaction routing under different processing load can be studied in Figure 4.7. When the utilization increases, the communication load incurred from assigning transactions to a non-preferred processing system may deteriorate transaction response time more than the gain from balancing the load. Thus, this percentage will be reduced to avoid system saturation.

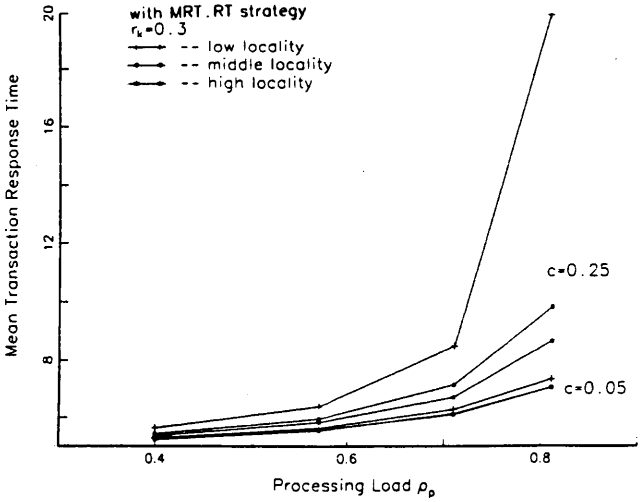


Figure 4.6. Mean transaction response time vs. processing load

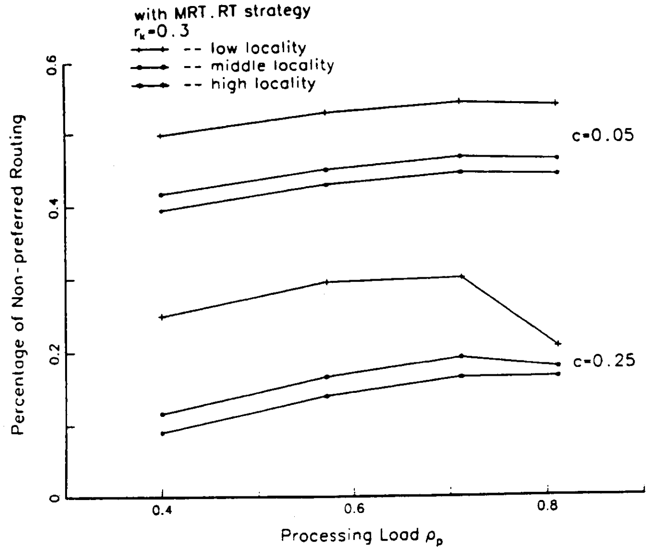


Figure 4.7. Non-preferred routing frequency vs. processing load

5.0 Conclusion

In this paper, we have evaluated different strategies for dynamic load sharing in a locally distributed database environment. The issue is how to make effective load sharing decisions at the front-end system with the least amount of communication between the front-end and processing systems to collect system state information. A total of four different dynamic strategies have been considered. The minimum queue length (MQL) strategy which does not consider the transaction's database reference pattern has the worst performance among all the dynamic strategies considered. It often performs worse than the optimal static strategy. The dynamic strategy which uses the instantaneous queue length to estimate the mean response time of each possible routing is also not very successful. This is due to the fact that instantaneous queue length does not accurately reflect the current loads of processing systems.

The other two dynamic strategies, MRT.ST and MRT.RT, estimate the mean response time at each processing system based on the number of executing transactions at each processing system. Both strategies lead to very satisfactory results when the partition dependent load is well balanced. The difference between the two strategies is in the estimation of the mean queue length at each processing system: the MRT.ST is simply based on the remaining processing load, whereas the MRT.RT further considers the queueing effect. The sensitivity analyses show that the MRT.RT is much more robust for unbalanced database load and can provide substantial performance improvement over the optimal static load sharing strategy.

In summary, dynamic strategies can be superior to the optimal static strategy if good routing decisions are made using appropriate information about the system state. Good dynamic strategies need to consider both balancing the load and reducing remote processing requests. The state information used in making decisions must be able to characterize the average system behavior instead of the instantaneous load fluctuation. An immediate extension of the study, which is currently being investigated, is to consider the collection of the above information and the load sharing strategies for a transaction processing complex with distributed front-end systems.

Acknowledgement

We would like to thank Steve Lavenberg for his comments and suggestions; Douglas Cornell for obtaining the optimal static routing strategy; Edward A. MacNair for assistance with the use of the RESQ package.

References

[Agra82] Agrawala, A. K., Tripathi, S. K., and Ricart, G., "Adaptive Routing Using a Virtual Waiting Time Technique," *IEEE trans. on Software Eng.*, Vol. SE-8, No. 1, Jan. 1982, pp. 76-81.

[Bard80] Bard, Y., "A Model of Shared DASD and Multipathing," *Comm. of the ACM*, Vol. 23, No. 10, (Oct. 1980), pp. 564-572.

[Bask75] Baskett, F., Chandy, K. M., Muntz, R. R., Palacios, F., "Open, Closed and Mixed Networks of Queues with Several Classes of Customers", *Journal of ACM*, Apr. 1975, pp. 248-260.

[Care84] Carey, M. J., Livny, M., and Lu, H., "Dynamic Task Allocation in a Distributed Database System," Computer Science Technical Report 556, University of Wisconsin-Madison, Sep. 1984.

[Chow79] Chow, Y-C. and Kohler, W. H., "Models for Dynamic Load Balancing in a Heterogeneous Multiple

Processor System," *IEEE Tran. on Computers,* Vol. C-28, No. 5, (May 1979), pp. 354-361.

[Corn85] Cornell, D.W., Dias, D.M., and Yu, P.S., "Analysis of Multi-system Function Request Shipping", IBM Research Report, RC11154, Yorktown Heights, NY (May 1985). Also, to appear in *2nd Int'l. conf. on Data Engineering,* Feb. 1986.

[Eagr84] Eager, D.L., Lazowska, E.D. and Zahorjan, J., "Adaptive Load Sharing in Homogenous Distributed Systems", Technical Report 84-10-01, Department of Computer Science, University of Washington (Oct. 1984).

[Eagr85] Eager, D.L., Lazowska, E.D. and Zahorjan, J., "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing", *Performance Evaluation Review,* Vol. 13, No. 2 (Aug. 1984), pp. 1-3.

[Ephr80] Ephremides, A., Varaiya, P. and Walrand, J., "A Simple Dynamic Routing Problem," *IEEE Trans. on Automatic Control,* Vol. AC-25, No. 4, Aug. 1980, pp. 690-693.

[Ni85] Ni, L. M. and Hwang, K., "Optimal Load Balancing in a Multiple Processor System with Many Job Classes," *IEEE on Soft. Eng.,* Vol. SE-11, No. 5, May 1985, pp. 491-496.

[Reis80] Reiser, M., Lavenberg, S. S., "Mean-Value Analysis of Closed Multichain Queueing Networks" *Journal of ACM,* Apr. 1980, pp. 313-322.

[Saue82] Sauer, C. H., MacNair, E. A., and Kurose, J. F., "The Research Queueing Package Version 2: Introduction and Examples," IBM Research Report, RA-138, Yorktown Height, NY, 1982.

[Schw79] Shweitzer, P., "Approximate Analysis of Multiclass Queueing Networks of Queues", *Int. Conf. on Stochastic Control and Optimization* North Holland, Amsterdam, 1979.

[Ston77] Stone, H. S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithm," *IEEE Trans. on Soft. Eng.,* Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

[Ston78] Stone, H. S., "Critical Load Factors in Two Processor Distributed Systems," *IEEE Trans. on Soft. Eng.,* Vol. SE-4, No. 3, May 1978, pp. 254-258.

[Tant85] Tantawi, A. N. and Towsley, D., "Optimal Static Load Balancing in Distributed Computer Systems," *Journal of ACM,* Apr. 1985, pp. 445-465.

[Wang85] Wang, Y.-T., and Morris, R.J.T., "Load Sharing in Distributed Systems", *IEEE Trans. on Computers,* Vol. C-34, No. 3, (March 1985), pp. 204-217.

[Wins77] Winston, W., "Optimality of the shortest line Discipline," *J. Appl. Prob.,* Vol. 14, 1977, pp. 181-189.

[Yu85a] Yu, P.S., Cornell, D.W., Dias, D.M., and Thomasian, A. "On Coupling Partitioned Database Systems" IBM Research Report, RC-11410, Yorktown Heights, NY, 1985.

[Yu85b] Yu, P.S., Dias, D.M., Robinson, J. T., Iyer, B. R., and Cornell, D. W., "Modelling of Centralized Concurrency Control in a Multi-System Environment," *Performance Evaluation Review,* Vol. 13, No. 2, 1985, pp. 183-191.

A LOAD INDEX FOR DYNAMIC LOAD BALANCING (*)

Domenico Ferrari and Songnian Zhou

Computer Systems Research Group, Computer Science Division
Department of Electrical Engineering and Computer Sciences and
the Electronics Research Laboratory, University of California, Berkeley

ABSTRACT

The problem of selecting the load index or indices to be used in dynamic load balancing policies is discussed. One such index, based on a mean-value equation, is proposed, and its main characteristics investigated. The index is obtained assuming that the goal of the load balancing scheme is the minimization of the response time of the user command being considered for possible remote execution. The main advantages of the proposed index are its being based on a mathematical foundation and on a clearly specified criterion, the simplicity of its calculation, and its suitability for use in a heterogeneous network. The main problems with it are its dependence on the command's arguments and the steady-state assumption on which it is based. However, it can be argued that, in most practical cases, the index does not have to be used in its full generality, but reduces to simpler expressions for which the former problem does not arise. The results of some experiments to study the latter problem are presented and discussed.

1. INTRODUCTION

One of the most important potential benefits of loosely-coupled distributed systems is in the area of resource sharing. By interconnecting a number of machines via a data communications network with an adequate bandwidth, a larger variety and a larger number of hardware and software resources can be made available to the users of the resulting distributed system than is usually possible in a centralized system. The processing powers of the hosts in a distributed system are among the sharable resources, and are indeed made available to remote users in most such systems, sometimes via a *remote login* mechanism, some other times by allowing users to subdivide the work to be accomplished between a foreground machine and a background "number cruncher," and some other times by having on the network a pool of "public" machines (compute servers) accessible on demand to the users.

In spite of these mechanisms and provisions, we cannot state that the processing resources of a distributed system are always shared as much as they could and should be. This unsatisfactory state of affairs is particularly noticeable, and particularly unfortunate, in many local-area network (LAN) based distributed systems, where the small intercomputer distances, the relatively broad bandwidths, and the greater homogeneity of host ownership (which is frequently restricted to a single organization) would make it easier and more rewarding to share the processing resources of the hosts among all or most of the system's users. The dramatic imbalances among the loads of the various hosts we often observe in these systems cause poor performance and a waste of system resources. In certain LAN-based installations, both the workload imposed on the system by each user and the set of active users have largely predictable and not too rapidly changing characteristics; under these conditions, the *manual* (or *static*) approach to load balancing, which consists of distributing the users over the available hosts (one of them being defined as the "usual host" of each user), may be quite successful. Since no workload is absolutely constant in its volume and characteristics, it will be necessary to rebalance the loads periodically; that is, we will have to retune the system when its operating point has gone far enough from the point of balance. However, a large fraction of the LAN-based installations are characterized by a workload so dynamic that the maximum frequency at which the load can be rebalanced manually is too low for manual rebalancing to be effective. As in the case of the bottlenecks that are found in centralized systems, the initial balancing and periodic retuning with the manual approach may be useful; however, shorter-lived bottlenecks, whose impact on performance grows with the width of the workload's frequency spectrum, can only be eliminated by *automatic* (or *dynamic*) approaches[1]. In principle, providing the users with such mechanisms as a *remote login* and a network-wide load reporting command allows them to give their individual contributions to the balancing of the loads. This may, however, be ineffective and even counterproductive, due to the frequency of the possible interventions, which is likely to be too low, and to the necessarily limited and incomplete information that each individual user has about the system's state. Thus, for a large number of installations, dynamic load balancing is required to improve resource usage and performance.

This paper discusses the problem of selecting the load index or indices to be used in dynamic load balancing policies, and proposes one such index. Section 2 defines the load balancing schemes the rest of the paper refers to. Our criteria for load index selection and our assumptions are presented in Section 3. The specific index we propose in this paper is introduced in Section 4, together with the arguments for its

(*) The research reported herein was partially supported by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under Contract No. N00039-84-C-0235, and by the National Science Foundation under Grant DMC-8503575. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, of the National Science Foundation, or of the U.S. Government.

CH2345-7/86/0000/0684$01.00 © 1986 IEEE

choice. Section 5 presents the results of experiments performed to verify the validity of some of the assumptions on which the new index is based. The main characteristics of the index and the major problems that arise with its use in a load balancing scheme are discussed in Section 6.

2. DYNAMIC LOAD BALANCING POLICIES

Since the terminology used in the load balancing literature is still fluid, we must provide our definitions of the terms we will use, and clarify, by introducing a classification of schemes as well as the various assumptions we are making, the scope of our investigation. This will be done both in this and in the next section.

First, we notice that the two terms *load balancing* and *load sharing* very often appear in the literature with the same meaning. One could easily introduce a distinction between them based on the different meanings the terms "balancing" and "sharing" suggest: for instance, use of the term "balancing" could be restricted to those schemes whose objective is to keep the loads on the machines within a relatively narrow band around the instantaneous average, whereas "sharing" could refer to those schemes in which a machine sends some of its load away (or accepts some of the load of other machines) only when its load goes beyond an upper threshold (or falls below a lower threshold). In both types of schemes the decision-maker must know the load existing on the machine being considered; in the load balancing ones, also the current average system load[2] must be known. However, even though drawing a distinction between "balancing" and "sharing" may be useful in certain contexts, both types of schemes are dealt with in the same way in this paper. We therefore use "load balancing" as a generic term encompassing both, even though our primary objective in selecting a load index is not that of equalizing the loads (we shall indeed see that, with our approach, this objective would be meaningless).

It is also useful to distinguish *preemptive* from *non-preemptive* load balancing schemes. In the former, a running process may be suspended and migrated to a remote machine, where its execution will resume from the point of suspension. A non-preemptive scheme is one in which a process is assigned to a machine before beginning its execution, and cannot be moved to another after its execution has begun. We shall usually refer to non-preemptive schemes in the sequel, though most of our considerations apply to the preemptive ones as well. In either type of scheme, the *local machine* is the machine at which a given process entered the system.

Another classification of load balancing schemes that is sometimes useful is the one based on the identity of the machine which takes the initiative. When the scheme involves a centralized controller that makes the placement decisions, this controller can be thought of as the initiative-taker as well as the decision-maker, even though it may be a prospective *source* or *destination* machine which takes the initiative and asks the controller to decide. In other schemes (the so-called *sender-initiated* ones[3]), the local machine takes the initiative when a new process is to be executed or when its load has gone beyond the upper threshold. In the *receiver-initiated* schemes[3], instead, underloaded prospective receivers take the initiative of broadcasting information about their enviable state so as to attract currently running or soon-to-arrive new processes.

Thus, the initiator will have to select senders and receivers in the centralized controller case, one of the eligible receivers in sender-initiated schemes, and one of the eligible senders in receiver-initiated schemes. This selection can be either load-independent or load-dependent.

Among the *load-independent* policies are Random[4] or Random Splitting[5], which chooses at random the destination of a process to be executed remotely; Fixed Destination or Fixed Source[5], which, in the sender-initiated version, statically binds a given receiver to a given sender or group of senders, or, in the receiver-initiated version, a given sender to a given receiver or group of receivers; Cyclic Splitting or Cyclic Service[5], in which the destination (respectively, the source) is selected according to a cyclic ordering of the available machines; and Proportional Branching[6], a probabilistic policy which selects destinations or sources according to probabilities proportional to their processing speeds.

Examples of *load-dependent* policies include Lowest Load[6,7,8], which selects the machine with the smallest load at the time the decision is to be made; Shortest[4], which does the same but restricts its search to a randomly chosen subset of the eligible machines; Threshold[4], which probes randomly selected machines to determine whether adding a process would raise their loads above a given threshold; Broadcast When Idle or Poll When Idle[7], a receiver-initiated policy in which a lightly loaded machine invites or polls the others, so that only those prospective senders whose loads are heavier than a threshold will ship processes to the initiating machine (note that, if priority is given to the most heavily loaded machine, we obtain the Highest Load policy, i.e., the receiver-initiated analog of Lowest Load); Neighbor Pairing[9], which shifts load from the more heavily to the less heavily loaded machine in dynamically defined pairs of machines; and the policy used in the MOS operating system[10], which cyclically selects machines among the lightly loaded ones in a subset of the eligible machines.

While in load-dependent policies the load of each machine in the system is to be measured, and its value must be known by at least some of the possible decision-makers, load-independent policies do not have such a requirement. However, as was observed at the beginning of this section, at the very least a policy must rely on the knowledge of the load on the local machine, as moving processes is not a zero-cost operation and should be done only when necessary. Thus, the load of each machine must be measured, i.e., quantitatively expressed, in all cases. What index (or indices) should be used to measure a machine's load?

3. A CRITERION FOR LOAD INDEX SELECTION

Many indices have been explicitly or implicitly used in the load balancing literature to express the load existing on a machine at a given time. Examples of such load indices include the utilization of the CPU, the length of the ready queue (in UNIX* terminology, the "load average"), the stretch factor (defined as the ratio between the execution time of a process on a loaded machine and its execution time on the same machine when it is empty), and more complicated functions of these simple variables. However, to the authors' knowledge, a scientific justification for the choice of a load index has never been given. In fact, some papers

* UNIX is a trademark of AT & T Bell Laboratories

even ignore the question altogether, and simply refer to "the load" as if there existed a well-known and universally accepted definition of this term. Is any of the indices used in the literature a valid one? Which? And what does it mean for a load index to be "valid"?

To simplify our discussion, we shall assume that the object to which load balancing applies is the *interactive user command*, as represented by the typing in of a command line or equivalent action on the part of an interactive user. In other words, the execution of a command will be considered atomic from the load balancing viewpoint, even when a command causes the creation and execution of several processes that, in principle, could be executed on different machines. This assumption is made to facilitate the description, but is not essential for the application of our approach.

An assumption that is essential is the choice of *the command's response time* as the performance index to be minimized by the load balancing scheme. This is another area in which some of the papers on load balancing leave something to be desired: the objectives of the schemes proposed therein are sometimes not clearly specified. Our choice of the response time objective is certainly questionable, since the exact relationships between it and such system-wide objectives as throughput maximization are unknown, but (we believe) not unreasonable.

Under these assumptions, a valid load index *li* must be such that the relationship between the response time *rt* of a command and the index is represented by a *single-valued curve*. In other words, *rt* must be a *function* of *li*. The reason for this condition is obvious: if the function *rt*(*li*) for a given machine, a given configuration, and a given command is known, the value of *li* at the time a decision is to be made can be used to predict the response time that the command will have if it will be sent to that machine. The predicted response times for all the eligible machines (including the local one), adjusted for the expected communications delays due to the shipment of the command, of the output, and possibly of the files it needs to execute, can be compared by the decision-maker in order to determine the machine which is likely to process the command in the shortest possible time. Note that this condition, even though it seems not to be very restrictive, is not generally satisfied by any of the indices proposed in the literature. Simple experiments performed by the authors have shown that the curves relating *rt* to CPU utilization, ready queue length, stretch factor, and other indices are multi-valued ones for at least some types of commands.

If more restrictive conditions are imposed, the selection of the least loaded machine can be made more efficient. For instance, in a system consisting of identical machines, a monotonically non-decreasing *rt*(*li*) function would allow the decision-maker to restrict the choice to the local machine and the remote machine with the smallest value of *li*, and to compute the function only for these two machines. Furthermore, if the monotonic function has a known minimum slope or, even better, is linear, then all comparisons will involve only the values of *li*, and no computation of *rt* will have to be performed. These observations can be extended to heterogenous systems, but will apply only to each group of identical machines within them.

Note that our choice of command response time minimization as the objective of load balancing makes it impossible to define the load of a machine as a command-independent quantity. This means that our answer to the question: "How much load is there on this machine?" will be another question, that is: "For what command?". Thus, our approach does not help when the policy is load-independent, unless a standard "basket" of commands is defined to be used in the computation of a command-independent load index for each machine.

4. A LOAD INDEX BASED ON A MEAN-VALUE EQUATION

In this section, we shall propose a load index which, under certain assumptions, satisfies the criterion introduced in the previous section.

Consider a machine M, a command A, and a mix of commands B. Among all the possible loads that M may be processing, consider the following three:

(A) command A *alone*;

(B) mix B (the *background* load);

(C) the *combination* of A and B.

Our problem can now be expressed in these terms: predict the response time of A when load C is running on M from the knowledge of A and of the background load B (the load that was there just before the arrival of A).

We make the assumption that machine M can be accurately modeled by a closed queueing network model having:

(i) R chains;

(ii) L service centers 1, 2, . . . L of the FCFS, PS (processor sharing), LCFSPR (last-come-first-served-preemptive-resume), and IS (infinite servers) types [11]; center 1 is an IS-type service center representing user terminals;

(iii) a fixed number of customers (i.e., commands) in each chain (note that customers never change chain);

(iv) service rates independent of the number of customers at the respective centers;

(v) FCFS centers that are all single-server centers, and have a chain-independent service rate.

The three loads A, B, and C can be modeled as follows:

(a) command A is the only customer in chain 1;

(b) the commands in load B are clustered, and each cluster is represented by one of the chains 2 through R (R is set equal to the number of clusters of B plus 1);

(c) load C is represented by chains 1 through R.

Under these assumptions, the mean-value equation of Corollary 1 of [12] holds for each non-IS center l in such a model:

$$w_{r,l}(\mathbf{K}) = \tau_{r,l}[1 + n_l(\mathbf{K} - \mathbf{e}_r)], \tag{1}$$

where

$\mathbf{K} = (k_1, k_2, \cdots k_r)$: population vector ($k_r$ = population size of chain r);

$\mathbf{e}_r$ = R-dimensional unit vector in direction r;

$w_{r,l}(\mathbf{K})$ = mean time spent by a chain r customer at center l at each visit in the network with population vector $\mathbf{K}$;

$\tau_{r,l}$ = mean service time per visit of a chain r customer at center l;

$n_l(\mathbf{K} - \mathbf{e}_r)$ = mean number of customers (mean "queue length") at center l in the same queueing network with one less customer in chain r.

Note that, for a FCFS center l, we have $\tau_{r,l} = \tau_l$, and that, for an IS center,

$$w_{r,l} = \tau_{r,l}. \tag{2}$$

If the model includes IS centers other than center 1, the corresponding n_l will be defined to be 0.

Denoting by $rt_r(X)$ the mean response time (i.e., the mean time spent outside service center 1) of a chain r command under load X, and by $v_{r,l}$ the mean number of visits a chain r customer makes to service center l, we can write for command A

$$rt_1(A) = \sum_{l=2}^{L} v_{1,l}\tau_{1,l}, \tag{3}$$

and for load C

$$rt_1(C) = \sum_{l=2}^{L} v_{1,l}\, w_{1,l}(\mathbf{K}). \tag{4}$$

Substituting (1) into (4), and using (3), we obtain

$$rt_1(C) = \sum_{l=2}^{L} v_{1,l}\tau_{1,l} + \sum_{l=2}^{L} v_{1,l}\tau_{1,l}\, n_l(\mathbf{K} - \mathbf{e}_1) =$$

$$= rt_1(A) + \sum_{l=2}^{L} v_{1,l}\tau_{1,l}\, n_l(B), \tag{5}$$

since $\mathbf{K} - \mathbf{e}_1$ represents load C with one less customer in chain 1, i.e., with no customers in that chain; in other words, it represents load B.

By (2), the increase in command response time can be written as:

$$\Delta rt = rt_1(C) - rt_1(A) = \sum_{l=2}^{L} v_{1,l}\, w_{1,l}(A)\, n_l(B). \tag{6}$$

Thus, under the assumptions made, the response time of a command A is a ***linear combination*** of the mean queue lengths at the non-IS centers under load B, the coefficients being the total times spent by A in the respective centers when running alone on the same machine. Note that, in the terminology adopted here, the queue length at a center also includes the customer in service at that center. Furthermore, note that IS centers do not contribute to the sum on the right-hand side of (6).

Equation (6) can be used to predict $rt_1(C)$. Its right-hand side is a load index satisfying the condition discussed in Section 3. Furthermore, rt is a linear function of the index, which, in turn, is a linear function of the mean queue lengths.

The load index we have just introduced does not have to be always used in its full generality. Indeed, it could be argued that this will in practice be the exception rather than the rule. We expect that very often one of the terms in the linear combination on the right hand side of (6) will dominate its value. If the machine has a bottleneck, the term corresponding to the saturated resource will tend to be the dominant one, and only for commands making much heavier use of other resources this will not be the case. For instance, all of the machines we have observed on our local internetwork are very substantially CPU-bound; thus, only for the (few) heavily I/O-bound commands will it not be possible to define the load as expressed by the normalized CPU queue length (note that the coefficient of this queue length in (6) for a given command is inversely proportional to the CPU's instruction execution speed, hence, in a heterogeneous network, the CPU queue length of each machine will have to be normalized by dividing it by this speed). I/O-bound commands are rare in an environment similar to ours. Thus, the mistakes possibly made by a load balancing scheme that ignores them (e.g., one based on the normalized CPU queue length as the load index), are likely to have a small impact on the scheme's performance.

In summary, the index proposed in this section may be approximated in practical cases by indices that have been adopted in a number of previous studies, such as the CPU queue length, and even implemented in existing systems, such as the load average in UNIX. The main contribution of this proposal does not consist so much of a new index, but of a theoretical foundation for all the load indices based on resource queue lengths. This foundation provides a justification for these indices, specifies the assumptions under which they are valid, and gives an indication of the limits of their applicability. As for those indices that are mentioned in the literature and cannot be obtained from (6), we have not proved their lack of validity (though some experiments based on a simple queueing model have convinced us that they do not satisfy the criterion in Section 3). We would only like to see any proponent of a load index specify the reasons for choosing it in a way that will subject such a choice to scientific scrutiny instead of requiring much guesswork or an act of faith.

5. EXPERIMENTAL VERIFICATIONS

In the preceding section, we pointed out that, according to a mean value equation valid for closed queueing networks, a linear combination of the queue lengths of a machine's resources should be a good predictor of command response time, provided that the system is in steady state and that the queueing disciplines of the resources are FCFS, PS, LCFSPR, or IS. However, these assumptions are not generally satisfied in real-world computer systems. For example, in Berkeley UNIX, a multi-class priority scheduling algorithm with round-robin preemption is used for the CPU. The scheduling discipline for the disks in Berkeley UNIX is one-way scan; thus, an incoming I/O request may find itself right ahead of the disk arm, and hence be serviced first. With the usually complicated queueing disciplines adopted for various resources, it is not clear whether a linear relation exists between resource queue lengths and command response times. The closed system and steady state assumptions in the model are also not generally satisfied in computer systems; users log on and off, and the number of jobs submitted can fluctuate widely. On the other hand, modeling experiences have repeatedly demonstrated that fairly simple models can provide results that closely reflect reality, even though some of the assumptions of the model do not strictly hold.

In order to evaluate the suitability of the resource queue lengths as load indices, and to study the sensitivity of responsiveness to variations in the queue lengths for different types of commands, we conducted a series of measurement experiments on production VAX-11/780 systems running Berkeley UNIX 4.3BSD and under both natural workloads and artificial workloads. The resource queues measured were:

(1) the *CPU ready queue*: the number of processes that are loaded and running or waiting to be scheduled to run; the process in execution is included;

(2) the *disk request queue*: the number of disk I/O requests that are waiting to be processed; this is the aggregate queue for *all* the disks in the machine's configuration.

There are other types of resources, such as memory space and network bandwidth, but our measurements show that the contention for them is very low.

While the resource queue lengths and other indices were being traced, we ran benchmark commands periodically and measured their response times. Three types of commands were used at different times:

(1) A text processing command (TROFF) that is mostly CPU-bound, with some I/O.

(2) A purely CPU-bound benchmark (CPU) that performs a large number of arithmetic computations.

(3) A predominantly I/O-bound benchmark (IO) that copies a large file block by block in reverse order to defeat the read ahead mechanism in UNIX.

For each of the measurement sessions, we computed the correlation coefficient between the average queue lengths *during* the entire command execution and the command response times. Linear regression tests were also performed to determine the amount of improvement in correlation we can obtain by using both CPU and disk queue lengths. We found that the correlation between the average CPU queue length and the response time of a CPU-bound job (TROFF or CPU) is very close to 1 under a wide spectrum of system load conditions, both live and artificial. On the other hand, the correlation between the aggregate disk queue length and the response time is much lower (the correlation coefficient was between 0.45 and 0.85), and the amount of improvement obtained by taking it into consideration is negligible. This is not only true for live workloads that usually do not cause much contention on the disks, but also with artificial workloads having high disk demands. Similarly, for the I/O-bound benchmark, when the disk is highly contended, the correlation between the disk queue length and the response time is very high (always greater than 0.85), whereas the CPU queue length has a much lower correlation (0.65 - 0.75). Considering both CPU and disk queues provides little improvement even in this case.

These results suggest that, at least for strongly CPU-bound or I/O-bound commands, resource queue lengths are excellent load indices. We have thus experimentally verified the boundary-case validity of the load index proposed in the previous section. To evaluate the ability of the resource queue lengths in predicting the response times of commands still to be started, we computed the correlation coefficient between the command response time and the average queue lengths measured during a period (5, 10, 15, 30 or 60 seconds) *before* the start of the command's execution, instead of those measured *during* the execution of the command. The coefficients were found to be much lower than in the "during" case, almost never exceeding 0.7. Attempts to improve the accuracy of the predictions by using an exponentially smoothed average of the queue lengths yielded little improvement. Furthermore, the higher the load, the greater the fluctuations observed, and the poorer the prediction. The consistently poor correlation in all the sessions suggests that the resource queue lengths are changing so rapidly that their values just before the execution of a command poorly predict those during the execution, and hence the command's response time cannot be accurately predicted. In other words, the steady state assumption in the model on which the load index proposed in Section 4 is based does not hold. This is confirmed by our measurements of the rate of change of the CPU queue length. Figure 1 shows the distribution of net CPU queue length changes during 30-second intervals over an entire session (about eight hours). As can be observed, 33% of the time the net change is 3 or beyond. For an average queue length of 6, which is fairly high, this represents a 50% change in CPU load. For the machine we measured, the average net change in CPU queue length in a 30-second interval is 2.31.

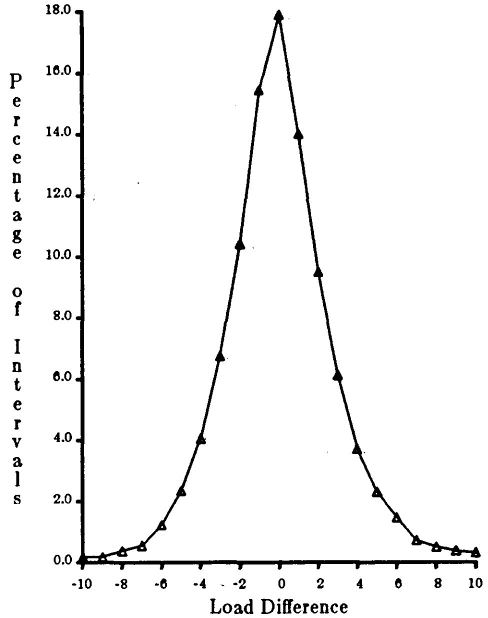


Figure 1. Load Change Frequency (in 30 Seconds) from Measurement

The prediction of command response times gets worse as the command's execution gets longer, which is a negative characteristic of our index since load balancing is most advantageous when it deals with heavy commands. However, it should be pointed out that, for load balancing, we are interested in *comparing* the loads of a number of machines in order to choose a machine that is lightly loaded, rather than determining the absolute response times the command would have if it were executed on each of the eligible machines. Another factor that is likely to ease the difficulty of prediction is that load balancing, if effective, should tend to reduce the fluctuations of the load on each machine, thereby improving the accuracy of response time prediction. In other words, load balancing should provide a *negative feedback* that will tend to stabilize machine loads. This conjecture was confirmed by the results obtained from a trace-driven simulator that we constructed for studying load balancing[14]. Records containing the CPU and disk I/O resource demands for all the jobs submitted to a production system during a number of tracing sessions were used to drive the model that simulated the executions of the jobs on a number of hosts, with and without load balancing. It was found that load balancing tends to smooth out the temporal fluctuations in the load of each machine. For a typical machine, the distributions of load changes during 30-second intervals as described above were computed for cases with and without load balancing, and are shown in Figure 2. The average net change in load decreased from 2.42 for the no-load-balancing case to 0.93 for that with load balancing. Our simulation study also demonstrates that load balancing based on resource queue lengths can reduce the average job response time by half or more when the system is loaded (average CPU utilization above 60%), even when the costs of message exchanges and job transfers are taken into consideration.

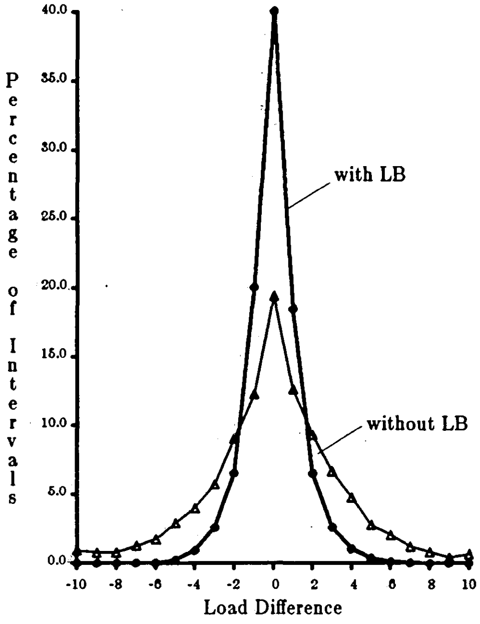


Figure 2. Load Change Frequencies (in 30 Seconds) from Simulation

6. CONCLUSION

A load index for dynamic load balancing schemes has been presented. The index is based on a mean-value equation that applies to closed multichain queueing network models with population-independent service rates and PS, LCFSPR, IS, or single-server FCFS service centers. The objective that has guided the choice of the index is the minimization of a command's execution time. The proposed index is a linear combination of the mean queue lengths in the machine being considered, the coefficients being the total times the command would spend in each service center if it ran alone on the machine.

Our measurement experiments have shown that mean queue lengths can be good load indices indeed, if used properly. For CPU-bound commands, the CPU queue length accurately predicts the response time, whereas the disk queue lengths have negligible effect. For I/O-bound commands, the particular disk queue length reflects the response time very well when processes are queued up there. However, for live workloads characterized by short queues at the disks, the response time seems to be relatively insensitive to both CPU and disk queue lengths in this case.

The load of the live systems we measured in our environment changes very rapidly, making response time predictions difficult. On the other hand, load balancing is expected to smooth out the wide fluctuations, thereby improving the accuracy of the predictions. Also, predicting absolute response times accurately is not necessary, as long as the relative rankings of the machine loads are reflected in the respective values of the index.

Instantaneous queue length measurements are not difficult to perform. They could be gathered periodically (or on demand) and broadcast (or sent to the requesting machine). Whether instantaneous or smoothed queue lengths should be used is not clear, and can only be decided after careful experimentation; however, we note that smoothing can be easily performed when instantaneous values are available. In the fully general case, the contribution of each machine to the value of the load index would thus be a vector[13]. That of each command would be another vector, with components equal to the times spent by the command in each server when running alone. Clearly, each command must be represented by different vectors of coefficients for different machines or even different configurations. This characteristic should not, however, be regarded only as a drawback: in fact, the ability elegantly to adapt to configurationally and architecturally heterogeneous networks is a major advantage of the load index proposed in this paper.

The dependence of the value of the index on the particular command being considered is a very simple one, and the coefficients that characterize each command are easy to measure. However, command dependence causes two problems:

(i) the *absolute* load of a machine cannot be defined; this problem can be alleviated, as noted in Section 3, by referring to a standard workload (a "basket" of commands); in any case, the index only measures the load *relative* to a given command or mix of commands;

(ii) the coefficients characterizing a command generally depend on the command's arguments (input files, input data, options, and so on); thus, an accurate characterization of a command type may turn out to be very complex and very expensive to build; also, any modifications to the command's code may cause appreciable changes in the values of the coefficients.

As stated at the end of Section 4, the index will very seldom, if ever, have to be applied in its full generality. Problem (ii), which does not arise when only the CPU term of the linear combination is taken into account, is a very hard one to solve. Our hope is that, for most important commands, the dependence of the coefficients on the arguments can be approximated by functions with simple mathematical forms, and that their sensitivity to changes is low. For instance, we conjecture that the coefficients (or at least some of them) of text processing and compilation commands are roughly linear functions of the size of the input file. These and other similar conjectures will, however, have to be validated by an extensive study which is being planned now.

ACKNOWLEDGEMENTS

The authors are grateful to Harry Rubin for making his benchmark program available to them, and for his comments on the manuscript. They are also indebted to Hamid Bahadori, who provided useful comments, and to Jean Richter for her impeccable preparation of the various human-readable versions of this paper.

REFERENCES

[1] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems*. Prentice-Hall, 1983.

[2] A. Barak and Z. Drezner, "Distributed algorithms for the average load of a multicomputer," Tech. Rept. CRL-TR-17-84, Computing Research Laboratory, University of Michigan, March 1984.

[3] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated dynamic load sharing," *Proc. 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (August 1985), pp. 1-3.

[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Dynamic load sharing in homogeneous distributed systems," Technical Report 84-10-01, Department of Computer Science, University of Washington, Seattle, October 1984.

[5] Y.-T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Transactions on Computers C-34*, vol. 3 (March 1985), pp. 204-217.

[6] Y.-C. Chow and W. H. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Transactions on Computers C-28*, vol. 5 (May 1979), pp. 356-361.

[7] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," *Proc. ACM Computer Network Performance Symposium* (April 1982), pp. 47-55.

[8] J. A. Stankovic, "Decentralized control of job scheduling," *IEEE Transactions on Computers C-34*, vol. 2 (Feb. 1985), pp. 117-130.

[9] R. Bryant and R. A. Finkel, "A stable distributed scheduling algorithm," *Proc. 2nd International Conference on Distributed Computing Systems* (April 1981), pp. 314-323.

[10] A. Barak and A. Shiloh, "A distributed load balancing policy for a multicomputer," Internal Report, Department of Computer Science, The Hebrew University of Jerusalem, 1984.

[11] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *Journal of the ACM 22*, vol. 3 (April 1975), pp. 248-260.

[12] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queueing networks," *Journal of the ACM 27*, vol. 2 (April 1980), pp. 313-322.

[13] S. Zatti, "A multivariable information scheme to balance the load in a distributed system," Rept. No. UCB/CSD 85/234 (PROGRES Rept. No. 85.6), University of California, Berkeley, May 1985.

[14] S. Zhou, "A trace-driven simulation study of dynamic load balancing," Rept. No. UCB/CSD 86/, University of California, Berkeley, June 1986.

AN APPROXIMATION OF THE PROCESSING TIME FOR A RANDOM GRAPH MODEL OF PARALLEL COMPUTATION

E. Gelenbe (*), R. Nelson, T. Philips, and A. Tantawi

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract

The task graph of a parallel computation is modeled by a random, acyclic, directed graph (K, p) where K is the number of nodes and p is the probability that an arc exists between two nodes (from a smaller numbered node to a larger one). In this model, each node corresponds to a task that is to be processed and each arc implies a precedence relationship that must be satisfied during processing. An approximation for the expected processing time of a task graph of this type on an infinite number of processors is derived under the assumption that each task requires an exponentially distributed amount of processing. This approximation shows that the expected processing time grows linearly with K at rate $2p/(1+p)$. Furthermore, maximum likelihood estimators for the value of p given specified characteristics of a particular task graph are obtained.

1. Introduction and Model Description

In multiprocessing computer systems, a job is divided into a number of tasks, each executing on a processing unit concurrently with other tasks. The division of a job into a set of tasks is governed by the logical structure of the job and the interdependencies among tasks. This is usually described by a computation graph where nodes in the graph represent tasks and directed edges represent precedence relations. Such a graph may be used to represent the detailed processing stages in a single program; it may also be used to represent the dependencies between separate programs in a large software system. Another interpretation of this graph model is in terms of a PERT diagram (or activity network), commonly used to organize or schedule tasks in industrial engineering and operations research.

In general, one associates processing or execution times with the nodes of a computation graph in order to represent the time it would take to carry it out assuming a given processing system. Furthermore, it is assumed that each task represented by a node cannot itself be subdivided any further into smaller tasks. An important performance measure of parallel processing systems is the processing time of a job described by a computation graph.

The computation graph, also referred to as a task graph, is characterized by the number of tasks, their processing times, and the structure of precedence relations. Special cases of structures of precedence relations represented by series parallel graph models have been considered[5,6]. Fayolle et al.[1] consider a tree structure of precedence relations where the number of sons of a node in the tree is described by a random variable. They analyze such a structure and obtain the distribution of the job processing time of a job on an infinite number of processors, in the special case that the number of sons is geometrically distributed and the task processing times are constant. Otherwise, numerical solutions are calculated.

Again, Mussi and Nain[4] assume a tree structure with exponentially distributed task processing times and consider two execution policies: (1) level-by-level and (2) take-a-leaf. They obtain the Laplace-Stieltjes transform of the processing time of the tree under the two policies. For policy (1), they assume an arbitrary number

(*) This work was performed while E. Gelenbe was visiting the IBM T. J. Watson research center. His present address is: ISEM, Bat 508, University Paris Sud, 91405 Orsay, FRANCE.

CH2345-7/86/0000/0691$01.00 © 1986 IEEE

of processors, whereas for policy (2) they assume equal expected task processing times and consider two cases: two processors and an infinite number of processors.

In many cases the structure of precedence relations is either unknown or the number of tasks is so large that a characterization of the structure of precedence relations is hard to obtain. In such cases, since a detailed analysis of the task graph is intractable, one attempts to characterize the interdependencies between nodes using a higher level model. Solutions of such models can provide insights into performance characteristics inherent in the underlying system. One such model is to assume that precedence relationships between nodes are determined in a probabilistic manner. Specifically, in such models one assumes that for every ordered pair of nodes there is a specified probability that the first node is a predecessor of the second. By varying the value of these probabilities one can model both very sparse and very dense task graphs. Random graph models of this type can be found in the literature for various applications. Indurkhya et al.[3], for example, consider a random graph model of a distributed program and derive properties of the optimal task assignment policies for heterogeneous processors. Random graphs are also used to model concurrency in transaction processing systems. A node in such a graph represents a transaction and an edge represents a conflict. An edge between pairs of transactions exists with some fixed probability. Franaszek and Robinson[2] consider such a random graph model and obtain bounds on the expected number of transactions that can run concurrently.

In this paper, we consider parallel computations which are modeled by random, acyclic, directed graphs consisting of K nodes in which an arc from node i, $i = 1, 2, ..., K - 1$ to node j, $j = i + 1, i + 2, ..., K$ exists independently with a fixed probability p. In this model, each node corresponds to a task that is to be processed and each arc implies a precedence relationship that must be satisfied during processing. We assume that each task requires an exponentially distributed amount of processing with mean 1. The processing time of a task graph of this type on an infinite number of processors is denoted by $T_K(p)$, and its expectation by $E[T_K(p)]$.

An approximate expression for the expected processing time of a job described by such a graph is derived in Section 2. In Section 3, this approximate expression is compared to simulation results. In Section 4, maximum likelihood estimators for the value of the probability p are derived.

2. Approximate Analysis of the Expected Processing Time

We next show that the expected processing time varies almost linearly with K, the slope being given by $2p/(1 + p)$. To do this, we will consider a particular realization of a $(K + 1, p)$ graph. We can imagine that this graph is generated by taking an already generated (K, p) graph, adding a $(K + 1)$st node and then adding directed edges from nodes $1, 2, ..., K$ to node $K + 1$ independently with probability p. We note that because no directed edge exists from node $K + 1$ to any lower numbered node, the completion times of the nodes in the original (K, p) graph are not influenced by the addition of node $K + 1$. Furthermore, we note that the distribution of the processing time of the $(K + 1)$st node is not influenced by the existence of edges leading into it.

Consider the execution of the original (K, p) graph, with $K > 1$ (the case of $K = 1$ is trivial). We relabel the nodes in the graph in the following manner: Let 0 be the last node to finish execution and let 1 be the direct predecessor of 0 that, among all the predecessors of 0 finished executing last. By this definition, the completion time of node 1 is the starting time of node 0. In a similar manner, let node i, $i = 1, 2, ..., L - 1$, be the direct predecessor of node $i - 1$ that was the last to finish executing. It is clear that all the nodes of the original (K, p) graph are not necessarily relabeled by this procedure but those that are labeled lie on a path. Furthermore, by definition, this path contains the node that completes last. Accordingly we will call, for this particular execution, the path defined by nodes $L - 1, L - 2, ..., 0$ the *longest path* of the original (K, p) graph. Nodes not labeled in this procedure will play no role in our approximation. We will now use these relabeled nodes to derive an approximation to the expected processing time for the $(K + 1, p)$ graph defined above. Basically our method of approximating the expected processing time of the $(K + 1, p)$ graph is to consider the relationship between the longest path of the original (K, p) graph and the longest path of the new $(K + 1, p)$ graph.

The processing time of the $(K + 1, p)$ graph depends on the processing time of the longest path for the underlying (K, p) graph and upon the arcs directed into the

$(K + 1)$st node. Define e_i, $i = 0, 1, \ldots, L - 1$ to be the event that in the reordered graph, an arc exists from node i to the $(K + 1)$st node, but that no arc exists between node j and it for $j < i$. Furthermore, let e_L be the event that no arcs exist from node i, $i = 0, 1, \ldots, L - 1$, to the $(K + 1)$st node. Denote the probability of event e_i by $P[e_i]$. It is clear that

$$P[e_i] = \begin{cases} pq^i, & i = 0, 1, ..., L - 1, \\ q^L, & i = L. \end{cases} \qquad (1)$$

By conditioning on these events, we write the expected increase in processing time as

$$\mathrm{E}[T_{K+1}(p) - T_K(p)] = \sum_{i=0}^{L} \mathrm{E}[T_{K+1}(p) - T_K(p) \,|\, e_i] \times P[e_i]. \qquad (2)$$

We will develop an approximation to this expression by assuming that nodes not lying on the longest path do not influence the increase in processing time that arises when the $(K + 1)$st node is added to the graph. One can easily create examples where this assumption is false. Our justification for this assumption and for the assumption that we will next make concerning the processing time of nodes along the longest path, essentially lies in the mathematical difficulty of obtaining an exact solution to equation (2) and the intuitively plausible argument that the processing time of a random graph of this nature should depend on the processing time of the longest path. Using simulation we will show that these assumptions lead to a good approximate formulae for the expected processing time of these graphs.

To continue with the derivation of our approximation we next must determine an approximation to the conditional expected increase in processing time given in equation (2). If $i = 0$, then the expected processing time is clearly incremented by 1. For $i > 0$, we will assume that the intercompletion intervals are exponentially distributed with parameter 1. Since node i, $i = 1, 2, ..., L - 1$, is defined to be the direct predecessor of node $i - 1$ that finished execution *last*, its expected processing time is lower bounded by the expectation of an exponential random variable with parameter 1. This assumption will thus lead to an approximate expected processing time of the $(K + 1, p)$ graph that is a lower bound to the actual expected processing time. If we condition on the event e_i it is apparent that there are two possible cases for the execution of the resultant $(K + 1, p)$ graph. The first is that node $K + 1$ finishes executing before nodes lying on the path i, $i - 1, ..., 0$ and the second is that it finishes after these nodes along the longest path. Since the processing times of nodes i, $i - 1, ..., 0$ are assumed to be independent and exponentially distributed with parameter 1, the sum of their processing times is Erlangian with parameter i. Then the expected increase in the processing time that results from the addition of node $K + 1$ can be calculated by determining the expectation of the positive difference between an exponentially distributed random variable and one whose distribution is Erlangian with parameter i. In Appendix A it is shown that this quantity equals $(1/2)^i$.

By substituting $\mathrm{E}[T_{K+1}(p) - T_K(p) \,|\, e_i] \cong (1/2)^i$ into equation (2) we get

$$\mathrm{E}[T_{K+1}(p) - T_K(p)] \cong \sum_{i=0}^{L} (1/2)^i P[e_i]. \qquad (3)$$

As K grows the number of nodes on the longest path also increases. For large K we will approximate the finite sum in equation (3) with an infinte sum and thus write:

$$\mathrm{E}[T_{K+1}(p) - T_K(p)] \cong \sum_{i=0}^{\infty} p(\frac{q}{2})^i = \frac{2p}{1 + p}.$$

The expected processing time can now be approximated by

$$E[T_K(p)] \cong E[T_N(p)] + (K - N) \times 2p/(1 + p), \quad (4)$$

$K \geq N \geq 1$, where $E[T_N(p)]$ is an initial value that is determined by some other method. One possible method to determine this initial value is to use the boundary condition $E[T_1(p)] = 1$. Another method is to run a simulation for some suitably large N.

3. Comparison of Approximation and Simulation

In this section we compare our approximation to results obtained from simulation. We ran simulations for values of p ranging from .01 to .9 and values of K in the range of 25 to 500. Accurate simulation for greater K values was found to be computationally intractable. For each (K, p) graph our simulation worked as follows: We first generated a random (K, p) graph. This is performed by creating a $K \times K$ adjacency matrix where each element is initially set to 0. We added random arcs to this graph by independently replacing each zero in the lower triangular portion of of the matrix with a 1 with probability p. The resultant (K, p) graph was then simulated to determine its expected processing time. To do this we generated confidence intervals using independent replications. More precisely, after 30 replications we sampled the processing times obtained and continued the replication process until the width of the 95% confidence interval was less than 5% of the sampled average or until a maximum of 200 replications had been performed. We then continued by generating another instantiation of a (K, p) graph and determined its expected processing time in the same manner. After generating processing times for 30 different (K, p) graphs we checked the confidence intervals obtained over this set of graphs and continued to generate graphs until the 95% confidence interval was less than 5% of the average or until a maximum of 200 graphs had been generated.

The results of this simulation are shown in Figure 1 where we also plot our approximation using equation (4). The accuracy of our simulation is shown by the fact that the 95% confidence interval widths are approximately the size of the points used to indicate the values obtained from simulation. The value of the offset values in equation (4), denoted by $E[T_N(p)]$, is given by the simulated values obtained for $N = 25$ and the corresponding value of p. As shown in the Figure, our approximation compares favorably with the simulation results and, as noted before, is an underestimation for the expected processing time. It is important to note that for a fixed value of p, our approximation holds for sufficiently large K. If $p \geq .1$, the approximation is accurate for graphs with as few as twenty or thirty nodes. On the other hand, if p is close to 0, K may have to be very large indeed (on the order of a thousand nodes) for the approximation to be useful. The reason for this behavior is as follows. If both p and K are small, then the graph consists primarily of roots. As all the roots can be processed simultaneously, one would expect the completion time to grow logarithmically with K. As K grows, however, a rich interdependency among the various nodes appears, and it is this interdependency that finally gives rise to the observed linear growth in the processing time with K. These considerations explain the observation, a shown in the figure, that for a given value of K the approximation is more accurate for large values of p.

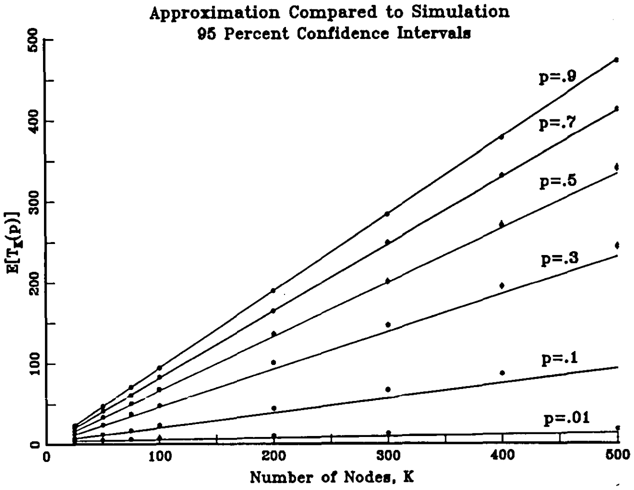


Figure 1

4. Maximum Likelihood Estimation of the Probability of Conflict

Often, when modeling an actual computation by a random graph, no apriori knowledge of the probability of conflict is available, and consequently it is necessary to have an estimation procedure so that the value of p can be determined experimentally. In this section, three estimators for p are presented. The first estimates p from the indegree of a randomly chosen node, the second estimates p from the number of 'roots' or nodes of indegree 0 in the graph, and the third estimates p from the total number of arcs in the graph.

4.1. Estimating p from the Indegree of a Randomly Chosen Node

Given a (K, p) graph, we denote the probability that the indegree of a randomly chosen node equals D by $L_K(D, p)$. This likelihood function is given by

$$L_K(D,\ p) = \frac{1}{(K-D)} \sum_{i=0}^{K-D-1} \binom{D+i}{D} p^D (1-p)^i,$$

where the ith term in the summation is the probability that vertex $D + i$ has indegree D. Expanding the binomial coefficient out we get

$$L_K(D, p)$$
$$= \frac{(-p)^D}{(K-D)\, D!} \times \frac{d^D}{dp^D}\Big[\sum_{i=0}^{K-D-1} (1-p)^{D+i} \Big]$$

$$= \frac{(-p)^D}{(K-D)\,D!} \times \frac{d^D}{dp^D}\left[\frac{(1-p)^D - (1-p)^K}{p}\right]$$

$$= \frac{(-p)^D}{(K-D)\,D!}$$

$$\times \frac{d^D}{dp^D} \sum_{i=1}^{K}\left[\binom{K}{i} - \binom{D}{i}\right](-1)^{i-1} p^{i-1}.$$

Only powers of p greater than D survive the differentiation, and some rearrangement gives us

$L_K(D,p)$

$$= \frac{-1}{K-D}\binom{K}{D} p^{D-1} \sum_{i=0}^{K-D} \binom{K-D}{i} \frac{i}{i+D}(-p)^j$$

$$= \frac{-1}{K-D}\binom{K}{D} p^{D-1}$$

$$\times \int_0^1 \frac{d}{dx}\sum_{i=0}^{K-D}\binom{K-D}{i}(-x)^i y^{D+i-1}\Big|_{x=p}\, dy$$

$$= \binom{K}{D} p^D \int_0^1 y^D (1-py)^{K-D-1}\, dy$$

$$\cong \binom{K}{D} p^D \int_0^1 y^D e^{-pyK}\, dy$$

$$= \binom{K}{D} p^D \frac{D!}{(pK)^{D+1}}\left[1 - e^{-pK}\sum_{i=0}^{D}\frac{(Kp)^i}{i!}\right].$$

Let $\hat{p}(D)$ be a maximum likelihood estimate of p, i.e.

$$L_K(D,\hat{p}) \geq L_K(D,p'), \qquad \forall p' \in [0,1].$$

To find $\hat{p}$, the expression for $L_K(D,P)$ must be maximized, and this is most easily done by differentiating it with respect to p, equating it to 0, and then solving it for $\hat{p}$. On doing so, we get after some simplification

$$\frac{(K\hat{p})^{D+1}}{D!} = \sum_{i=D+1}^{\infty}\frac{(K\hat{p})^i}{i!}$$

or

$$\sum_{i=D+2}^{\infty}\frac{(K\hat{p})^{i-D-1}\times(D+1)!}{i!} = D. \qquad (5)$$

If we replace the product $K\hat{p}$ by the single variable x the left hand side can be seen to be a polynomial of infinite degree. As it is strictly convex, and takes on a value of when $x = 0$, we may conclude that a unique solution exists. This in turn implies that $\hat{p} = \alpha(D)/K$ where $\alpha(D)$ is the solution to the above equation. To insure numerical stability of the solution to equation (5), it is convenient to define the following set of functions. We let $F_0(x) = x/(D+2)$, and let $F_{j+1}(x) = F_j(x)(1 + x/(D + j + 3))$. It is clear that $F_j(x)$ is the value of the summation of equation (5) truncated at $j + D + 2$. Using these relationships, the equation $F_j(x) - D = 0$ can be approximated numerically to any degree of desired precision by choosing j suitably large. Clearly, p can be estimated from the outdegree of a randomly chosen node in exactly the same manner.

4.2. Estimating p from the Number of Roots

Let R be the number of roots, or nodes of indegree 0 in a particular realization of a (K,p) graph. As all of these nodes can be processed simultaneously the number of roots in a (K,p) graph is the maximum level of concurrency that can be achieved when the program or task begins execution. Define $P_K(R,p)$ to be the probability that a (K,p) graph possesses exactly R roots. As before, define $\hat{p}(R)$ to be a maximum likelihood estimate of p, i.e.

$$P_K(R,\hat{p}) \geq P_K(R,p'), \qquad \forall p' \in [0,1].$$

The following recurrence can be written for the distribution of the number of roots:

$P_K(j,p) =$

$$\sum_{i=\max(1,j-1)}^{K-1} \binom{i}{i-j+1} p^{i-j+1}(1-p)^{j-1} P_{K-1}(i,p).(6)$$

The ith term in the sum is the probability that the addition of a single node to a $(K-1,p)$ graph with i roots results in a (K,p) graph with j roots. Unfortunately, this recurrence cannot be solved to give $P_K(j,p)$ in closed form, though it can be computed from (6) starting with $P_1(1,p) = 1$ and $P_1(0,p) = 0$. On the other hand, the mean of the distribution is easily found as follows. Define a set of K indicator random variables $\{x_i\}$ as

$$x_i = \begin{cases} 1 & \text{if node i is a root,} \\ 0 & \text{if node i is not a root.} \end{cases}$$

Clearly, $Pr\{x_i = 1\} = q^{i-1}$. Let X be a random variable that counts the number of roots. Then $X = \sum_{i=1}^{K} x_i$, and

$$\overline{X} = \sum_{i=1}^{K} q^{i-1} = \frac{1 - q^{K+1}}{p}. \tag{7}$$

This leads us to consider the following estimator for p, which is *not* a maximum likelihood estimator. Our estimate of p, $\tilde{p}$, is the solution to the equation

$$\tilde{p} = \frac{1 - \tilde{q}^{K+1}}{R}, \tag{8}$$

where R, as stated before is the observed number of roots. As the distribution of $\tilde{p}$ is not known, the mean and the variance of the estimator cannot be found. Computational experience with (8) however shows $\tilde{p}$ to lie within 5% of the maximum likelihood estimate of p.

4.3. *Estimating p from the Number of Edges in the Graph*

Let E be the number of arcs in a particular sample realization of a (K,p) graph, and $L_K(E,p)$ the likelihood function for p. Then, we have

$$L_K(E,p) = \begin{pmatrix} \binom{K}{2} \\ E \end{pmatrix} \times p^{E} \times q^{\binom{K}{2} - E}.$$

By differentiating this expression and equating it to 0, we find that the maximum likelihood estimate of p, $\hat{p}$ is the solution to

$$E(1 - \hat{p}) = \hat{p}(\binom{k}{2} - E)$$

or that

$$\hat{p} = \frac{E}{\binom{K}{2}}$$

exactly as one would suspect.

5. Conclusions and Future Research

In this paper we have considered the execution of a job modeled by a task graph with precedence relationships on an infinite number of processors. Our model of the task graph was a random, acyclic, directed graph consisting of K nodes in which an arc existed from node i, $i = 1, 2, ..., K - 1$ to node $j, j = i + 1, i + 2, ..., K$ independently with probability p. In this model, each vertex corresponds to a task that is to be processed and each arc implies a precedence relationship that must be satisfied during processing. The processing time of the tasks in our model was assumed to consist of independent and identically distributed exponential random variables. The approximation we derived for the expected processing time of the task graph, which we call a (K,p) graph, exhibits linear growth with the number of tasks for a given value of p and compares favorably to simulation values.

In our model we assume an infinite number of processors, or more precisely that whenever a task is available for execution there exists an idle processor upon which it can be executed. For other than very small values of p this is not a restrictive assumption. The reason for this follows from equation (7) which shows that, for large K, the expected number of tasks that can be simultaneously executed in a (K,p) graph is approximately equal to $1/p$. This approximation becomes more accurate with increasing K. Thus the expected number of processors necessary to *begin* execution of a (K,p) graph is not large for all but small values of p. The same reasoning cannot be used to determine the expected number of processors needed *during* the execution of a (K,p) graph. This follows from the fact that as tasks are removed from the graph (i.e. they finish executing) the remaining graph is no longer an example of a (K,p) graph. In other words, conditioned on the fact that a root is removed from the graph, the underlying statistics are not the same as those for the initially generated graph. Simulation results (not shown here) however, show that as the graph is executed the expected number of processors required is non-increasing. Thus we can conclude that $1/p$ is an approximation to the expected number of processors required at any point in the execution.

Appendix A

Lemma 1.

Let X be an exponentially distributed random variable with parameter 1, and let Y be the sum of $i \geq 1$ independent exponentials with parameter 1. Assume that X and Y are independent and let $Z = \max(0, X - Y)$. Then $E[Z] = (1/2)^i$.

Proof.

The density functions for X and Y are defined on the non-negative real line and are given by

$$f_X(x) = e^{-x}, \qquad x \geq 0,$$
$$f_Y(y) = \frac{y^{i-1} e^{-y}}{(i-1)!}, \quad y \geq 0,$$

respectively. The density function for Z consists of an impulse at $z = 0$ and for $z > 0$ can be written as

$$\begin{aligned} f_Z(z) &= \int_{0^+}^{\infty} f_X(x+z) f_Y(x) dx \\ &= \frac{e^{-z}}{(i-1)!} \int_{0^+}^{\infty} x^{i-1} e^{-2x} dx \\ &= \frac{e^{-z}}{2^i}, \quad z > 0. \end{aligned}$$

Using this expression we thus have that the expectation of Z is given by

$$E[Z] = \int_{0^+}^{\infty} \frac{ze^{-z} dz}{2^i} = (1/2)^i. \qquad \blacksquare$$

References

[1] G. Fayolle, P. J. B. King, and I. Mitrani, "On the Execution of Programs by Many Processors", in Performance'83, A. K. Agrawala and S. K. Tripathi, eds., North Holland, pp. 217-228, 1983.

[2] P. Franaszek and J. T. Robinson, "Limitations of Concurrency in Transaction processing", ACM Transactions on Database Systems 10, 1, pp. 1-28, March 1985.

[3] B. Indurkhya, H. S. Stone, and L. Xi-Cheng, "Optimal Partitioning of Randomly Generated Distributed Programs", IEEE Transactions on Software Engineering 12, 3, pp. 483-495, March 1986.

[4] P. Mussi and P. Nain, "Evaluation of Parallel Execution of Program Tree Structures", Proc. 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Cambridge, Mass., pp. 78-87, August 1984.

[5] J. T. Robinson, "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms", IEEE Transactions on Software Engineering 5, 1, pp. 24-31, January 1979.

[6] R. A. Sahner and K. S. Trivedi, "SPADE: A tool for Performance and Reliability Evaluation", Research report CS-1984-15, Duke University, 1984.

PERFORMANCE ANALYSIS OF DYNAMIC LOCKING

In Kyung Ryu
Dept. of Electrical Eng-Systems, SAL 300
University of Southern California
Los Angeles, CA 90089

Alexander Thomasian
IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

The paper is concerned with performance analysis of a transaction processing system due to both hardware and data resource contention. Hardware resource contention is taken into account by analyzing the queueing network model of the computer system on which the transactions are executed. Transactions lock the data items they access from the database dynamically, according to strict two-phase locking. Data contention manifests itself as lock conflicts and deadlocks. Lock conflicts (resp. deadlocks) are resolved by blocking (resp. restarting) the transaction that caused the lock conflict. Probabilistic analysis is used to determine the frequency of lock conflicts/deadlocks. A multilevel solution method based on decomposition is then used to obtain system and user-level performance measures. An iterative solution scheme with reduced computation cost is also derived from the decomposition solution method. Graphs/Tables are provided to show validation results against simulation and to study the behavior of the system. The analysis presented in this paper can be generalized in several directions.

1. Introduction

In this paper we develop a detailed Queueing Network Model (QNM) of transaction processing in a centralized database. Concurrency control of dynamic lock requests is attained by a strict two-phase locking (2PL) scheme [4], i.e., locks are released upon transaction completion. Transactions encountering lock conflicts are blocked and the transaction causing a deadlock is restarted.

An analytic solution based on multilevel decomposition is adopted. A top-down method is used to identify the parameters required to compute the performance measures of interest at each level with the ultimate goal of user-level performance measures. Such parameters are then computed at the lower level, perhaps requiring the computation/measurement/estimation of additional lower level parameters. Probabilistic analysis is used to estimate the probability of lock conflict based on an estimate of locks held by in-progress transactions. We also present an iterative solution based on the analysis developed for the decomposition method.

The performance of dynamic locking schemes has been studied by numerous researchers. Extensive simulation studies of dynamic locking were presented in [9, 1]. A system with two-phase locking was simulated in [8]. Regression analysis was then applied to simulation results to obtain equations relating lock conflict probabilities to key system parameters such as number of transactions, transaction size, and database size.

The probability of lock conflict and deadlock was computed in [5, 14]. In [5], each transaction was assumed to hold half of the locks it will request in its lifetime. Simple expressions for probability of conflict and deadlock were obtained based on this simple assumption. In [14], the state of the system was represented by a forest of trees, where the roots of the trees denote active transactions and the non-root nodes of each tree denote transactions blocked by their parent node.

A two-phase locking scheme with dynamic lock request policy was presented in [6, 18], using a lock-wait station to represent the delay due to blocking. In [6], the lock-wait station was modeled by an infinite-server station with a fixed service time corresponding to the no-queueing transaction response time. A more accurate solution was presented in [18], by using a service time at the lock-wait stations related to residual transaction residence times. Since transaction residence times are unknown a priori, iteration was used to solve the model. The results of this analysis could be further improved by adopting a state-dependent server for the lock-wait queue.

A mean value analysis method to study locking in database appeared in [15, 16, 17]. The analysis was based on estimating the mean number of transactions holding a given number of locks in the active and blocked states.

Individual transactions were treated in isolation and analyzed in steady state in [2]. The influence of other transactions in the system was represented by the probabilities that a data item required by a transaction is unavailable on the first and second attempts. An iterative method was used for the solution and a proof of convergence was given.

The work reported in this study was undertaken to alleviate the shortcomings of previous work from the viewpoint of obtaining a very accurate model for dynamic locking. A solution method is based on decomposition for a comprehensive model of centralized databases. We were able to determine detailed performance measures,

CH2345-7/86/0000/0698$01.00 © 1986 IEEE

previously only reported in simulation studies. A comparison of some of our conclusions with those of others appears in Section 4.

In what follows, we first describe the model of the computer system. A solution method based on multilevel decomposition is discussed in Section 3. In Section 4 we validate the proposed analytic methods and carry out a limited parametric study of the system behavior. A detailed analysis of Data Contention Model appears in the Appendix.

2. Description of the Model

We are concerned with concurrent transaction processing on a multiprogrammed computer system. The behavior of transactions from the viewpoint of accessing the database is described in Section 2.1. We then describe the processing of transactions by the computer system in Section 2.2.

2.1. Database Model

The database consists of **D** data items. There are **N** locks in total, each lock is associated with **G** data items (G=D/N). A single level of locking is considered in our studies. Lock requests are exclusive. Each transaction accesses a fixed number of data items (**n**), which are uniformly distributed over the D data items in the database. The case of variable number of lock requests is considered in [10]. The probability $P_m(k|n)$ that k locks are requested by a transaction accessing n data items can be computed according to the following recursion for k>0 and $n \geq 1$:

$$P_m(k|n)=\frac{kG-n+1}{D-n+1}P_m(k|n-1)+\frac{D-(k-1)G}{D-n+1}P_m(k-1|n-1) \qquad (1)$$

with the initialization $P_m(k|0)=1$ for k=0 and $P_m(k|0)=0$ for k>0. Thus the mean number of locks requested by a transaction ($\bar{L}$) is given by:

$$\bar{L} = \sum_{k=1}^{L} P_m(k|n)\ k \qquad (2)$$

when n is fixed and L is the maximum number of lock requests per transaction: L=min(n,N).

We ignore the variability of the number of lock requests and assume that each transaction requests $\bar{L}$ locks and that the $\bar{L}$ lock requests are distributed uniformly over the lifetime of a transaction (it attains the same amounts of service demands at the devices of the computer system in each interval preceding a lock request and leading to transaction completion). The fact that data item requests are uniformly distributed over the D data items in the database implies that the $\bar{L}$ lock requests are also uniformly distributed over the N locks (all lock requests are unique). A transaction requesting $\bar{L}$ locks has $\bar{L}$+1 phases, where the last phase results in the release of all locks rather than a new lock request. Such a model has been used in [5, 6, 16, 18]. Note that this assumption is compatible with an assumption that n database accesses are uniformly distributed over the lifetime of a transaction in the case of the finest granularity of locking (one lock per data item), since each access to the database results in a lock request. This is not so for coarse granularity of locking, where it is possible that the transaction already holds the lock on a data item when it makes a lock request. Thus the lock request rate by a transaction decreases as it acquires more locks, i.e., lock requests are concentrated in the beginning of transaction execution. Thus the uniform lock request distribution assumption will result in underestimating lock utilization, since locks are in fact held for more than half of transaction's residence time in the system. The results of our analysis tend to be less accurate for coarse granularity of locking (see Figure 4.1 in Section 4.3). This is of little concern, since a coarse granularity of locking is inappropriate for dynamic locking.

2.2. The Computer System Model

Transaction arrivals at the computer system are assumed to be Markovian, according to a random (Poisson) process with fixed rate λ, or terminal-driven arrivals, in which case the arrival rate is a function of the number of transactions in the system.

The maximum number of transactions that can be activated is limited to W, which acts as a congestion control parameter.[1] Activation means that the transaction is admitted from the Pending Queue into the Memory Queue, as shown in Figure 2-1. The number of Activated transactions is V=min(A,W), where **A** is the total number of transactions available for processing. Activated transactions can be processed by the Computer System, which has a maximum Multi-Programming Level (MPL) equal to **M**. The value of W should be chosen appropriately in order not to degrade system throughput by reducing the effective MPL. On the other hand activating a large number of transactions can result in excessive lock conflicts and deadlocks. In fact, thrashing, similarly to multiprogrammed virtual memory computer systems, may occur if congestion control is not exercised (see Section 4.5).

The flow of transactions is shown in Figure 2-1. Transactions that are admitted into the Memory Queue and proceed to execute at the Computer System are referred to as Eligible transactions. The number of Eligible transactions is denoted by J and there are min(J,M) Active transactions and max(0,J-M) transactions delayed in the Memory Queue. A transaction making a lock request either succeeds, in which case it continues with its processing, or it fails: lock conflict. A transaction causing a deadlock releases all of its locks (unblocking at least one transaction from the Lock-Wait Queue) and is immediately restarted by becoming an Eligible transaction again. Otherwise, the transaction is blocked in the Lock-Wait Queue until the requested lock becomes available.

[1] We use capital letters to refer to terms with a special meaning in our model.

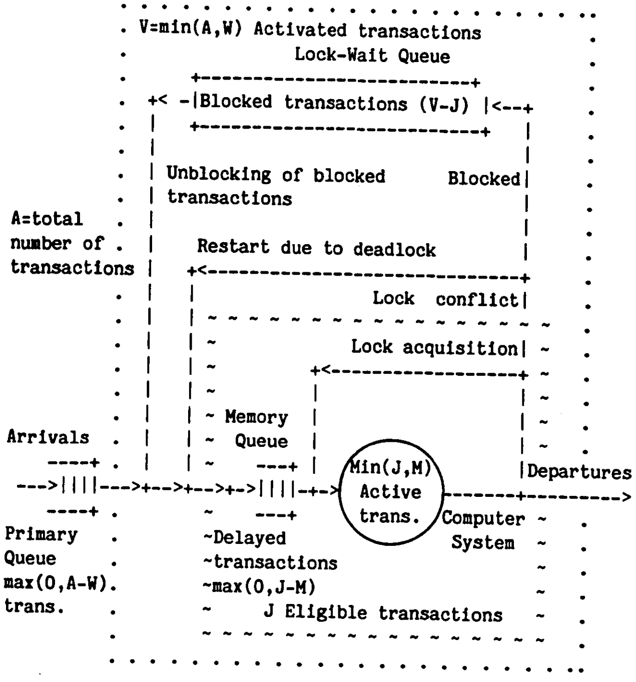

V=min(A,W) Activated transactions
Lock-Wait Queue
Blocked transactions (V-J)
Unblocking of blocked transactions
Blocked
A=total number of transactions
Restart due to deadlock
Lock conflict
Lock acquisition
Memory Queue
Arrivals
Min(J,M) Active trans.
Departures
Computer System
Primary Queue max(0,A-W) trans.
Delayed transactions max(0,J-M)
J Eligible transactions

Figure 2-1: The computer system model.

Each transaction is considered to consist of elementary transactions, each corresponding to a data item access or a lock request. Only the latter model is considered in this paper, while the former model is considered in [11, 10]. Based on an earlier assumption that lock requests are uniformly distributed over the lifetime of a transaction, it follows that all elementary transactions are homogeneous, i.e., belong to a single job-type. A transaction consists of $\bar{L}+1$ elementary transactions on the average, with the first $\bar{L}$ phases leading to a lock request and the last phase resulting in the release of all locks. Upon the completion of a transaction a check is made to unblock transactions in the Lock-Wait Queue, which were waiting on the locks held by the completed transaction. Newly unblocked transactions and delayed transactions in the Memory Queue are allocated at the Computer System in FCFS order (observing the MPL constraint).

The Computer System consists of **K** state-independent devices: a CPU and K-1 disks. It has a maximum MPL equal to **M**. Transactions are characterized by the service demands at the K devices of the Computer System per data item access/lock request: s_k, $0\le k\le K-1$. Note that a more comprehensive queueing network model for the multiprogrammed computer system incorporating other workloads can be handled under this modeling framework and would only affect Level 3 of the analysis (see Section 3.3).

3. Analysis of the Model

An analysis of the model described in Section 2 using hierarchical decomposition similarly to [19] is presented at this point. The analysis is carried out from top to bottom in the following four sections. The analysis at each level uses quantities derived at lower levels. Also the details of the probabilistic analysis of the Data Contention Model appears in the Appendix. The iterative solution approach is discussed in Section 3.5.

3.1. User Response Model (Level 1)

The final stage of analysis deals with computing user response times. The Computer System is specified by its mean throughput as a function of Activated transactions ($\mu(V)$). Since only V=min(A,W) transactions can be activated, we compute $\mu(V)$, $1\le V\le W$. Note that $\mu(V)=\mu(W)$, $V>W$. The behavior of the system can be represented by a one-dimensional birth-death process, where the arrival (birth) process is random or terminal-driven. The birth-death model can be solved easily yielding the state probabilities. The mean transaction response time (R) can be computed, using the state probabilities and applying Little's law. The computation cost at this level is determined by the cost of solving the birth-death model.

3.2. Transaction Throughput Model (Level 2)

This stage of analysis entails the computation of the effective system throughputs $\mu(V)$, $1\le V\le W$. There are J Eligible and V-J Blocked transactions in a closed system with V Activated transactions such that the completion of each transaction results in the immediate activation of a new transaction. The behavior of the system can be represented by a Markov chain with V states, where the number of Eligible transactions (J) determines the state.

The transition rate from state J (S_J) is determined by the completion rate (throughput) of *elementary transactions*: t(J), which is computed at the Hardware Contention Model (Level 3). The following transitions are possible from S_J as implied by Figure 2-1:

1. Successful lock request with probability $P_a(J|V)$: $S_J \rightarrow S_J$,
2. Conflicting lock request without causing a deadlock with probability $P_c(J|V)$: $S_J \rightarrow S_{J-1}$.
3. Transaction restart due to a deadlock with probability $P_r(J|V)$: $S_J \rightarrow S_I$, $I>J$.
4. Transaction completion with probability $P_t(J|V)$: $S_J \rightarrow S_I$, $I\ge J$.

The routing probabilities for elementary transactions: $P_a(J|V)$, $P_c(J|V)$, $P_r(J|V)$, and $P_t(J|V)$ are derived in the Appendix based on the analysis carried out for the Data Contention Model (Level 4).

The transition rate from S_J to S_I is given by:

$$r_{tran}(J,I|V)=P_{tran}(J,I|V)t(J) \qquad 1\le I\le V \qquad (3)$$

where $P_{tran}(J,I|V)$ is the transition probability from S_J to S_I at the completion instants of elementary transactions and is obtained by solving the Data Contention Model (Level 4). The balance equations for the state probabilities of the system are given as follows:

$$P_S(J|V)t(J) = \sum_{I=1}^{V} P_S(I|V)r_{tran}(I,J|V) \quad 1\le J\le V \tag{4}$$

and solved to obtain the steady-state probabilities $P_S(J|V)$, $1\le J\le V$.

The throughput of elementary transactions is:

$$\mu_e(V) = \sum_{J=1}^{V} t(J)\ P_S(J|V) \tag{5}$$

The effective transaction throughput under the same condition is given by:

$$\mu(V) = \sum_{J=1}^{V} P_t(J|V)\ t(J)\ P_S(J|V) \tag{6}$$

where $P_t(J|V)$ is the probability that the completed elementary transaction was the last phase of a transaction. When the fraction of restarted transactions is negligible, $\mu(V) \approx \mu_e(V)/(\bar{L}+1)$.

The residence time components of transactions are computed using Little's result:

$$R_a(V) = \frac{1}{\mu(V)} \sum_{J=1}^{V} \min(J,M)\ P_S(J|V) \tag{7}$$

$$R_m(V) = \frac{1}{\mu(V)} \sum_{J=1}^{V} \max(J-M,0)\ P_S(J|V) \tag{8}$$

$$R_b(V) = \frac{1}{\mu(V)} \sum_{J=1}^{V} (V-J)\ P_S(J|V) \tag{9}$$

$$R(V) = \frac{V}{\mu(V)} \tag{10}$$

where $R_a(V)$, $R_m(V)$, $R_b(V)$, and $R(V)$ are the mean residence time at the Computer System, the mean waiting time at the Memory Queue, the mean waiting time at the Lock-Wait Queue, and the total residence time in the system, respectively, when V transactions are activated. The wasted time due to restarts can be obtained as follows:

$$R_d(V) = R(V) - (\bar{L}+1)\ \frac{V}{\mu_e(V)} \tag{11}$$

The computational cost at this level for a given value of V is determined mainly by the cost of solving the balance equations (4).

3.3. Hardware Contention Model (Level 3)

At this level we compute the throughput of the Computer System in processing elementary transactions: t(J). The Computer System model was described in Section 2.2, where we noted that it has a maximum MPL equal to M. The service demands for a transaction that accesses n data items is $S_k = n\cdot s_k$, $0\le k\le K-1$. The service demand for elementary transactions is given then by $b_k(J) = S_k(J)/(\bar{L}+1)$, $0\le k\le K-1$.

The throughputs t(J), $1\le J\le \min(V,M)$ for elementary transactions are computed using one of the available algorithms in [7]. When we have a product-form queueing network model for the Computer System the computational cost to obtain these throughputs is O(KM).

3.4. Data Contention Model (Level 4)

At this level we proceed to compute $P_{tran}(J,I|V)$ the transition probabilities of the Markov chain for the Transaction Throughput Model (Level 2). The routing probabilities $P_a(J|V)$, $P_c(J|V)$, $P_r(J|V)$, and $P_t(J|V)$ defined in Section 3.2 are also computed. The derivation of the probabilities, due to its lengthy nature, is given in the Appendix.

3.5. An Iterative Solution Method

In the decomposition method described in previous sections, we must solve the linear equations at Level 2 and solve Level 3 and 4 for $1\le J\le V$. The computational complexity of the decomposition method increases rapidly with V. To reduce the computational cost, an iterative solution method based on the decomposition method is proposed for a given V as follows. Define A(J|V) as mean of the variation in the number of eligible transactions at the completion of an elementary transaction. A(J|V) can be computed as follows:

$$A(J|V) = \sum_{I=J-1}^{V} (I-J)\ P_{tran}(J,I|V) \tag{12}$$

It can be shown numerically (using the decomposition method) that A(J|V) is a monotonically decreasing function in J and that:

$$A(J|V) = 0 \qquad \text{for } J = \bar{J} \tag{13}$$

where $\bar{J}$ denotes the mean state at the completion instants of elementary transactions. The interpretation of equation (13) is that the system's mean state is its balance point, such that the system tends to stay there [3]. For fractional values of J, A(J|V) is defined as a linear interpolation of the neighboring two states $\lceil J\rceil, \lfloor J\rfloor$:

$$A(J|V) = A(\lceil J\rceil)P_H(J) + A(\lfloor J\rfloor)P_L(J) \tag{14}$$

where $P_H(J) = J-\lfloor J\rfloor$ and $P_L(J) = 1-P_H(J)$.

Equation (13) can be solved using the bisection method, since A(J|V) is a monotonically decreasing function in J. This implies that the computation at the Data Contention Model (Level 4) needs to be carried out only for values of J encountered during the iteration. The number of iterations required is bounded by log_2V, and the reduction in computational cost is therefore proportional to V/log_2V. The QNM at Level 3 need only be solved for the two states $\lceil J\rceil$ and $\lfloor J\rfloor$. The need to solve the set of linear equations at Level 2 is also obviated and substituted by the computation of throughput when the system is in its mean state.

In order to compute the throughput of elementary transactions, we first compute the mean interdeparture time of elementary transactions when the system is at its mean state $\bar{J}$. This can be accomplished by noting that $P_L(\bar{J})$ and $P_H(\bar{J})$ are the fractions of elementary transactions that leave behind $\bar{J}$ transactions, i.e., the probabilities pertain to *departure instants*. It follows:

$$\frac{1}{\mu_e(V)} = \frac{P_H(\bar{J})}{t(\lceil \bar{J} \rceil | V)} + \frac{P_L(\bar{J})}{t(\lfloor \bar{J} \rfloor | V)} \qquad (15)$$

The probability of termination at an elementary transaction completion is obtained by interpolation:

$$P_T(V) = P_t(\lceil \bar{J} \rceil) P_H(\bar{J}) + P_t(\lfloor \bar{J} \rfloor) P_L(\bar{J}) \qquad (16)$$

The effective system throughput at the Transaction Throughput Model is computed by:

$$\mu(V) = P_T(V)\ \mu_e(V) \qquad (17)$$

instead of equation (6) in Section 3.1.

4. Validation and Results

We first define the parameters for the test case. A summary of validation results and comments about the accuracy of the solution methods are then given. After ascertaining that the iterative method has an accuracy comparable to the decomposition method (see explanation in Section 4.1) we proceed to use this technique in reporting further results. We proceed to discuss the effect of varying the granularity of locking, database size, number of transactions, and the no-waiting policy, i.e., transactions making conflicting lock requests are restarted.

4.1. Parameters of the Test Case

A database with D=1024 data items was considered. The granularity of locking was varied such that the number of locks was N=32, 64, 128, 256, 512, and 1024. The number of data items accessed by a transaction (n) was set to 2, 4, 8, 16, and 32 to investigate the effect of transaction size. The Computer System consists of 10 devices disks and the service demands at all devices are the same (balanced): s_k=0.002, $0 \le k \le K-1$. The maximum MPL of the Computer System was set to M=15 and the number of Activated transactions (V) was varied.

4.2. Validation

The hierarchical nature of our model lends itself to hierarchical validation (from bottom to top), such that inaccuracies at each level can be rectified by adopting a more accurate model at the appropriate level. In this manner inaccuracies are not allowed to propagate to higher levels, where the tracing of the sources of inaccuracy is more difficult. Due to space limitation, we consider a system with a fixed number of Activated transactions (V) since the open model at Level 1 has been shown to be accurate elsewhere [20].

A simulation program was written taking advantage of hierarchical decomposition to reduce the simulation cost. The Computer System was represented by a flow-equivalent service center characterized by its throughput characteristic for elementary transactions: t(J), $1 \le J \le \min(M,V)$. The residence times of active elementary transactions were sampled from the exponential distribution with the mean J/t(J). The hierarchical simulation was validated using a discrete-event simulation and shown to be adequately accurate. In the simulation elementary transactions were specified at the data item request level. The service demands for elementary transactions were $n \cdot s_k/(n+1)$. When a transaction requested a conflicting lock, the deadlock detection routine was invoked. When a cycle was found in the waiting-for-graph, the transaction causing the deadlock was restarted after releasing its locks. The data items (locks) requested by restarted transactions were *resampled*. This has a negligible effect on the performance measures obtained by simulation since the probability of deadlock is expected to be low in the region under consideration.

The analysis of the Data Contention Model (Level 4) is based on the initial estimates that the number of locks held by Eligible and Blocked transactions are $L_e \approx \bar{L}/2$ and $L_b \approx (\bar{L}-1)/2$, respectively. This estimate is quite accurate when lock contention is low [11]. Iteration[2] can be used to improve on the initial estimates for L_e and L_b (see Appendix). The estimates obtained by this iteration used in conjunction with the decomposition (DEC) and iterative (ITR) solution methods are very accurate except for cases of severe data contention [11]. No iteration is required to estimate L_e and L_b when lock contention is not severe. A comparison of blocking and restart probabilities derived here with other results [5, 16] appears in [11].

		Lock Grant		Blocking		Completion		Deadlock	
n	V	SIM	DEC	SIM	DEC	SIM	DEC	SIM	DEC
4	5	.794	.794	.0056	.0062	.200	.200	.00000	.00002
	10	.787	.786	.0145	.0140	.200	.200	.00004	.00003
	15	.779	.778	.0214	.0215	.200	.200	.00016	.00005
	20	.770	.771	.0301	.0291	.200	.200	.00008	.00007
	25	.765	.763	.0353	.0366	.200	.200	.00000	.00009
8	5	.876	.875	.0129	.0136	.111	.111	.0003	.0002
	10	.859	.859	.0300	.0303	.111	.111	.0005	.0005
	15	.843	.842	.0460	.0466	.111	.111	.0005	.0008
	20	.827	.826	.0619	.0625	.111	.111	.0007	.0012
16	5	.914	.914	.0273	.0270	.0580	.573	.0011	.0018
	10	.882	.882	.0581	.0580	.0561	.556	.0039	.0050
	15	.856	.854	.0829	.0823	.0534	.520	.0077	.0114
32	5	.925	.923	.0419	.0413	.0245	.0233	.0082	.0119
	10	.889	.881	.0734	.0772	.0171	.0156	.0201	.0264

Table 4-1: Transaction routing probabilities.

[2] This iteration is different from the one described in Section 3.5.

Table 4-1 shows the routing probabilities at the completion of an elementary transaction for a given value of V. The results show that the probabilistic analysis in the Data Contention Model provides accurate estimates for the routing probabilities. It can be observed that the probability of lock conflict increases linearly with the number of activated transactions minus one (V-1) and the number of lock requests per transaction. The probability of deadlock increases even more rapidly. Since the average number of transaction completions in simulation runs was 50000, the estimate for probability of deadlock is not accurate enough for some cases. When lock contention is very severe, e.g., L=32 and V=10, the probability of deadlock per transaction exceeds 60%. This is atypical of operational databases, but this parameter settings were used to estimate the accuracy of our analysis under heavy data contention.

		Throughput			Mean MPL			Residence Time		
n	V	SIM	DEC	ITR	SIM	DEC	ITR	SIM	DEC	ITR
	5	44.6	44.1	44.1	4.95	4.91	4.92	.113	.113	.113
4	10	64.7	64.6	64.6	9.72	9.63	9.65	.155	.155	.155
	15	76.6	76.4	76.4	14.4	14.2	14.2	.196	.197	.116
	20	78.7	78.1	78.1	18.8	18.7	18.8	.318	.320	.320
	5	21.6	21.1	21.2	4.80	4.64	4.69	.232	.237	.236
8	10	31.1	30.2	30.5	9.07	8.56	8.66	.322	.331	.328
	15	36.1	35.0	35.5	12.5	11.8	11.9	.416	.429	.423
	20	38.2	37.2	38.3	15.7	14.3	14.5	.525	.538	.522
	5	9.78	8.68	9.16	4.21	3.70	3.88	.512	.576	.546
16	10	12.2	10.2	11.2	6.43	5.17	5.50	.820	.980	.897
	15	11.9	9.67	10.8	6.87	5.37	5.70	1.27	1.55	1.39
	5	3.41	2.07	2.14	3.14	2.50	2.50	2.25	2.42	2.34
32	10	1.80	1.52	1.52	3.76	2.88	2.76	6.38	6.58	6.58

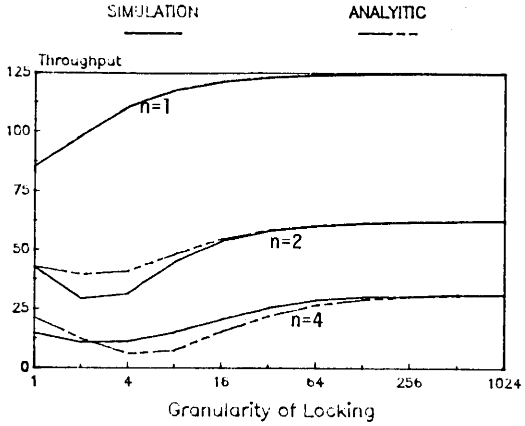
Table 4-2: Transaction throughput, mean MPL, and mean residence time of transactions.

Transaction throughput (μ(V)), the mean number of eligible transactions (mean MPL), and mean transaction residence time (R(V)) are given in Table 4-2. Throughputs given for the simulation are the mean values with relative half-widths about the mean of less than 5% at 90% confidence level. The proposed analytic methods provide reasonably accurate estimates for all three parameters except in the case of heavy data contention. It is observed that the transaction throughput decreases when the number of Activated transactions becomes too large. This is due to the increase in the probability of lock conflict and deadlock. The loss of accuracy for L=32 is due to the inadequacy of the Data Contention Model to estimate lock conflict probabilities for the case of high data contention.

Table 4-3 shows the components of the mean transaction residence times and the mean time wasted due to restarts. The decomposition method provides quite accurate estimates for all components. In heavy data contention, the delay at the Lock-Wait Queue is overestimated due to the inaccuracy in modeling the unblocking probability of blocked transactions. Note that the delay at the Lock-Wait Queue due to blocking is much larger than the time wasted due to restarts, except for the case of heavy data contention. The degradation in performance can be ascribed mainly to blocking due to lock conflicts rather than restarts due to deadlocks. The analysis can be simplified without a significant effect on accuracy (except in the case of high data contention) by setting the probability of deadlock equal to zero and ignoring transaction restarts.

		Active: Ra		Blocked:Rb		Memory: Rm		Restart: Rd	
n	V	SIM	DEC	SIM	DEC	SIM	DEC	SIM	DEC
	5	.111	.111	.001	.002	--	--	.00000	.00001
4	10	.150	.149	.004	.006	--	--	.00003	.00002
	15	.188	.186	.008	.011	--	--	.00016	.00003
	20	.190	.192	.014	.019	.050	.046	.00008	.00006
	25	.190	.192	.019	.029	.108	.099	.00000	.00009
	5	.216	.220	.009	.017	--	--	.00052	.00028
8	10	.283	.283	.030	.048	--	--	.00090	.00079
	15	.330	.336	.069	.093	--	--	.00169	.00195
	5	.420	.423	.081	.123	--	--	.0090	.0107
16	10	.475	.493	.294	.404	--	--	.0412	.0490
	15	.518	.528	.694	.861	--	--	.159	.146
	5	.965	.988	.639	.988	--	--	.320	.456
32	10	1.42	1.49	2.72	3.92	--	--	2.05	.185

Table 4-3: Residence time components.



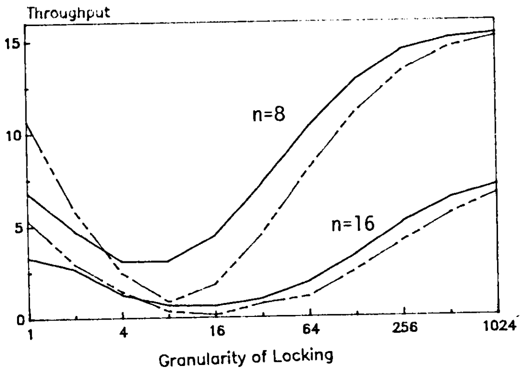


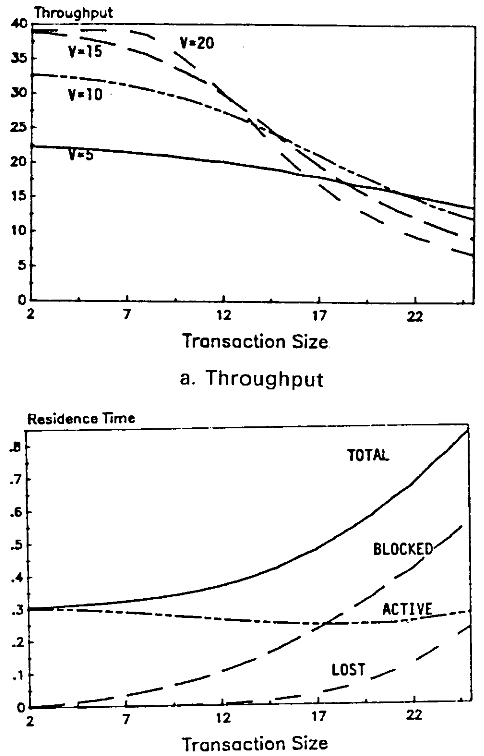
Figure 4-1: Throughput versus granularity of locking with V=10 and s_k=0.001, 0≤k≤9.

In Figure 4-1 we plot transaction throughput versus granularity of locking for different transaction sizes. Simulation results were obtained using the data item access model, while the analysis was carried out using the iterative solution method in Section 3.5 with the lock access model. It is observed that the analysis is accurate for fine and medium granularities of locking as long as transaction sizes are not excessively large, in which case analytic results tend to be pessimistic for fine granularity of locking. This is partially due to inaccuracies in estimating the lock conflict probabilities. For coarse granularity of locking lock requests are usually made in the first stages of processing while the analytic model presumes that they are uniformly distributed over the lifetime of a transaction. The analytic method overestimates the throughput for coarse granularity as was predicted in specifying the database model in Section 2.1.

Analytic results are used in Section 4.2-4.6 with the understanding that they are somewhat inaccurate for high data contention regions, which are of little practical interest.

4.3. Granularity of Locking

Fine granularity of locking is beneficial as can be observed in Figure 4-1. For small transactions the throughput reaches its peak (saturates) very quickly as a finer granularity of locking is adopted. The degree of data contention becomes insignificant for fine granularity and the throughput is determined mainly by hardware resource contention. For large transactions the throughput initially drops in coarse granularity due to increased restarts as the number of locks increases, i.e., the system progresses from serial processing to concurrent processing with high degree of conflicts. The throughput eventually improves for our test case since the largest transactions with n=16 access a small fraction of the database granules.



a. Throughput

b. Residence time components

Figure 4-2: Performance measures vs transaction sizes.

4.4. Transaction Size

Figure 4-2 shows the transaction throughput and residence time components as a function of transaction size. For small transaction size, few transactions are blocked due to lock conflicts and the throughput is determined mainly by hardware contention. The residence time at the Lock-Wait Queue and the time wasted due to restarts are insignificant. When transaction size increases, the effect of lock conflicts becomes significant. The mean MPL and throughput decrease with transaction size. Up to medium size transactions, the performance degradation is mainly due to lock conflicts since the increase of residence time is mainly due to the increase of blocking time. For very large transactions, the mean MPL is independent of the number of Activated transactions (V). The residence time at the Lock-Wait Queue increases rapidly with transaction size as shown in Figure 4-2.b to the extent that there is a drop in mean MPL, which also results in a drop in active residence time at the Computer System. The wasted time in this figure corresponds to the sum of active and blocked time spent in the system by transactions which were restarted.

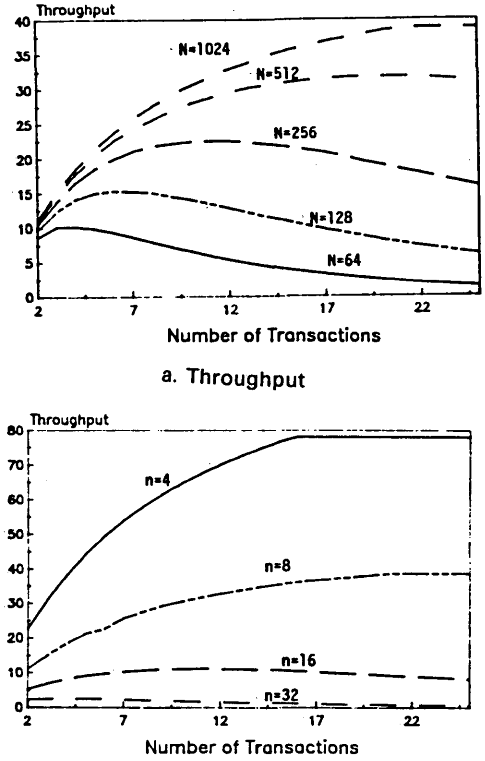
4.5. Number of Transactions

The effect that the number of transactions (V) has on the system performance was discussed qualitatively in Section 2.2, and it was noted that increasing V potentially results in an increase in the effective MPL. On the other hand, lock conflicts are more probable due to the fact that more locks are held by transactions. To investigate the effect of varying number of transactions quantitatively, we plot throughput versus V for different granularities of locking in Figure 4-3.a. In Figure 4-3.b we plot the throughput as a function of V for different transaction sizes.

From Figure 4-3.a, it is observed that system throughput increases with V for fine granularity of locking ($V \leq 10$ and N=256, 512, 1024) for small transactions. On the other hand a degradation in performance is observed for coarse granularity of locking, which is attributable to excessive lock conflicts.

It follows from Figure 4-3.b that increasing V initially results in an increase in throughput. On the other hand, for large transactions (n>8) a point of diminishing returns is soon reached and performance actually degrades. This is because of the increase of lock conflicts, such that the mean MPL decreases with increasing V. This thrashing behavior phenomenon due to lock conflicts was also observed in [15, 18].

Degradation in performance is possible in transaction processing systems with very high MPL's even for fine granularities of locking, as can be observed in Figure 4-3 for large V. It should be noted, however, that the



a. Throughput

b. Throughput

Figure 4-3: Performance measures vs number of transactions.

probability of lock conflict per transaction is proportional to V-1 and the second power of transaction size (n) [5]. Therefore the combination of high V and n has the potential of inducing thrashing and W can be used for congestion control in this case.

4.6. The No-Waiting Policy

At this point we consider the no-waiting policy, according to which a transaction is restarted whenever there is a lock conflict. The no-waiting policy obviates the need to check for deadlock, since there are no blocked transactions. On the other hand, the processing attained by a transaction is lost when it is restarted.

In Table 4-4 we show the transaction throughput versus the number of transactions (V) for different transaction sizes for the waiting (our original model) and the no-waiting policy considered in [15], where a transaction encountering a lock conflict is replaced by a new transaction (resampling of locks). It can be observed that the performance of the no-waiting policy is inferior to the waiting policy with the exception of small transactions. This can be ascribed to the fact that the analysis in [15] considers transaction processing in a system with no hardware contention. A more detailed analysis of the no-waiting case appears in [12].

5. Conclusion

Two methods based on decomposition and iteration were proposed in this paper for analyzing the performance of a centralized database with dynamic locking scheme. The iterative solution method was introduced to reduce the computational cost of the decomposition method by estimating the mean number of eligible transactions, rather than their distribution (as obtained by the decomposition method). Simulation was used to validate both methods, which were shown to be of adequate accuracy except for systems with excessive data contention.

n	Case	V=5	V=10	V=15	V=20
2	Waiting	89.06	131.0	155.3	156.2
	No Wait	89.05	130.8	154.8	154.3
4	Waiting	44.09	64.59	76.43	78.11
	No Wait	44.02	63.73	76.43	78.11
8	Waiting	21.23	30.53	35.49	38.33
	No Wait	21.01	28.72	31.68	29.46
16	Waiting	9.163	11.15	10.80	9.462
	No Wait	8.758	9.761	8.982	7.108
32	Waiting	2.530	1.846	1.251	0.838
	No Wait	2.328	1.531	0.975	0.690

Table 4-4: Throughputs for Waiting and No-Waiting cases.

We also investigated the effect of varying system parameters on system performance. Our conclusions generally concur with those of other researchers. The same framework can also be used to study the effect of exclusive/shared locks, update/readonly transactions [11] and to compare the performance of concurrency control schemes based on locking, optimistic concurrency control, and timestamp ordering [13].

The main contribution of this paper is the methodology developed for the probabilistic analysis of lock conflict/deadlock probabilities. These are then used to compute transaction routing and transition probabilities for the embedded Markov chain. An advantage of the hierarchical solution method is that the probabilities for the Data Contention Model can be obtained by means of simulation (trace or random number driven).

I. Analysis of the Data Contention Model

This appendix is concerned with computing the transition probabilities among the states of the Markov chain model for the Transaction Throughput Model (Level 2) as explained in Section 3.2. and the lock conflict probability for the Hardware Contention Model. The reader is forewarned that the derivations are carried out in a top-down manner, such that parameters used in earlier derivations are themselves derived later. Also note that the parameter V is elided, since it remains fixed in the following derivations.

The transition probability $P_{tran}(J,I|V)$ depends on whether the completed elementary transaction belongs to a transaction holding k-1 locks and its routing probabilities: $P_a(k)$ {successful lock request}, $P_c(k)$ {blocking}, $P_t(k)$

{completion}, and $P_r(k)$ {deadlock}. These probabilities are computed in (21), (24), (20), and (22), respectively.

Defining $P_{tr}(J,I|k)$ as the probability that transition occurs from S_J to S_I when an elementary transaction holding k-1 locks is completed, we have:

$$P_{tr}(J,I|k)=\begin{cases} P_c(k) & \text{for } I=J-1 \\ P_a(k)+P_t(k)P_{ta}(J,I|k) & \text{for } I=J \\ P_t(k)P_{ta}(J,I|k)+P_r(k)P_{ra}(J,I|k) & \\ & \text{for } I>J \end{cases} \qquad (18)$$

where $P_{ta}(J,I|k)$ (resp. $P_{ra}(J,I|k)$ is the probability that I-J blocked transactions are unblocked when a transaction holding k-1 locks teminates (resp. is restarted after releasing its locks). These probabilities are derived in (33) and (34), respectively. The unconditional transition probability from S_J to S_I is:

$$P_{tran}(J,I|V) = \sum_{k=1}^{L+1} P_{tr}(J,I|k)\ P_i(k) \qquad (19)$$

where $P_i(k)$ is the probability that the completing elementary transaction holds has k-1 locks and is obtained in (25) and (26).

The routing probabilities ($P_a(J|V)$, $P_c(J|V)$, $P_r(J|V)$, and $P_t(J|V)$) are then computed by unconditioning on k using $P_i(k)$ from $P_a(k)$, $P_c(k)$, $P_r(k)$, and $P_t(k)$.

When all transactions access a fixed number of data items: n, $P_t(k)$ is given by:

$$P_t(k) = \frac{P_m(k|n)}{\sum_{h=k}^{L} P_m(k-1|n)} \qquad (20)$$

where $P_m(k|n)$ is the distribution of number of locks requested by a transaction requesting n locks and was given by equation 1 in Section 2.1. Defining $P_{a1}(k)$ as the probability that k'th lock request is successful (which is derived in equation (35)), the probability of lock acquisition $P_a(k)$ can be obtained as:

$$P_a(k) = (1-P_t(k))\ P_{a1}(k) \qquad (21)$$

The probability of deadlock $P_r(k)$ is then:

$$P_r(k) = (1-P_t(k)-P_a(k))\ P_{r1}(k) \qquad (22)$$

where $P_{r1}(k)$ is the probability that a conflict at k'th lock request causes a deadlock and is represented by the sum of probabilities of deadlocks with different cycle lengths:

$$P_{r1}(k) = \sum_{c=2}^{V-J+1} P_{rr}(c|k) \qquad (23)$$

where $P_{rr}(c|k)$ is the probability of deadlock with cycle length c at a conflicting k'th lock request and is derived in equation (37).

The probability of blocking $P_c(k)$ is simply the complement of the other routing probabilities as follows:

$$P_c(k) = 1 - P_t(k) - P_r(k) - P_a(k) \qquad (24)$$

$P_i(k)$, the probability that an elementary transaction holds k-1 locks, is computed as follows:

$$P_i(k) = \begin{cases} 1/A & \text{for } 1\leq k\leq 2 \\ 1/A \prod_{h=2}^{k-1} (1-P_t(h)-P_r(h)) & \text{for } 2\leq k \end{cases} \qquad (25)$$

where where A is the normalization constant. When there are no transaction restarts due to deadlocks (or the fraction of such transactions is negligible) in the finest granularity of locking, then all transactions request exactly n locks, and:

$$P_i(k) = \frac{1}{n+1} \qquad 1\leq k\leq n+1 \qquad (26)$$

which means that the number of locks held by elementary transactions is distributed uniformly over [0,n].

The mean number of locks held by an Eligible (resp. Blocked) transaction is denoted by L_e (resp. L_b). We have:

$$L_e = \sum_{k=1}^{L+1} (k-1)\ P_i(k) \qquad (27)$$

$$L_b = \sum_{k=1}^{L+1} (k-1) \frac{P_i(k)P_c(k)}{\sum_{k=1} P_i(k)P_c(k)} \qquad (28)$$

For finest granularity of locking with small transactions and low data contention:

$$L_e \simeq \frac{\bar{L}}{2} \simeq \frac{n}{2} \qquad (29)$$

$$L_b \simeq \frac{(\bar{L}-1)}{2} \simeq \frac{(n-1)}{2} \qquad (30)$$

This was observed in simulation studies [11] and can also be argued intuitively [5]. This estimates given by equation (29) and (30) were used with fine granularity locking, in which case equation (27) and (28) confirm the initial estimate. In the case of medium granularity of locking an initial estimate for L_e and L_b was chosen and iteration was used to improve the initial estimate. The iteration was carried out from equations (21) through (28) until L_e and L_b converge.

To derive $P_{ta}(J,I|k)$ {resp. $P_{ra}(J,I|k)$} used in equation (18), we first compute $P_{at}(I|b,J,k)$ {resp. $P_{ar}(I|b,J,k)$} as the probability that I-J of b blocked transactions are unblocked from the Lock-Wait queue when k-1 locks are released due to completions {resp. restarts}. $P_{at}(I|b,J,n)$ and $P_{ar}(I|b,J,n)$ are then computed by the following recursion for $1\leq b\leq V-J$, $J\leq I\leq \min(V,J+k-1)$:

$$\begin{aligned} P_{at}(I|b,J,k) &= P_{at}(I|b-1,J,k)(1-P_o(I-J|k)) \\ &\quad + P_{at}(I-1|b-1,J,k)P_o(I-J-1|k) \end{aligned} \qquad (31)$$

$$\begin{aligned} P_{ar}(I|b,J,k) &= P_{ar}(I|b-1,J,k)(1-P_o(I-J-1|k-1)) \\ &\quad + P_{ar}(I-1|b-1,J,k)P_o(I-J-2|k-1) \end{aligned} \qquad (32)$$

with the initial condition of

$$P_{at}(I|b,J,k) = \begin{cases} 1 & \text{for } I=J \text{ and } b=0 \\ 0 & \text{otherwise} \end{cases}$$

$$P_{ar}(I|b,J,k) = \begin{cases} 1 & \text{for } I=J+1 \text{ and } b=1 \\ 0 & \text{otherwise} \end{cases}$$

where $P_o(q|k)$ is the probability that a blocked transaction following q transactions is unblocked when k-1 locks are released and is computed from (41). The second initialization is different from the first one since at least one blocked transaction is unblocked upon the restart of a transaction. Since we have V-J blocked transactions in the Lock-Wait Queue, then probabilities $P_{ta}(J,I|k)$ and $P_{ra}(J,I|k)$ become:

$$P_{ta}(J,I|k) = P_{at}(I|V-J,J,k) \qquad (33)$$

$$P_{ra}(J,I|k) = P_{ar}(I|V-J,J,k) \qquad (34)$$

We now proceed to compute the basic probabilities: $P_{a1}(k)$, $P_{rr}(c|k)$, and $P_o(q|k)$ when lock requests are mutually exclusive, independent and uniformly distributed over the N locks. The key assumptions in our analysis is that (i) an outstanding lock request by a transaction is conflicting equally probably with all busy locks held by other transactions. and (ii) eligible (resp. blocked) transactions hold L_e (resp. L_b) locks. $P_{a1}(k)$, the probability that the k'th lock request is successful, is obtained by:

$$P_{a1}(k) = 1 - \frac{L_e(J-1)+L_b(V-J)}{N-k+1} \qquad (35)$$

where the ratio is the number of locks taken by other transactions divided by the total number of available locks.

We now proceed to derive $P_{rr}(c|k)$, the probability that a transaction requesting k'th conflicting lock causes a deadlock of cycle c. Consider a deadlock involving three transactions T_1, T_2, and T_3 (c=3), where T_1 is the transaction requesting a conflicting lock. A deadlock cycle of length three is made, i.e., $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$, where $T_i \rightarrow T_j$ denotes that T_i is blocked by T_j. $P_{rr}(3|k)$ is then computed by the product of $\text{Prob}[T_1 \rightarrow T_2]$, $\text{Prob}[T_2 \rightarrow T_3 \mid T_1 \rightarrow T_2]$, $\text{Prob}[T_3 \rightarrow T_1 \mid T_1 \rightarrow T_2 \rightarrow T_3]$, and the number of choices to choose T_2 and T_3 among blocked transactions. Note that $\text{Prob}[T_3 \rightarrow T_1 \mid T_1 \rightarrow T_2 \rightarrow T_3]$ is not equal to $\text{Prob}[T_3 \rightarrow T_1]$. $T_3 \rightarrow T_2$ and $T_2 \rightarrow T_3$ can not happen simultaneously since all deadlocks have been resolved before T_1 makes a lock request.

Defining $P_{w1}(k)$, $P_{w2}(k|i)$, and $P_{w3}(k|i)$ as follows:

$P_{w1}(k)$ = Prob[A transaction requesting k'th lock is blocked by a specific blocked transaction | Lock conflict occurred].

$P_{w2}(k|i)$ = Prob[A blocked transaction waits for an eligible transaction which holds k-1 locks | It does not wait for the i other blocked transactions.]

$P_{w3}(k|i)$ = Prob[A blocked transaction is blocked by a specific blocked transaction | It does not wait for the i other blocked transactions.]

$\text{Prob}[T_1 \rightarrow T_2]$ becomes $P_{w1}(k)$, $\text{Prob}[T_2 \rightarrow T_3 \mid T_1 \rightarrow T_2]$ becomes $P_{w3}(k|0)$, and $\text{Prob}[T_3 \rightarrow T_1 \mid T_1 \rightarrow T_2 \rightarrow T_3]$ becomes $P_{w2}(k|1)$. The number of ways to choose T_2 and T_3 from V-J transactions is obtained by (V-J)·(V-J-1). Hence the probability of deadlock involving three transactions when a transaction makes its k'th conflicting lock request is represented as follows:

$$P_{rr}(3|k)=P_{w1}(k)P_{w2}(k|1)(V-J)(V-J-1)P_{w3}(k|0) \qquad (36)$$

The probability of deadlock involving m transactions at k'th conflicting lock request in general is obtained for $2 \leq m \leq V-J+1$ by:

$$P_{rr}(m|k)=P_{w1}(k)P_{w2}(k|m-2) \prod_{i=0}^{m-2} \{(V-J-i)P_{w3}(k|i)\} \qquad (37)$$

$P_{w1}(k)$ is obtained from the mean number of locks held by other transactions as follows:

$$P_{w1}(k) = \frac{L_b}{L_e(J-1)+L_b(V-J)} \qquad (38)$$

Then $P_{w2}(k|i)$ and $P_{w3}(k|i)$ are then obtained by:

$$P_{w2}(k|i) = \frac{k-1}{L_e(J-1)+L_b(V-J-i-1)+k-1} \qquad (39)$$

$$P_{w3}(k|i) = \frac{L_b}{L_e(J-1)+L_b(V-J-i-1)+k-1} \qquad (40)$$

Finally $P_o(q|k)$,the probability that a transaction is unblocked following q transactions when k-1 locks are released, is computed as follows:

$$P_o(q|k) = \frac{k-1-q}{(J-1)L_e+(V-J-1)L_b+k-1} \quad \text{for } q<k-1 \qquad (41)$$

References

1. Carey, M. *Modeling and evaluation of database concurrency control algorithms.* Ph.D. Th., University of California, Berkeley, September 1983.

2. Chesnais, A., Gelenbe, E., and Mitrani, I. "On the modeling of parallel access to shared data." *Comm. ACM 26*, 3 (March 1983), 196-202.

3. Courtois, P.J.. *Decomposability: Queueing and Computer System Applications.* Academic Press, 1977.

4. Date, C.J.. *An Introduction to Database Systems, Vol. II.* Addison-Wesley, 1983.

5. Gray, J. N., Homan, R., Obermack, R., and Korth, H. A straw-man analysis of waiting and deadlock. Tech. Rept. IBM Research Center Report RJ 3066, February, 1981. Also in Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks, 1981.

6. Irani, K.B. and Lin, H.L. Queueing network models for concurrent transaction processing in a database system. Proc. ACM SIGMOD Conf. on Management of Data, ACM, Boston, Mass., June, 1978, pp. 134-142.

7. Lavenberg, S.S. (Ed.). *Computer Performance Modeling Handbook.* Academic Press, 1983.

8. Lin, W.T.K. and Nolte, J. Communication delay and two phase locking. Proc. 3rd Int'l Conf. on Distributed Computing systems, IEEE, Miami, Florida, October, 1982, pp. 502-507.

9. Ries, D.R. and Stonebraker, M.R. "Locking granularity revisited." *ACM Trans. Database Systems 4*, 2 (June 1979), 210-227.

10. Ryu, I.K. and Thomasian, A. Analysis of database performance with dynamic locking. Tech. Rept. RC 11428, IBM Research Report, October, 1985.

11. Ryu, I.K. *Performace evaluation of concurrency control in database systems.* Ph.D. Th., Dept EE-Systems, University of Southern California, 1985.

12. Ryu, I.K. and Thomasian, A. Performance study of a centralized DBMS with dynamic locking using no-waiting policy. Submitted for publication.

13. Ryu, I.K. and Thomasian, A. Performance comparison of concurrency control schemes for centralized DBMS. Submitted for publication.

14. Shum, A.W. and Spirakis, P.G. Performance analysis of concuurency control methods in database systems. In *Performance 81*, Kylstra, F.J., Ed.,North-Holand, 1981, pp. 1-19.

15. Tay, Y.C. A mean value performance model for locking in databases. Tech. Rept. TR-04-84, Harvard University, 1984.

16. Tay, Y.C., Suri, R., and Goodman, N. "A mean value performance model for locking in databases." *Journal ACM 32*, 3 (July 1985), 618-651.

17. Tay, Y.C., Goodman, N., and Suri, R. "Locking performance in centralized databases." *ACM Trans. Database Systems 10*, 4 (December 1985), 415-462.

18. Thomasian, A. An iterative solution to the queueing network model of a DBMS with dynamic locking. Proc. 13th Computer Measurement Group Conf., San Diego, CA, December, 1982, pp. 252-261.

19. Thomasian, A. and Ryu, I. K. A decomposion solution to the queueing network model of the centralized DBMS with static locking. Proc. ACM Conf. on Measurement and Modeling of Computer Systems, ACM, Minneapolis, MN, August, 1983, pp. 82-92.

20. Thomasian, A. "Performance evaluation of centralized databases with static locking." *IEEE Trans. Software Engineering 11*, 2 (April 1985), 346-355.

A Graphical Interface for Specification of Extended Queueing Network Models

J. B. Sinclair
S. Madala

Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77251-1892

ABSTRACT

GUIDE is a graphical user interface to a performance evaluation tool based on extended queueing network models. The interface allows a user to specify the structure of a model by creating a graphical representation of the objects in the model and their interconnections. Additional information about the specific objects and/or the model as a whole is input through a set of dialog windows that adapt to the specific context and prior information entered by the user. This enforces the creation of complete models that are syntactically correct. We briefly describe the implementation of GUIDE and its modeling capabilities, and illustrate its usage through a small example.

1. Introduction

GUIDE (the Graphical User Interface and Dialog Editor) is a tool for constructing extended queueing network models. It is part of a simulation-based modeling and performance evaluation package called GIST developed at Rice. [1] In designing GUIDE (and GIST), we had several objectives:

(1) The interface should be easy to learn and simple to use.

(2) It must be powerful in its descriptive capabilities, to allow very complex systems to be modeled.

(3) Users should not be able to create a syntactically incorrect model specification.

(4) The user should be able to easily specify large models (dozens or even hundreds of components).

(5) The interface should be flexible in allowing the user freedom to construct a model with as few restrictions as possible, other than syntax enforcement.

(6) The interface implementation hardware should be inexpensive to make its use practical in an educational environment.

In this paper, we discuss both the implementation and use of GUIDE. We also describe some proposed extensions to the current implementation that will improve its utility and bring us closer to achieving these objectives.

2. Background

Queueing models (QN's) and extended queueing models (EQN's) can be specified in several ways. The most primitive way is to describe them in a conventional programming language such as FORTRAN, C, or PASCAL. This approach offers complete generality but, in the case of simulation, requires that the programmer also construct the mechanism for the execution of the simulation model. To avoid this we can use a simulation language, which is often an extension of a conventional programming language that provides high-level modeling constructs and a run-time environment for model execution. [2, 3, 4, 5, 6]

An alternative to this "language-oriented" approach, and the approach chosen for GIST, is to use an interface tailored to the specification of EQN's. In this "transaction-oriented" approach, the interface deals with the model at the same level of abstraction as the user. Examples of this approach include RESQ, [7] QNA, [8] QNAP2, [9] XL, [10] SNAP, [11] SuperNet, [12] PANACEA, [13] PERFORMS, [14] BEST/1, [15] COPE, [16] PAWS, [17] PLANS, [18] NUMAS, [19] the Performance Analysis Workstation, [20] and STEP-1. [21] There are several advantages to a transaction-oriented approach. Since the tool allows the user to describe the model directly in terms of the abstractions used to construct the model, it is easier to learn and to use the specification tool. It is also easier to insure that the specification matches the model. Some types of errors are avoided completely or can be detected at model specification time. A major advantage is that the model specifications are easier to understand and to modify.

In most cases the user of one of these performance tools interacts with the tool through a textual interface, but graphics interfaces have been proposed or implemented for at least four of them: PAWS, [22] RESQ, [23] the Performance Analysis Workstation, and NUMAS.

3. Description of GIST

GIST allows specifications of EQN models through a graphical or a textual interface and produces executable images to simulate a model of the system under consideration. The EQN models are specified as a network of high level objects drawn from a set of object types. GIST is based on CSIM, a subroutine package that provides runtime support environment for discrete event simulation. [24] The major parts of GIST are two distinct user interfaces that accept specifications from the user, a translator that generates CSIM source code, and a library of object routines written in CSIM to model the various object types (Fig. 1).

3.1. User Interfaces

The two interfaces have similar modeling capabilities, but one is graphics-oriented (GUIDE) while the other (TIDE) is intended for use on terminals without graphics.

3.1.1. GUIDE

A natural way to specify many aspects of an EQN model is graphical. GUIDE allows users to assemble EQNs from elements of a set of icons representing the various object types, interconnected by routing paths. Users can define or edit object specific parameters through the use of "dialog" windows.

CH2345-7/86/0000/0709$01.00 © 1986 IEEE

GUIDE was implemented on a 512K Macintosh personal computer because the window environment, mouse input, and pull down menus provided a friendly, easy-to-learn, easy-to-use interface and the software tool base made the prototype development much simpler. Also, a Macintosh is a relatively inexpensive graphics workstation. GUIDE runs standalone; no host computer is needed during the specification process. This reduces the load on the mainframe that executes the simulation model. The output of GUIDE is a file containing the EQN model specifications. The file is transferred to the host for processing by the translator.

3.1.2. TIDE

The Textual Interface and Dialog Editor (TIDE) is provided as an alternative when a Macintosh is unavailable. [25] It runs on a Unix-based system and is intended to be used with a VT100 or equivalent terminal. It uses the CURSES windowing library routines for efficient screen management.

The user is guided through a series of hierarchically structured menus. A unique feature of this interface is that, unlike the dialog interfaces for some performance evaluation tools such as RESQ, interaction is menu-driven and conducted via overlapping "windows" on the screen. This helps the user maintain a sense of perspective of the current context of the specification process. The output of TIDE is again a specifications file to be processed by the translator.

3.2. Translator

The translator is the link between the user interface and the object routines. Its input is the specifications file created by the user interface and its output includes CSIM source code (see below). The translator produces as output the following files: a main file, a definitions file, a "make" file and optionally a report file. The main file contains code for initialization and calls to the appropriate object routines. The definitions file contains declarations of special CSIM constructs. The "make" file is provided to simplify the task of creating the final executable image. The report file contains a user-readable summary of the EQN model. A parser, generated using the Unix facilities lex and yacc, is used for parsing the user-specified conditions.

3.3. CSIM

CSIM is a subroutine package, written in C, that provides a runtime support environment for discrete event simulation. It is based on the process interaction approach, in which related sets of events are combined into a single process. A simulation specification in CSIM consists of a set of procedures, one for each type of process, and declarations of instances of special constructs such as conditions, semaphores, and state variables. CSIM provides routines to manipulate both processes and these special constructs.

4. GIST Objects

One of the approaches used in attempting to meet the goal of providing GIST users with a high-level model specification capability is the implementation of a set of object types that can be used for realistic and detailed representation of relevant properties of systems of interest. An EQN model can be built from a number of objects, each object of a predefined object type. The object types in a GIST model are briefly described below.

SOURCE: Jobs of a specific jobclass are created at a SOURCE object. The creation of a job may be condition-dependent, or jobs may be created at intervals determined by an inter-generation time distribution.

SINK: Jobs arriving at a SINK object are removed from the network.

ALLOCATE: An ALLOCATE object assigns units of a specific resource to the visiting jobs that request it. Jobs request resources in integer amounts according to jobclass-dependent distributions. Sometimes, a request from a job cannot be satisfied immediately due to an insufficiency of the resource. In that case, the job is delayed until its request is met in accordance with an allocation policy that is followed at the object.

DEALLOCATE: At a DEALLOCATE object, all the units of a specific resource held by visiting jobs are reclaimed and returned to the pool of available resources.

When jobs visit a CREATE object, an integer amount of a particular resource is produced. The amount of resource created is in accordance with a jobclass-independent distribution.

DESTROY: At a DESTROY object, all the units of a specific resource held by visiting jobs are removed, but they are not returned to the pool of available resources.

FORK: A FORK object creates a new job when a job of a specific jobclass visits it. The new job is defined to be either a **child** job or a **peer** job in relation to the visiting job. The created job can have a jobclass different from that of the visiting (creator) job.

JOIN: When a job that has a parent or child relationship with any other job reaches a JOIN object, it waits there until its child or parent reaches there. When the two meet, the child job terminates and the parent job continues through the network. When a job has more than one child jobs in the network, it waits at a JOIN object for the arrival there of its most recent unterminated child job. Jobs that have no child jobs or parent jobs are unaffected by a visit to a JOIN object and are not delayed there.

PROBE: A PROBE object collects user-specified statistics from the visiting jobs. The jobs themselves are unaffected by visit to a PROBE object.

SWITCH: SWITCH objects perform job routing. The routing decision made at a SWITCH object can be based on a condition-evaluation, stochastic selection, or a static, jobclass-based routing specification. A SWITCH object does not delay the jobs that visit it.

QUEUE, SERVER: The two functions, queuing and service, of a QUEUE object in a QN model can be performed separately by QUEUE and SERVER objects in a GIST EQN model. QUEUE objects perform only the queuing function (without service). Jobs joining a QUEUE object wait there while service is unavailable. Jobs at a QUEUE object conform to that object's **queuing discipline**, a policy which determines the order in which jobs become eligible to leave the object. The next job eligible to leave is said to be at the **head** of the QUEUE.

QSERVER: A QSERVER object in an EQN model is equivalent to a QUEUE object in a QN model. It is an object where both the queuing and service functions are performed. It is computationally more efficient to use a QSERVER if the generality provided by separate QUEUE and SERVER objects is not required in an EQN model.

We will use the term QUEUE to refer to the EQN model object. If the discussion concerns a QUEUE object type as in a QN model, we make the context explicit.

5. Implementation of GUIDE

EQN models are characterized by a network of objects and their interconnections. An analyst almost always draw a pictorial representation of an EQN model before actually specifying it to a typical modeling tool. Such representations or block diagrams are much more readable than textual information accepted by EQN modeling tools. Then the analyst in effect manually translates the graphical representation, augmenting it with additional information, into a form recognized by a modeling tool or programming language. The translation process is time consuming and is a potential source of errors. An obvious way to bypass this step is to provide the user with an interface that directly accepts graphical specifications. However, certain information associated with EQNs is necessarily specified textually. Examples of such information include the attributes of each object such as parameters of probability distributions, names of objects, and initial conditions. The graphical interface shields the user from implementation details and minimizes the possibility of errors during the specification process. It lets the user concentrate on the model specification rather than on the tool itself.

GUIDE is written in the C language, using the Stanford University Mac C (sumacc) development software. [26] It is tailored to the Macintosh and presently is not portable to other computers/graphics terminals. Users familiar with some typical Macintosh application programs can learn to use GUIDE in a few minutes. Provision has been made to give on-line help about the various objects of GIST and about GUIDE itself, but at present this feature is unimplemented.

The minimum hardware requirements for the graphical interface are a high resolution display, sufficient memory to specify large models, and a graphical input device such as a mouse, trackball or digitizing pad along with a keyboard. Several workstations and personal computers satisfy the above requirements. We chose the Macintosh computer primarily because it was the least expensive system which meets these minimum requirements, and is more widely available at Rice than more sophisticated (and more expensive) workstations. Moreover, the Macintosh has an excellent library of graphics routines and a well defined standard user interface that reduced our implementation effort considerably.

The software for GUIDE can be divided into two parts, the graphical input package and a set of specification routines. The input package allows the user to create a graphical representation of the network topology, i.e., the objects and their interconnections. It also handles all generic system functions such as saving partial or complete specification files, and opening existing files. The specification routines allow the object-dependent data to be entered in windows tailored to each object type. These routines have been implemented in a modular fashion, making it easy to modify existing object types or add new object types.

GUIDE presents the user with a set of menus, a working window and an options window (Fig. 2). The working window is the region of the screen displaying the part of the larger work area in which the user is currently creating the model. The work area can be scrolled horizontally and vertically in the working window to aid in specifying large models. The options window contains graphical representations (icons) for each object type available in GIST, as well as an interconnect option. The icons for the object types are given in Fig. 3.

There are seven pulldown menus: 1) the apple menu, 2) the file menu, 3) the edit menu, 4) the help menu, 5) the specify menu, 6) the transfer menu, and 7) the debug menu. Moving the cursor over any menu item and pressing the mouse button results in the display of a set of items grouped under this menu. Any item within the menu can then be selected by moving the cursor over it and releasing the mouse button. The apple menu is common to most Macintosh application programs. The transfer and debug menus will not appear in future implementations of GUIDE and will not be discussed.

The file menu lets a user invoke the following functions: open a new file, open an existing file, close the current file, save the current specifications in a file, save the current specifications in a different file, revert back to the last saved version of the specifications, check for partial correctness of specifications, and quit the program.

The edit menu allows a user to delete objects or interconnections. Other items in this menu are the standard cut, copy and paste items found in other Macintosh applications. These items at present work only with concurrent mini-applications (Desk Accessories) that are invoked via the apple menu.

The help menu is used to invoke on-line help. Currently this feature is not implemented.

The specify menu lets users specify initial jobs, runtime parameters, and initial quantities of passive resources. In addition, it lets user view information about job classes and objects.

6. Example of the Use of GUIDE

We shall now illustrate the usage of GUIDE through a small example. Consider a computer system which services jobs of two different classes, batch and interactive. Batch jobs arrive from the external world, receive service from the system, and depart. Interactive jobs continually circulate in the system between terminals and the CPU. Both classes of jobs contend for primary memory before they receive service and release the memory allocation after service. The system consists of two processors, a "fast" cpu and a "slow" cpu. In the interest of high throughput the batch jobs are processed on the fast cpu, while the interactive jobs are processed on both processors, pre-empting the batch jobs whenever necessary to reduce response time.

This system can be modeled using the GIST object types SOURCE, SINK, ALLOC, DEALLOC, QUEUE, SERVER, SWITCH and PROBE. The SOURCE and SINK objects model arrival and departure of batch jobs, the ALLOC and DEALLOC objects model acquisition and release of primary memory by both classes of jobs, the QUEUE and SERVER objects model the waiting jobs and the two processors, and the SWITCH object models job routing.

The model might be specified using GUIDE as follows. Positioning the cursor over an icon in the options window and pressing the mouse button selects and highlights the icon by inverting it. We now can specify a new object of the selected type by moving the cursor into the working window without releasing the mouse button (Fig. 4). During this "dragging" process an outline of a box enclosing the icon follows the cursor movement.

After some or all of the required objects have been dragged into the working window we can specify the interconnections between them using the interconnect option in the options window. The cursor changes shape to a cross to keep the user aware of this option. To interconnect two objects we click first on the source icon and then on the destination icon. The source icon represents the object from which jobs depart and the destination icon represents the object to which jobs are routed. In Fig. 5 the source icon and the destination icon represent the SOURCE type object and the ALLOC type object, respectively. During this process an "elastic" line anchored at the source object tracks the motion of the cursor. Interconnections with more than one line segment can also be specified by clicking in succession on the source icon, intermediate points that define the line segments, and the destination icon. The completed EQN model diagram for the example appears in Fig. 6.

Object type specifications for an object can be entered at any time after the icon for the object has been placed in the working window. Clicking twice in succession (double clicking) on an icon brings up a "dialog" window specific to the object type represented by that icon. These windows contain boxes to accept object names, items that permit selection among a number of alternatives, items that accept yes/no options and items that start immediate actions.

As an example, double clicking on the SOURCE icon in Fig. 6 brings up the SOURCE specification dialog window in Fig. 7. This window contains text boxes to accept the name of the object, the class of jobs to be generated by this SOURCE and initially boxes for distribution parameters. The interface is designed to hide as much irrelevant detail as is possible from the user. If the conditional generation option is chosen, the distribution related items disappear and a text box to accept a condition appears in their place.

Other items in the dialog window allow the user to specify the statistics that can be collected at this SOURCE. Clicking on the OK button confirms the specifications for the object. This closes the dialog window and takes the user back to the working window. Alternatively, clicking on the Cancel button undoes all changes made to this object's specification and returns the user to the working window. This undo feature is available in all object specification dialog windows.

A noteworthy feature of the interface is that the dialog windows adapt to different interconnections in the model being specified. For example, the appearance of the QUEUE specification dialog window depends upon the number of SERVER type objects to which the QUEUE type object is connected. The dialog window for the upper QUEUE of Fig. 6. ("Interact queue"), which is connected to two SERVER type objects, is shown in Fig. 8 and differs from the dialog window for the lower QUEUE ("Batch queue") shown in Fig. 9, in that it has additional items for specifying server selection. Clicking on the "Edit" button brings up another dialog window (Fig. 10) for the specification of server selection. GUIDE's context-dependent syntax is again illustrated in the dialog windows for the two SERVER objects (Figs. 11 and 12).

Modifying an already existing model specification is relatively easy with GUIDE. Objects and interconnections can be added or deleted at any time during the specification process. It is not necessary to have all the objects and interconnections ready before specification for some of the objects can begin. Some checking is done during additions and deletions so that internal data structures remain consistent at all times. Icons can be moved on the working area at any time, even after they have been connected to other icons, and the connections will be maintained. Multiple-segment routing paths can be manipulated by moving the point at which any two segments join.

Details which are not object-specific but are pertinent to the model as a whole, such as passive resources, run time parameters and initial jobs, can be specified in other dialog windows that are invoked by selecting items from the "Specify" menu. The Runtime parameters dialog window is shown in Fig. 13.

When specification of the model is complete, it can be saved in a file and transmitted to the host computer for processing by the translator component of GIST.

7. Summary and Discussion

GUIDE has only recently been made available for general use. However, even our limited experience with it offers some basis for judging how well our objectives have been achieved. We believe that it is extremely easy to learn to use for anyone with an understanding of extended queueing network models. The icons are similar or identical to those found in the literature. Virtually all of the object types that are typically found in EQN models are supported by GUIDE and by GIST, although in some cases the semantics of the objects have been modified. These include objects for modeling active and passive resources, concurrency, and open networks. New object types (principally the QUEUE, SERVER, SWITCH, and PROBE objects) provide additional modeling capabilities and/or make model specification easier.

The user cannot construct a complete model that is syntactically incorrect. By complete we mean a model in which all object inputs and outputs are connected to other objects. At any point in the specification process, GUIDE requests and a user can only enter information that is appropriate for the context. Menus for entering object information adapt to user responses.

Support for large models is currently limited to the ability to scroll within a large virtual screen. This causes an undesirable "peephole" effect, in that a user can only see a part of the model at one time. The problem is made more severe by the small size of the working window, which in turn is due to the small screen size of the Macintosh computer.

The user has a great deal of flexibility in creating and editing model specifications. Objects can be inserted into the model at any time. Information for an object can be entered when the object is inserted or later, and all of the information does not have to be entered at the same time. Icons and connections between icons can be deleted at any time, and icons (even with connections) can be moved around in the virtual workspace. The model specification can be saved for later additions and/or modification.

The major deficiencies of GUIDE are in the area of large model specification. Currently, GUIDE has no concept of a submodel, and hierarchically structured models are not possible. Scrolling is useful, but forcing the user to always see every part of the system at the same level of abstraction may result in an amount of detail that confuses rather than clarifies the nature of the model and the interactions among its components. Also, each model must be completely constructed from the set of GIST primitive objects. The user should be able to use previously specified models as higher level abstractions when constructing new models, and nesting of model definitions should be possible. The most important extension that we plan for GUIDE (and GIST) is the inclusion of single-input, single-output submodels.

Another useful feature for modeling systems, especially large ones, is that of a replicated object. That is, a user should be able to declare an instance of an object and then specify that the object is to be replicated several times. For instance, in a model of an interactive computer system, the set of all terminals/users could be modeled as a single replicated QSERVER. Enhancements to CSIM currently under way will make the task of implementing this feature easier and the result will be more efficient than it would be under the current CSIM implementation.

Overall, GUIDE offers an interesting and efficient alternative to more traditional methods of extended queueing network specification. We are in the process of gaining experience with its use in both courses and research and we will use that experience to identify other enhancements that will increase its utility.

References

[1] J.B. Sinclair, K.A. Doshi, and S. Madala, "GIST: The Graphical Input Simulation Tool," TR 8511, Department of Electrical and Computer Engineering, Rice University, Houston, TX 77251-1892, May 1985.

[2] O.-J. Dahl and K. Nygaard, "SIMULA - An ALGOL-Based Simulation Language," *CACM*, Vol.9, no.9, pp. 671-678, September 1966.

[3] A.A.B. Pritsker, *The GASP IV Simulation Language*. New York: Wiley, 1974.

[4] P.J. Kiviat, R. Villanueva, and H.M. Markowitz, *SIMSCRIPT II.5 Programming Language*. Los Angeles, CA: C.A.C.I., 1973.

[5] G. Gordon, *The Application of GPSS V to Discrete Systems Simulation*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[6] A.A.B. Pritsker, *Modeling and Analysis Using Q-GERT Networks*. New York: Wiley, 1977.

[7] C.H. Sauer, E.A. MacNair, and J.F. Kurose, "The Research Queueing Package Version 2: Introduction and Examples," RA 138, IBM T.J. Watson Research Center, Yorktown Heights, NY, April 1982.

[8] W. Whitt, "The Queueing Network Analyzer," *BSTJ*, Vol.62, no.9, Part 1, pp. 2779-2815, November 1983.

[9] M. Veran and D. Potier, "QNAP2: A Portable Environment for Queueing Systems Modelling," *Proc. International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, May 1984.

[10] A. Brandwajn, "Issues in Mainframe System Modelling - Lessons from Model Development at Amdahl," *Proc. International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, May 1984.

[11] M. Booyens, P.S. Kritzinger, A. Krzesinski, P. Teunissen, and S. van Wyk, "SNAP: An Analytic Multiclass Queueing Network Analyser," *Proc. International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, May 1984.

[12] S.C. Bruell, G. Balbo, S. Ghanta, and P.V. Afshari, "A Mean Value Analysis Based Package for the Solution of Product-Form Queueing Network Models," *Proc. International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, May 1984.

[13] K.G. Ramakrishnan and D. Mitra, "An Overview of PANACEA, a Software Package for Analyzing Markovian Queueing Networks," *BSTJ*, Vol.61, no.10, Part 1, pp. 2849-2872, December 1982.

[14] I. Kino and S. Morita, "PERFORMS - A Support System for Computer System Performance Evaluation," *Proc. International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, May 1984.

[15] J. Buzen, "BEST/1 - Design of a Tool for Computer System Capacity Planning," *Proc. 1978 AFIPS National Computer Conference*, Vol.47, pp. 447-455, 1978.

[16] H. Beilner and J. Maeter, "COPE: Past, Presence and Future," *Proc. International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, May 1984.

[17] "PAWS - Performance Analyst's Workbench System: INTRODUCTION AND TECHNICAL SUMMARY," Information Research Associates, Austin, TX, July 1983.

[18] T. Nishida, M. Murata, H. Miyahara, and K. Takashima, "PLANS: Modelling and Simulation System for LAN," *Proc. International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, May 1984.

[19] B. Mueller, "NUMAS: A Tool for the Numerical Modelling of Computer Systems," *Proc. International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, May 1984.

[20] B. Melamed and R.J.T. Morris, "Visual Simulation: The Performance Analysis Workstation," *Computer*, Vol.18, no.8, pp. 87-94, August 1985.

[21] A.K. Agrawala, S.K. Tripathi, M. Abrams, K.K. Ramakrishnan, M. Singhal, and S.H. Son, "STEP-1: A User Friendly Performance Analysis Tool," *Proc. International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, May 1984.

[22] J.C. Browne, D. Neuse, J. Dutton, and K.-C. Yu, "Graphical Programming for Simulation of Computer Systems," *Proc. 18th Annual Simulation Symposium*, pp. 109-126, March 1985.

[23] A. Blum, E.A. MacNair, and C.H. Sauer, "The Research Queueing Package: Graphics Developments," RC 9513, IBM T.J. Watson Research Center, Yorktown Heights, NY, August 1982.

[24] R.G. Covington, "CSIM: An Efficient Implementation of a Discrete Event Simulator," M.S. Thesis, Department of Electrical and Computer Engineering, Rice University, Houston, TX 77251-1892, April 1985.

[25] K.A. Doshi, "Extended Queueing Network Modeling," M.S. Thesis, Department of Electrical and Computer Engineering, Rice University, Houston, TX 77251-1892, April 1985.

[26] S. Madala, "Design of a Graphical Input Simulation Tool for Extended Queueing Network Models," M.S. Thesis, Department of Electrical and Computer Engineering, Rice University, Houston, TX 77251-1892, April 1985.

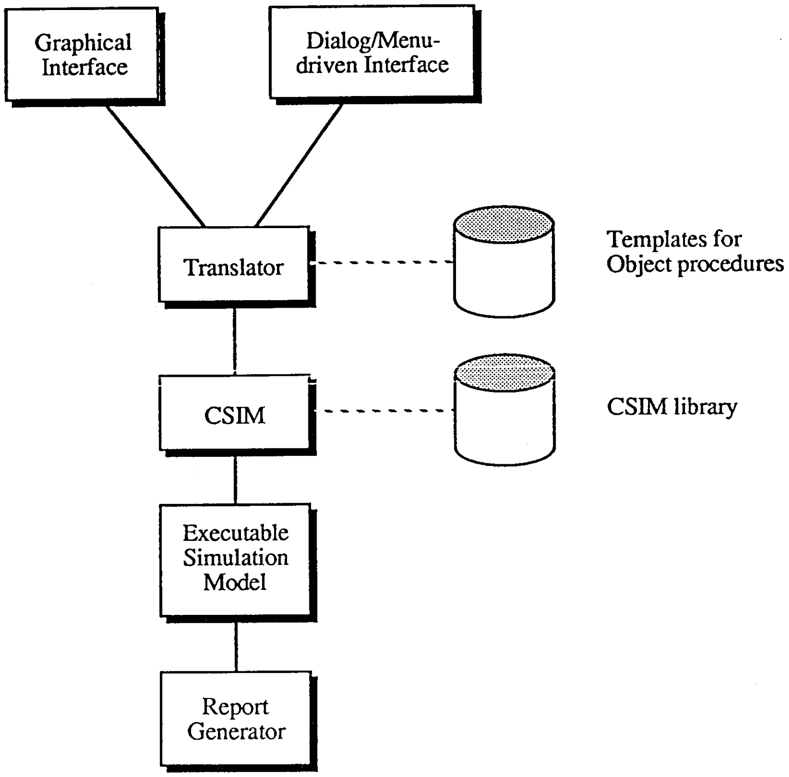

Graphical Interface
Dialog/Menu-driven Interface
Translator
Templates for Object procedures
CSIM
CSIM library
Executable Simulation Model
Report Generator

Figure 1 Organization and components of GIST

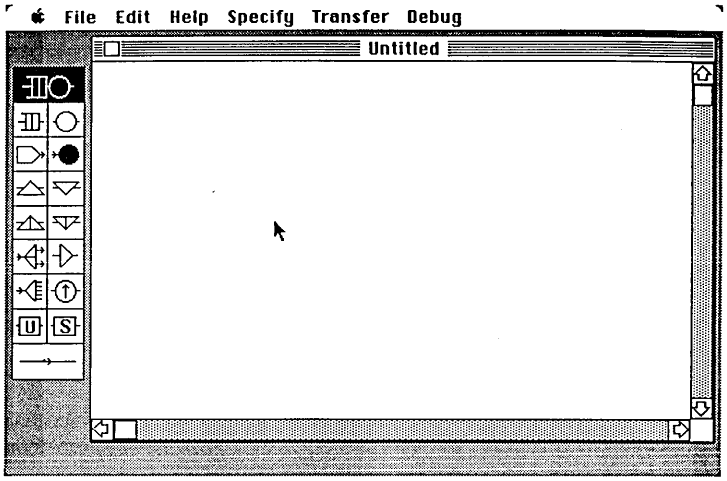


Figure 2 GUIDE windows

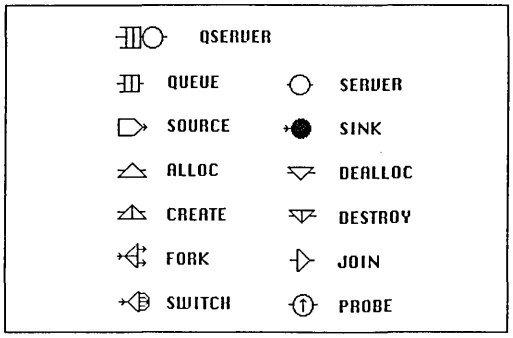


Figure 3 Object type icons

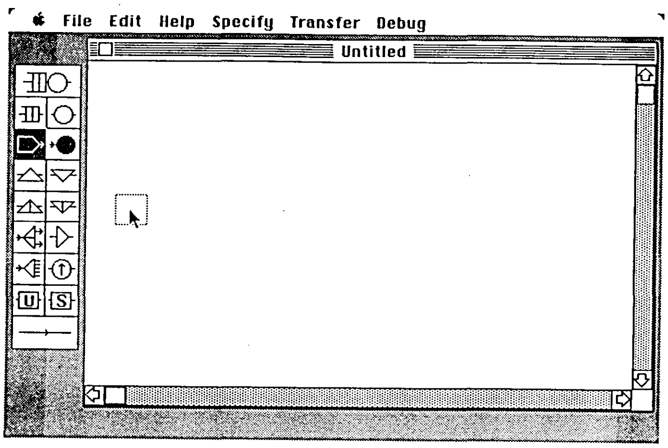


Figure 4 Dragging an icon into the working window

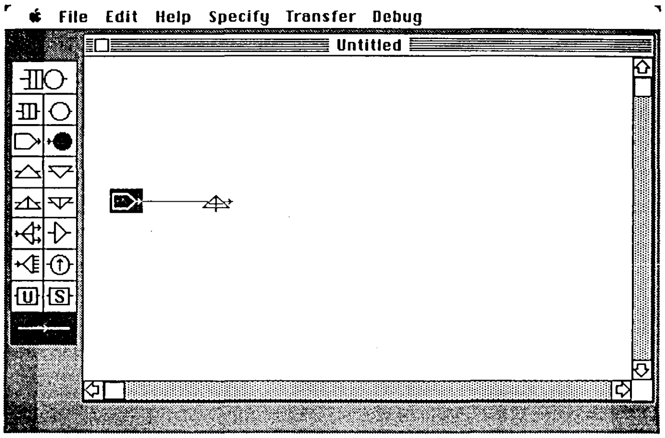


Figure 5 Specification of interconnections

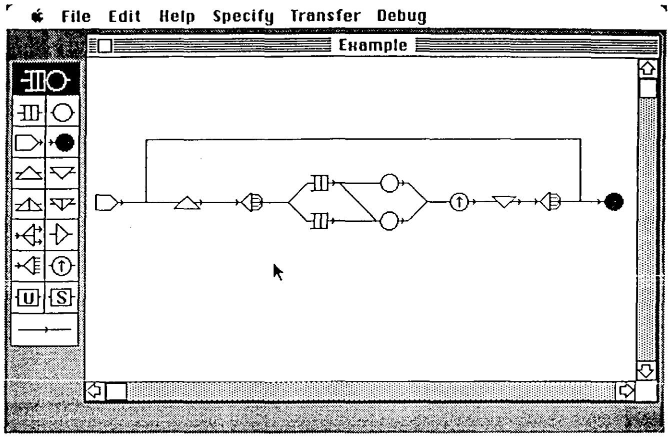
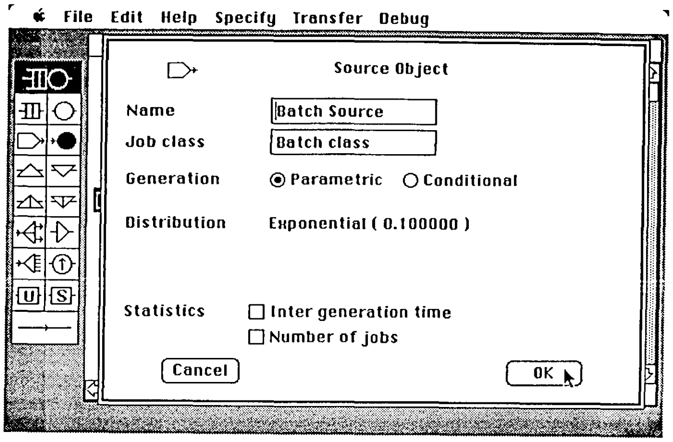


Figure 6 Complete Example



Source Object

Name: Batch Source

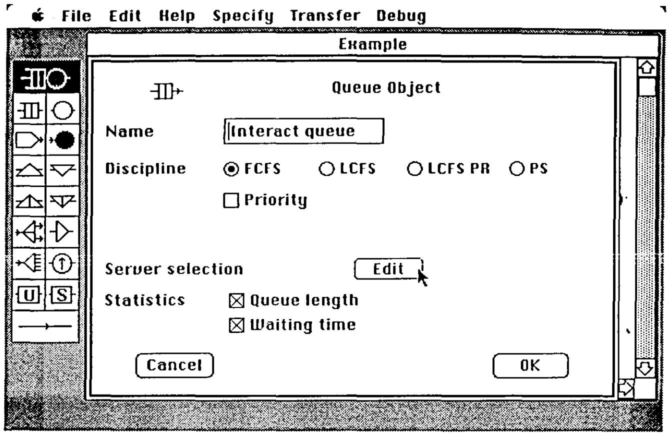
Job class: Batch class

Generation: ● Parametric ○ Conditional

Distribution: Exponential (0.100000)

Statistics: □ Inter generation time
□ Number of jobs

Figure 7 Source specification dialog window



Queue Object

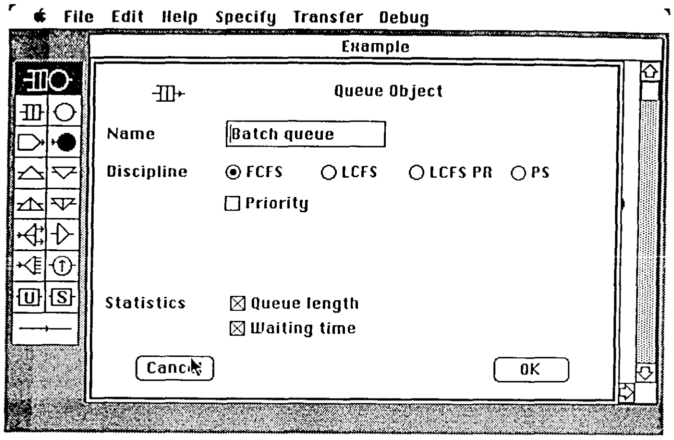
Name: Interact queue

Discipline: ● FCFS ○ LCFS ○ LCFS PR ○ PS
□ Priority

Server selection: Edit

Statistics: ☒ Queue length
☒ Waiting time

Figure 8 "Interact Queue" specification window



Queue Object

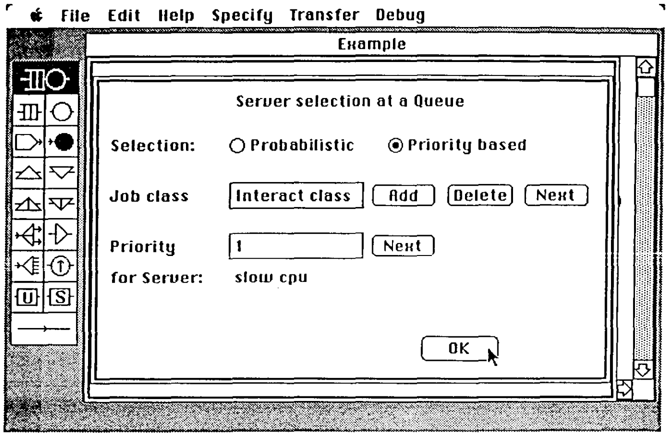
Name: Batch queue

Discipline: ● FCFS ○ LCFS ○ LCFS PR ○ PS

□ Priority

Statistics: ☒ Queue length ☒ Waiting time

Figure 9 "Batch queue" specification window



Server selection at a Queue

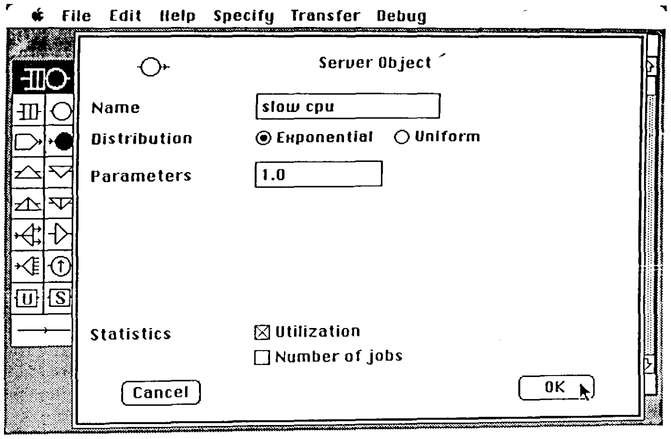
Selection: ○ Probabilistic ● Priority based

Job class: Interact class

Priority: 1

for Server: slow cpu

Figure 10 Server selection at "Interact queue"



Server Object

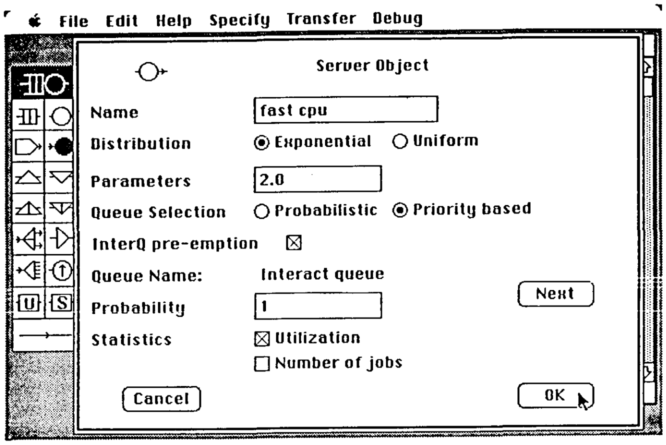
Name: slow cpu

Distribution: ● Exponential ○ Uniform

Parameters: 1.0

Statistics: ☒ Utilization □ Number of jobs

Figure 11 "Slow cpu" specification window



Server Object

Name: fast cpu

Distribution: ◉ Exponential ○ Uniform

Parameters: 2.0

Queue Selection: ○ Probabilistic ◉ Priority based

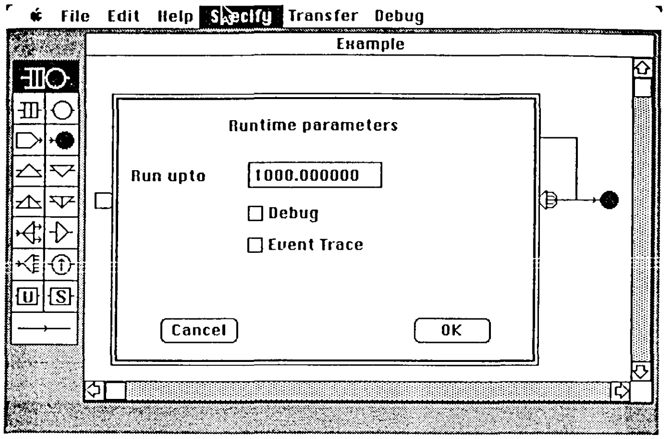
InterQ pre-emption ☒

Queue Name: Interact queue

Probability: 1

Statistics: ☒ Utilization
□ Number of jobs

Figure 12 "Fast cpu" specification window



Runtime parameters

Run upto: 1000.000000

□ Debug

□ Event Trace

Figure 13 Runtime parameters specification

A Graphics-Oriented Modeler's Workstation Environment for the RESearch Queueing Package (RESQ) [1]

James F. Kurose
Kurtiss J. Gordon

Department of Computer and Information Science
University of Massachusetts
Amherst, Mass. 01003

Robert F. Gordon
Edward A. MacNair
Peter D. Welch

Computer Science Department
IBM Hawthorne Research Laboratory
Yorktown Heights, NY 10598

ABSTRACT

The development of computer programs for evaluating the performance of resource contention systems such as computer, communication, and manufacturing systems, represents a significant software development effort. This effort can be very costly in terms of the work required to evaluate the results of these programs and to build, verify, modify, and maintain the programs themselves. In the paper we discuss our current efforts on design, development and implementation of an integrated, interactive, high-level, graphics-oriented environment for constructing and evaluating extended queueing network performance models of resource contention systems. The RESQ modeling language provides the foundation for the capabilities provided by the environment.

1. Introduction

Evaluating the performance of resource contention systems such as computer, communication, and manufacturing systems is a critical and complex step in the design, planning and evaluation process for such systems. Fortunately, the difficulty and complexity of this step can be considerably reduced by providing the performance analyst with an appropriate set of software tools to aid in the performance evaluation process.

Traditionally, the notion of an "appropriate" software tool has meant a declarative language in which a performance model may be specified and then evaluated (either numerically or through simulation). General purpose programming languages such as PASCAL or FORTRAN and general purpose simulation languages such as GPSS, SIMULA and SIMSCRIPT have often been used in the performance evaluation process. More recently, special-purpose modeling languages such as RESQ [Saue82] [Saue84], PAWS [IRA81] and STEP-1 [Trip84] have been developed. The languages in this latter group provide the modeler with a rich set of programming constructs or "modeling elements" which can be easily used to specify the objects and actions actually found in the modeling domain. At this level of abstraction, performance models can be easily and rapidly constructed, evaluated, and verified; the analyst's productivity and effectiveness thus increase dramatically.

With the recent widespread availability of relatively inexpensive, graphics-oriented workstations, the development of a *performance evaluation workstation environment* has become possible. At the heart of such an environment is the notion of a *graphical* representation of a performance model [Mela85]. We believe a graphical representation is the most natural form in which to express a performance model; indeed, there has always been a very strong pictorial flavor even in textual modeling languages such as RESQ and PAWS [Saue84]. Perhaps more importantly, the graphical representation of a model creates the opportunity to provide a single, uniform interface for *all* aspects of the modeling and performance evaluation process, and the environment can thus provide the analyst with an integrated set of software tools for use throughout the modeling lifecycle. A model may be graphically constructed, edited, and documented. In the case of simulation the solution may be animated. Performance results can be displayed, manipulated and examined together with the performance model itself in a single and integrated manner. We believe that the ability to easily create such a natural expression of performance models together with the ability to easily access all the modeler's software tools through a uniform and integrated interface will result in yet further increases in an analyst's capa-

[1] This work was partially supported by a grant from the International Business Machines Corporation.

CH2345-7/86/0000/0719$01.00 © 1986 IEEE

bilities and productivity.

In this paper, we discuss our current efforts in developing an integrated, graphics-oriented performance evaluation workstation environment for constructing, maintaining, revising, and evaluating performance models of resource contention systems. The RESQ modeling language provides the foundation for the capabilities provided by the environment. This language provides the modeler with a rich set of high-level modeling primitives with which a performance model may be constructed, multiple methods (both analysis and simulation) for solving these models, and sophisticated statistical techniques for evaluating performance results.

We believe our RESQ modeler's workstation environment offers several important advantages over other related efforts; indeed, we adopted what we have found to be the most promising aspects of this previous work and have drawn on the modeling experience and methodology of RESQ users (including ourselves) in designing the environment. We believe that the use of a *single* graphical description of the performance model throughout *all* phases of the modeling process is central to the development of a truly *integrated* modeler's workstation environment. We contrast this with the approach taken in SIMVISION and CINEMA [Heal85], which provide graphical animation of *textual* specifications of SIMSCRIPT II.5 [Russ85] and SIMAN [Pegd85] performance models, respectively. We also note that, since the RESQ language provides an extremely rich set of modeling primitives, our environment offers greater modeling power and flexibility than are available in other environments such as [Mela85] [Sinc85].

In the following section we provide an overview of the RESQ modeling language. In section 3 we describe the hardware and software configuration of the performance modeler's workstation environment. In section 4, we discuss the process of constructing and revising the model, solving the model, and displaying the output performance results. A sample modeling session is used to illustrate these ideas. Section 5 summarizes this paper and discusses directions for future work.

2. The Research Queueing Package (RESQ)

In this section we provide a brief overview of the RESQ language, since it forms the foundation upon which our performance modeler's workstation environment is built and demonstrates the wide range of modeling capabilities that must be supported by the environment. For a more detailed discussion of RESQ, the reader is referred to [Saue82] [Saue84].

The RESQ modeling language provides the analyst with a rich set of high-level modeling primitives with which performance models of resource contention systems (such as computer, communication, and manufacturing systems) may be *specified* and *evaluated*. At the highest level, a RESQ model consists of

- a population of jobs,
- a set of queues,
- a set of nodes,
- rules specifying how the jobs circulate among the nodes and queues,
- additional modeling constructs, and
- information concerning the solution of the model.

Queues typically represent resources in the system being modeled; jobs represent the objects in this system which require the use of these resources. The purpose of the model is typically to analyze this contention for resources and its effect on the flow of jobs through the system.

RESQ provides two type of queues: *active* queues and *passive* queues. Active queues have servers which provide service to a job. An active queue might be used to represent, for example, a computer's central processing unit, a communication link in a network, or an assembly station in a flexible manufacturing system. A job arrives at an active queue, waits until it is selected for service (according to the user-selected service discipline), receives service, and eventually leaves the queue. Passive queues allow for a more general notion of resource use and provide a natural mechanism for modeling phenomena such as simultaneous resource use and blocking. A passive queue consists of a pool of tokens, where each token typically represents a distinct unit of some resource (e.g. a buffer or a block of memory). A passive queue additionally consists of a set of nodes at which the passage of a job may cause a user-specified number of tokens to be allocated, deallocated, created or destroyed. A job may hold tokens from a passive queue as it visits other nodes or (active or passive) queues in the model.

Additional nodes are provided by RESQ in order to model phenomena other than strict resource contention. For example, if there are external arrivals or departures to/from the system being modeled, *source* and *sink* nodes (respectively) can be used to model these effects. *Set* nodes permit values to be assigned to variables associated with the simulation. *Split* nodes allow a job to create an independent copy of itself. *Wait* nodes cause a job to be delayed until a specified Boolean condition becomes true. These and other additional auxiliary nodes are described in [Saue82].

A *chain* specifies the route or path that a class of jobs will follow through the queues and nodes in a model. These routing decisions may be fixed, probabilistic, or dependent on the current state of the simulation (e.g., dependent on the number of jobs at a given queue). In an *open* chain, new jobs may be created at source nodes and jobs may leave the chain at a sink node. In a *closed* chain, an essentially fixed number of jobs circulates among the nodes in the chain.

RESQ also provides numerous modeling constructs in addition to those discussed above, such as symbolic constants and parameters, and arrays of nodes. Another important modeling construct is the *submodel*, which may be used to define a parameterized template of an interconnected "subnetwork" of nodes and queues. This submodel may then be *invoked* several times (much the same way that a macro may be invoked several times in a programming language) to create multiple instances of that subnetwork. We note that submodels provide an important mechanism for constructing modular, hierarchically-structured models of large and complicated systems.

The final component of a RESQ model concerns the solution itself. RESQ can both simulate and, in some cases, analytically solve for the various performance measures requested by the performance analyst. In the case of simulation, the analyst may specify information regarding the desired length of the simulation run, a method for generating confidence intervals, tracing parameters, and other simulation-related information. Resource utilization, throughput, average queue length, queueing time and queue length distribution are typically requested performance measures. Several methods are provided by RESQ for generating confidence intervals, including "independent replications", the "spectral" method, and the "regenerative" method [Saue82].

3. An Overview of the Modeler's Workstation Environment Configuration

In this section we provide an overview of the physical and logical structure of the performance modeler's workstation environment. A detailed description of the functioning of each of the major software components together with a sample modeling session is described in the following section.

The hardware and software configuration of the modeling environment is shown below in figure 1. The workstation currently in use is a personal computer connected to the mainframe with an all points addressable graphics display and a character display. The graphics screen is considered the primary display device, and a pictorial representation of the network model is visible to the user at all stages of model construction, evaluation, and output analysis. The analyst controls the modeling process by selecting menu items and pointing to objects on this display. The graphics display is programmed using the Virtual Device Interface (VDI) provided by the Graphics Development Toolkit.

The character screen is used for:

- entering the attributes of the elements in the RESQ model (e.g the service discipline of a queue, the number of jobs in a closed chain),
- specifying the parameters for model runs (e.g., service times, interarrival times),
- choosing the multiple performance measures to plot and specifying the non-default viewing attributes of output graphs (e.g., color, axis intervals), and
- displaying tutorial-like help.

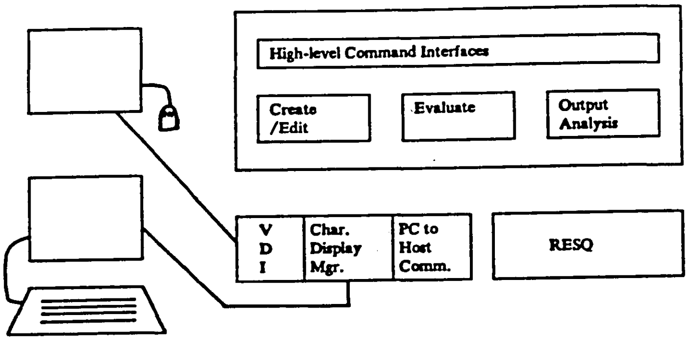


Figure 1: Input/Output Hardware and Software Configuration

The textual information is entered via the keyboard into dynamic forms on the character display.

Our decision to employ two screens was influenced by the work of [Gilb85] and motivated by our experience that complicated, real-world RESQ models may require a significant amount of textual attribute information to be specified. Since we desired to display a significant amount of graphical information at all times (to display the appropriate context for a given modeling element) as well as use the graphics display to cue the attribute specification process, a two-screen mode of interaction was a natural selection.

As shown in figure 1, the software divides broadly into three modules (model creation and editing, model evaluation, and output analysis); the functionality provided by each of these will be described in the following section. For now we note that uniform access is provided to each of these modules by a high level graphical command dispatcher. We also note that each of these modules directly relies upon portions of previously developed components of RESQ [Saue82].

The PC-to-host connection provides the best of two worlds; we may simultaneously exploit the processing power of the mainframe for execution of large RESQ models and the interactive graphics capability of the PC for constructing the model and viewing the graphical results. This allows the user through a consistent interface on the PC to iteratively create the model, view the simulation results, revise the model based on the output, and view comparative results for families of models. The user is presented with one workstation interface, making the PC-to-host connection transparent to the user.

4. System Modeling Using the Performance Analysis Workstation Environment

4.1 Introduction

In this section we describe the functionality provided by the RESQ performance modeler's workstation environment. Using a simple model of a time-shared computer system as an example, we trace the analyst's interaction with the environment from model construction to solution, to the display and manipulation of the performance results.

Let us suppose that the computer system to be modeled consists of a fixed number of terminals at which users enter jobs for execution. The computer itself consists of a single CPU, memory, a fixed disk, and a floppy disk. The purpose of the model is to analyze the contention for the memory, CPU, and I/O resources. The system operates as follows. Once a job is submitted to the system, it must first be allocated a single memory partition from the fixed number of partitions which comprise the computer's active memory. The "execution" of a job then consists of some random numbcr of rcpetitions of a CPU burst (of random length) followed by an I/O operation (also of random length). A job thus cycles some random number of times between the CPU and the I/O devices before terminating execution and returning results to a terminal. Once a job terminates, the user then waits some random amount of time before submitting the next job for execution.

Our RESQ model of this system — which we shall call CSMTM, for "central- server model with terminals and memory" — will be a variation on the well-known central server model [Saue81] for timesharing systems. The I/O devices will be represented by first-come first-served active queues with exponentially distributed services times; the CPU by an active queue with a processor-sharing service discipline. A special type of active queue known as an infinite server queue will be used to model the terminals, and the service time at this queue will be an exponentially distributed user "think time". A passive queue which contains a fixed number of tokens will be used to model memory contention, and each token will correspond to a memory partition. Finally, a fixed number of jobs (equal to the number of users in the system) will circulate among these various queues. A job waiting at the infinite server queue models the thinking user seated before a terminal; a job circulating among the CPU and I/O devices may be thought of as "executing".

4.2 Layout of the Graphics and Character Screens

Upon entering the workstation environment, the analyst is presented with the main command menu in the lower left of the graphics screen and a screen-management menu along its right border (see Figure 2). The main menu permits selection among the three subsystems of the environment: create/edit, evaluate, and output analysis. It also contains items to display help screens and to permit the redefinition of global default options. Based upon the user's selection, additional menus will pop up to display the more detailed suboptions. In particular, selection of one of the three subsystems first brings up a submenu containing a list of models currently available in the user's library or database, and the option "Other" for a textual specification of a model name.

The screen-management menu, which is visible throughout all three subsystems, has options for zooming in or out and panning the view of the model, for locating a particular model element by name, for traversing levels of a hierarchically structured model, and for toggling the visibility of the other menus. With the other menus visible, the modeling area is restricted to the upper 3/4 of the display, and a single-line prompt describing the action expected from the user appears just above the menus. When they are turned off, the modeling area expands to fill the entire screen.

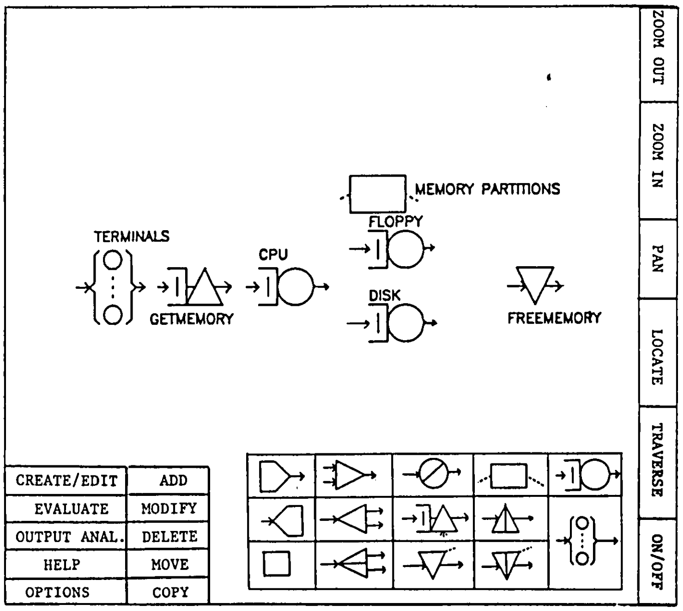
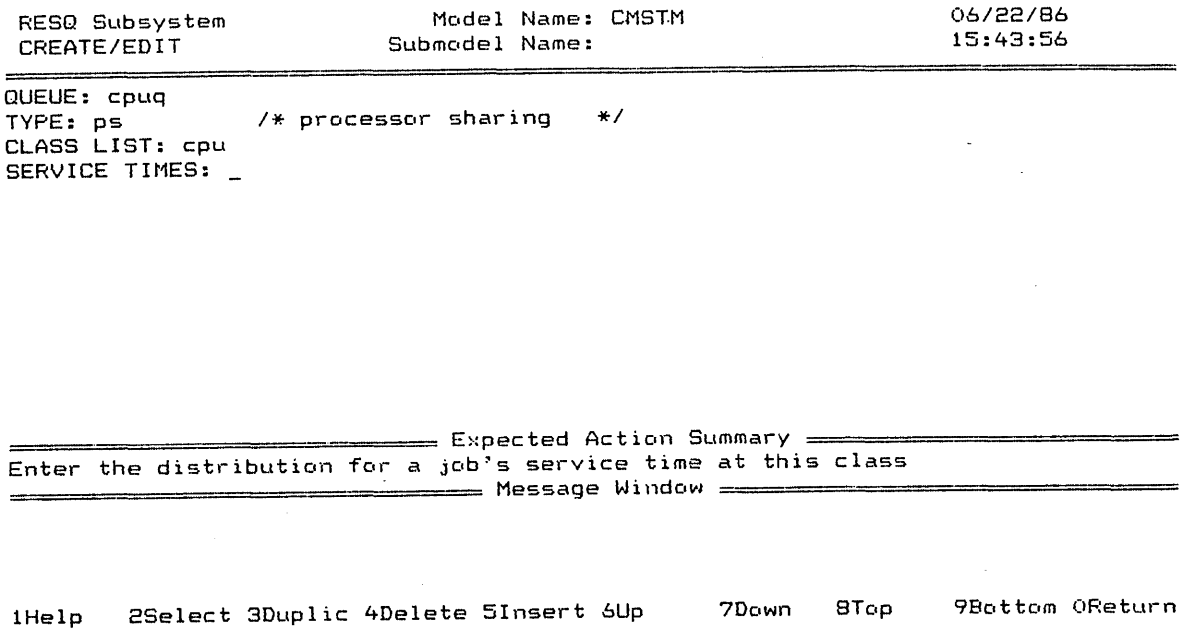


Figure 2: Graphics screen showing a partially completed RESQ model

The display on the character screen is divided horizontally into four sections (see Figure 3). The top section contains the name of which subsection of the environment is active, and the name and version (date and time) of the model which is being worked on. The second section is used for the dynamic forms controlling the entry and manipulation of textual information. The third section repeats the expected action prompt, also visible on the graphics screen. The final section provides space for messages from the environment to the user and summarizes the current meanings of the function (PF) keys. Initially, all of these sections except for the expected action window are blank.

4.3 Graphical Model Construction and Specification

When the analyst selects the model create/edit command, "Create/Edit" appears in the top section of the character screen and the environment prompts for the name of the model. Once this has been specified, the graphical representation of the model is displayed in the modeling area, and the menu of create/edit commands pops up in the lower portion of the graphics screen; any existing model element attribute specifications are also parsed at this point. If a new model is being defined (as we will assume in this example), the graphical display will be blank and no attribute definitions will be available to be parsed.



```
RESQ Subsystem              Model Name: CMSTM                  06/22/86
CREATE/EDIT              Submodel Name:                        15:43:56
========================================================================
QUEUE: cpuq
TYPE: ps          /* processor sharing    */
CLASS LIST: cpu
SERVICE TIMES: _



=========================== Expected Action Summary ====================
Enter the distribution for a job's service time at this class
================================ Message Window ========================


1Help   2Select 3Duplic 4Delete 5Insert 6Up     7Down   8Top    9Bottom 0Return
```

Figure 3: Character screen showing a partially completed form for a modeling element

The first time the modeler selects the "add" option from the create/edit menu, a palette of the available RESQ icons pops up. The graphical RESQ model is constructed by picking icons from the palette using a graphics pointing device (typically a mouse or a joystick) and placing these icons in the modeling area. Each icon consists of a stylized drawing of the RESQ modeling element it represents within a rectangular color-filled background; an icon's fill color is used to cue the modeler on the "state" of that modeling element. When a palette icon is picked, it is highlighted in red and connected (via a rubberband) to the graphics cursor until the icon is placed in the modeling area. Once in the modeling area, the icon's fill color is set to yellow to indicate that its attributes have not yet been specified. Figure 2 shows the icon palette, the main and "Create/Edit" command menus, and the seven model elements in our central server model. (Due to the reproduction process for this paper we are unable to show the fill colored background and thus have simply shown the stylized drawing of the elements).

Once an icon has been placed in the modeling area, that instance of the icon is considered an element of the model and its textual attributes can be defined. These attributes are specified in a context-sensitive form on the character display. The overall attributes of the model itself (specifically, the model parameters, numerical constants and maximum size information) are typically specified as a first step in the model construction process. Attributes for the other modeling objects may be entered or modified at any point in the model construction process. Figure 3 shows a partially completed attribute form in which the queue name, service discipline, and class name have been specified.

General information about the current model is displayed above the form. The form itself closely resembles the "dialogue file" definition of an element in the text-only version of RESQ. Each line in the form contains a "prompt" and a field in which the "reply" to the prompt is to be specified by the analyst. As the analyst moves the text cursor from one of these reply fields to another, a message concerning the information to be entered in the current reply field is dynamically displayed below the form in the Expected Action Window. Certain fields (such as the service discipline for a queue) may contain one of only a fixed number of possible replies. In such cases, the analyst simply scrolls through the various possible replies until the desired reply is displayed. Finally, we note that these forms must be *context sensitive* in the sense that a reply to one prompt may change the remainder of the form. For example, if the analyst had selected (through scrolling) a priority service discipline for the disk queue, prompts for the priority information would also have dynamically appeared in the form. The forms mechanism also supports the repetition or deletion of prompts (or sets of prompts) and the extension of a reply field over several lines.

Once the analyst completes an attribute form, the form is immediately parsed for syntactic and semantic correctness. A correctly completed form causes the background color of the icon to change from yellow to green. An incorrect form causes error messages to be displayed below the form and turns the background color to red. The analyst may correct the form either immediately or at some later point. The completed set of attribute forms for the central server model is given in Appendix A.

After the analyst has correctly specified the attributes of a set

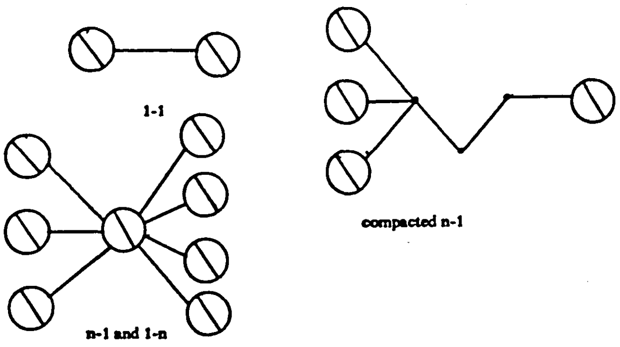


Figure 4: Line Drawing Types

of graphical elements in the modeling area, it is time to specify the routing of jobs which defines how they should flow "from" one of these elements (nodes) "to" other nodes in the model. The environment provides extensive support for this routing definition process. The routing between nodes is accomplished using the pointing device. As the analyst graphically defines the routing with the pointing device, a textual description of the routing also appears in the attribute form for the routing chain currently being defined.

In order to define the routing between nodes, the user first graphically picks a group consisting of one or more "from" nodes. A group of "to" nodes is then selected, and the environment connects the "from" nodes and "to" nodes using straight lines (except as noted below). If another set of "to" nodes is immediately selected, the previous "to" nodes are implicitly used as the set of "from" nodes; this considerably reduces the amount of work required in the routing specification.

Examples of the different types of line drawing supported by the environment are shown in figure 4. As discussed above, straight lines are automatically drawn by the environment. However, a user may alter a straight line by picking a point along the line, inserting a vertex, and then moving the vertex to another point in the modeling area; this permits lines such as the one immediately below the disk queue in figure 5 to be drawn. We note that the existence of a vertex in a line has no syntactic or semantic meaning in the RESQ model itself. Finally, a "compacted" line may be used to define a 1-N or N-1 routing, as shown in figure 4.

The above discussion has focused on the initial construction of a model. In addition to the capabilities (adding elements, panning and zooming) described above, a modeler may also delete, move, copy, modify, annotate, print and plot any portions of the graphical model description. Finally, we note that the environment provides direct graphical support for the use of submodels. Each submodel is defined on a different graphical "plane", and the *traverse* command is used to move the modeler from one plane to another. We believe this approach greatly encourages a top-down, hierarchical, and structured approach towards model development.

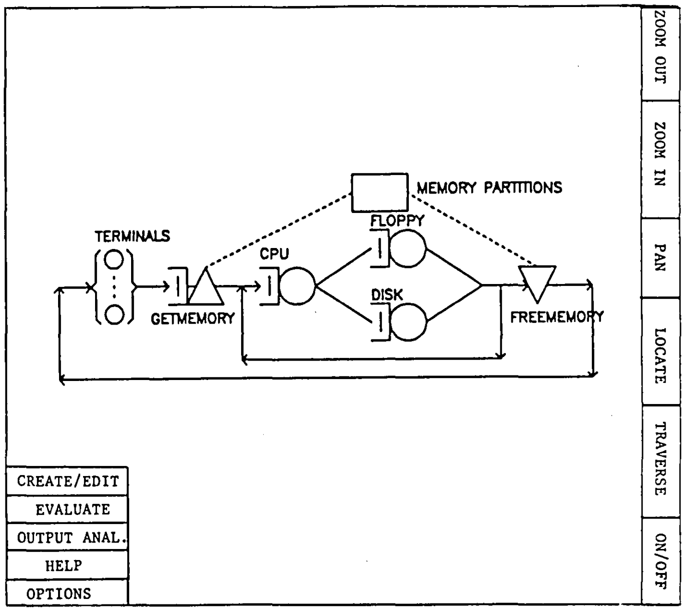


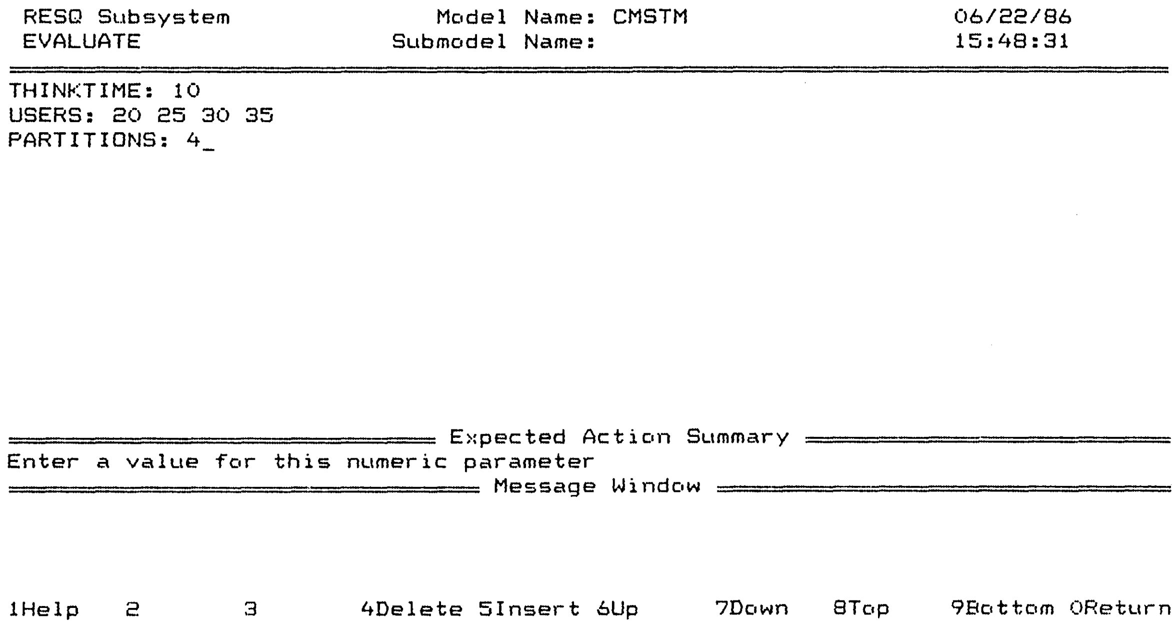
Figure 5: Graphical Display after Model Construction

Once a model's elements, the routing, and the simulation-dependent information (if any) have been specified, the model can now be solved.

4.4 Model Solution

The Model Solution component of the workstation software allows the analyst to specify run parameters (such as think time, number of users, and number of partitions in our CSMTM model example) and then to evaluate the model and produce the output statistics for analysis. In our workstation architecture, the parameter specifications and run commands are uploaded to the host where the model is executed. While the RESQ model is executing on the host, the analyst can continue to work on the PC to construct and modify models and to evaluate output statistics from previous runs. The analyst can check the progress of the host batch run and, when the model's run is completed, can directly view its results on the PC. The upload to the host of commands and parameters and the subsequent download to the PC and accompanying EBCDIC to ASCII translation of RESQ results are handled by the model solution software, transparently to the user. The analyst selects "Evaluate" on the main menu and need not be aware that there are host/PC communications and translations being done automatically for him.

When the analyst wants to evaluate his model, he selects "Evaluate" from the main menu on the graphics screen (see figure 5) and chooses the name of the model to execute. The template on the character screen then displays the parameters to be specified for that model. The analyst provides the values of the parameters for each run desired (see figure 6), and the model is then solved on the host. In the CSMTM model example, parameter values are input for four runs of the model with the number of users varying



```
RESQ Subsystem                 Model Name: CMSTM                         06/22/86
EVALUATE                    Submodel Name:                               15:48:31
=====================================================================================
THINKTIME: 10
USERS: 20 25 30 35
PARTITIONS: 4_









============================== Expected Action Summary ===============================
Enter a value for this numeric parameter
=================================== Message Window ===================================



1Help   2      3       4Delete 5Insert 6Up     7Down   8Top    9Bottom 0Return
```

Figure 6: Parameter Specification for Evaluate on the Character Screen

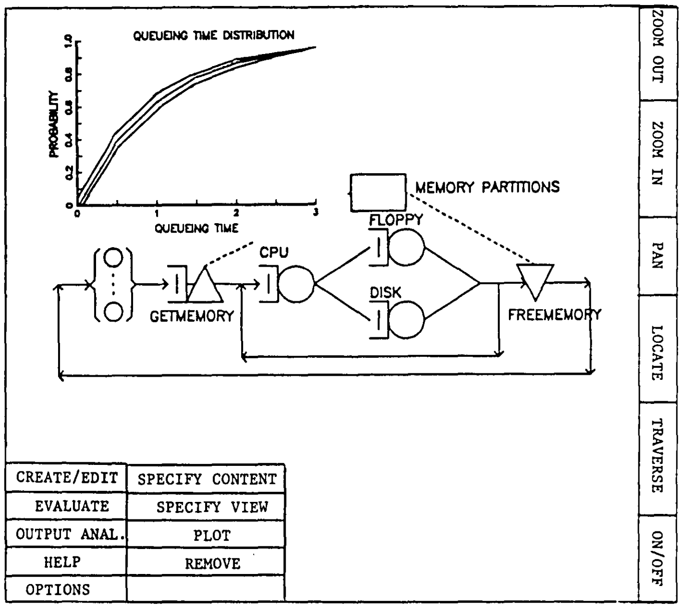


Figure 7: Graphics Display During Output Analysis

from 20 to 35, with think time fixed at 10 and partitions fixed at 4.

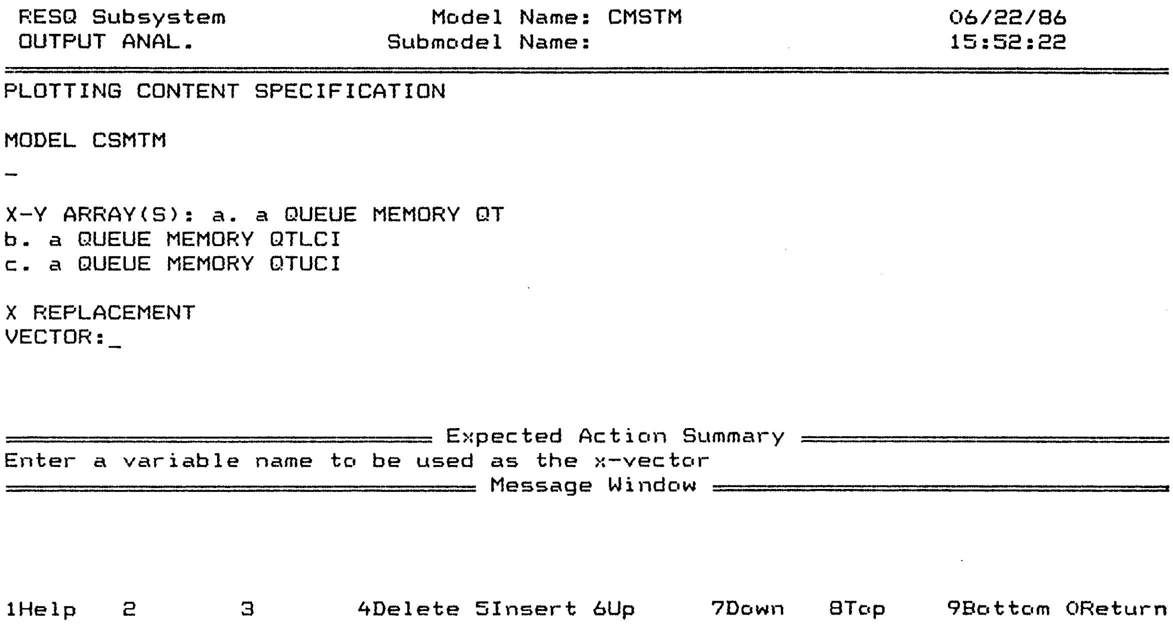
When the evaluation run has been completed, the output statistics are downloaded to the PC, and the performance measures are stored for output analysis.

4.5 Output Analysis

The Output Analysis component of the workstation software provides the analyst with the capability of viewing graphically all the performance measures of the simulation. The analyst selects the contents and form of the graphs to suit his purposes, such as for model verification, determination of transient phase, comparison of runs, or investigation of alternatives. Decisions based on this analysis can then be immediately translated into direct modification of the model diagram, re-running of the model, and the resulting production of the next version of performance measures. This cycle of model construction, execution, and output analysis is performed by the user without switching modes; all actions are accomplished from the one workstation environment.

Selecting "Output Analysis" from the main menu creates a submenu for providing plotting specifications. This submenu (see figure 7) provides the analyst with the ability to specify the content, location, and form of any graph, to plot the graph, and to remove it from the display screen. The latest content, location, and form are remembered by the system, so that the analyst can produce graphs without further specifications or, if desired, can change any specifications to tailor the next chart to his needs.

The following illustrates the use of the Output Analysis submenu to produce graphs of performance measures. For example, marking a node or group of nodes by pointing to them on the model diagram and then selecting the menu item "Specify Content" would provide the analyst with a list of output variable names associated with the marked area of the model. Each output variable name represents a vector of x-y value pairs that can



```
RESQ Subsystem              Model Name: CMSTM                   06/22/86
OUTPUT ANAL.             Submodel Name:                         15:52:22
========================================================================
PLOTTING CONTENT SPECIFICATION

MODEL CSMTM
_

X-Y ARRAY(S): a. a QUEUE MEMORY QT
b. a QUEUE MEMORY QTLCI
c. a QUEUE MEMORY QTUCI

X REPLACEMENT
VECTOR:_



========================= Expected Action Summary ======================
Enter a variable name to be used as the x-vector
============================ Message Window ============================


1Help   2      3      4Delete 5Insert 6Up     7Down   8Top    9Bottom 0Return
```

Figure 8: The Plot Content Specification Form

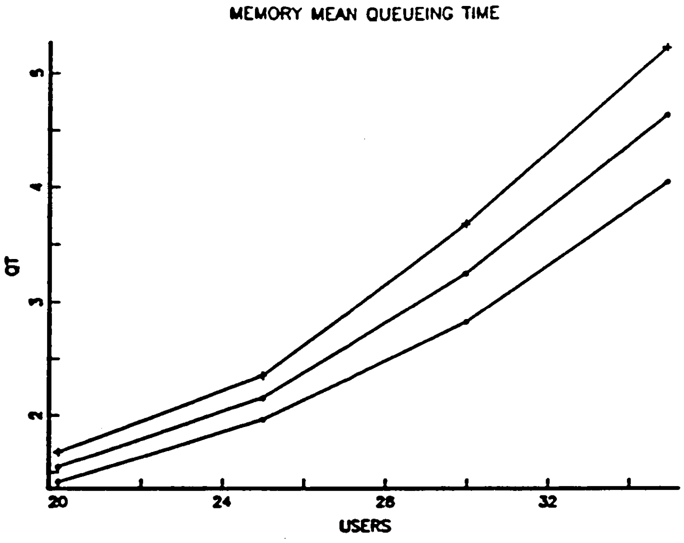


Figure 9: Plot of Mean Queueing Time

be plotted. Selecting one of the variable names and pressing the menu item "Plot" will immediately produce the plot with the latest default values for window position, size, x- and y-axis intervals, etc. In particular, pointing to the memory node in the CSMTM model, selecting the variable name "Queueing Time Distribution", and pressing "Plot" will produce the graph in figure 7, showing the queueing time distribution with its confidence interval for the memory queue. The analyst can also choose to display several variables on the same graph, or to substitute the y values of variable v' for the x values of variable v, or to fit a curve, or to request a projection. These specifications are made by filling in the template for Plotting Content Specifications on the character screen. Figure 8 shows the template to plot the mean queueing time of the memory queue in the CSMTM model with its confidence interval. Completing the template with the variable name for the number of users in each run and selecting "Plot" would produce the graph shown in figure 9.

Independently of content specification, the analyst can specify the form of the output graph by choosing "Specify View" in the Output Analysis submenu. He can choose the graph type, such as line graph, bar chart, histogram. The analyst can place the graph at any point on the modeling field. The analyst points to the desired spot on the graphics screen for one corner of the output window. A rubberbanding box allows him to view the resulting window location as he tacks down the opposite corner. Selecting the "Plot" menu item will produce the graph in this window using the remaining default values for its form. The analyst can use the Plotting Attribute Specifications template on the character screen to change the form attributes, such as the x- and y-axis tick marks, the axis labeling, graph colors. Selecting "Plot" will produce a graph of the latest specified performance measures in the form just specified. Selecting "Remove" will remove the graph and redraw the underlying portion of the model.

5. Summary and Future Directions

We have discussed an integrated graphics-oriented workstation environment for the specification, evaluation, and output analysis of network models. The environment provides a uniform interface for all aspects of the performance evaluation process. The model is constructed and modified by creating and editing the model diagrams on a graphics display and by entering textual information on a character display. Model specification and modification can be performed in various orders to accommodate the working styles of different analysts. In the evaluation phase, the user assigns values for any parameters defined in the model for one or more solutions, and the model is sent to the host for evaluation. The workstation is then free to be used for any other purposes. When the solution is complete, the output analysis phase permits the graphical and tabular display of any desired results. The graphical facility provides a flexible mechanism for plotting multiple performance measures. This will allow the user, through a consistent interface on the PC, to iteratively create the model, view the results of the analysis or simulation, revise the model based on the output, and compare results for families of models.

Future work is planned to extend the capabilities of the graphical performance modeler's environment. This work will include

- a tutorial facility to teach users about the structures and features of RESQ in a programmed-learning environment,
- an "include" facility to permit the user to extract segments from completed models as building blocks for new models,
- higher-level modeling by permitting the design and use of "smart" icons to represent particular models or submodels,
- animation capabilities to display the movement of jobs on the model diagram and the accompanying changes in performance measures,
- a database component to manage the storage and manipulation of models and permit the merging of results from solutions of related models,
- an experimental-design facility to aid the analyst in optimizing model designs through appropriate parameter-search techniques.

Acknowledgments

We would like to express our appreciation to Richard Gilbert for a number of fruitful discussions, which significantly influenced our choice of the design features of the environment.

References

Brow85 J. Browne et al., "Graphical Programming for Simulation of Computer Systems", *Annual Simulation Conference,* pp. 109-128., 1985

Gilb85 R. Gilbert and W. Kleinöder, "CNMGRAF - Graphic Presentation Services for Network Management", *Proc. 9th Data Communications Symposium*, (Whistler Mountain, BC), pp. 184-199.

Heal85 K.J. Healy, "Cinema Tutorial", *Proc. 1985 Winter Simulation Conference*, (San Francisco), pp. 94-100.

IRA81 "Performance Analyst's Workbench System (PAWS) Users Manual, Information Research Associates, Austin, TX, 1981.

Mela85 B. Melamed and R.J.T. Morris, "Visual Simulation: The Performance Analysis Workstation", *IEEE Computer*, Vol 18, No. 8 (Aug. 1985), pp. 87-94.

Pegd85 C. Pegden, "Introduction to SIMAN", *Proc. 1985 Winter Simulation Conference*, (San Francisco), pp. 66-73.

Russ85 E.C. Russell, "SIMSCRIPT II.5", *Proc. 1985 Winter Simulation Conference*, (San Francisco), pp. 57-59.

Saue81 C.H. Sauer and K.M. Chandy, *Computer System Performance Modeling*, Prentice Hall, 1981.

Saue82 C.H. Sauer, E.A. MacNair and J.F. Kurose, "RESQ: CMS User's Guide", IBM Research Report RA-139, Yorktown Heights, N.Y., April 1982.

Saue84 C.H. Sauer, E.A. MacNair and J.F. Kurose, "Queueing Network Simulation of Computer Communication", *IEEE Journal on Selected Areas in Communications*, Vol. SAC-2, No. 1, Jan. 1984, pp.203-220.

Sinc85 B. Sinclair, K. Doshi, S. Madala, "GIST: A Tool Specifying Extended Queueing Network Models", *Proc. 1985 Winter Simulation Conference*, (San Francisco), pp. 290-300.

Trip84 S. Tripathi et al., "STEP-1: A User Friendly Performance Analysis Tool", *Proc. Int. Conference on Modeling Techniques and Tools for Perf. Analysis,* INRIA, Paris 1984.

Appendix A

This appendix contains the attribute specification for the RESQ performance model shown in Figure 5.

```
MODEL:csmtm
   METHOD:simulation
   NUMERIC PARAMETERS:thinktime users partitions
   NUMERIC IDENTIFIERS:floppytime disktime cputime
      FLOPPYTIME:.22
      DISKTIME:.019
      CPUTIME:.05
   NUMERIC IDENTIFIERS:cpiocycles
      CPIOCYCLES:8


QUEUE:floppyq
   TYPE:fcfs
   CLASS LIST:floppy
      SERVICE TIMES:floppytime
```

```
QUEUE:diskq
   TYPE:fcfs
   CLASS LIST:disk
      SERVICE TIMES:disktime

QUEUE:cpuq
   TYPE:ps      /* processor sharing service discipline */
   CLASS LIST:cpu
      SERVICE TIMES:cputime

QUEUE:terminalsq
   TYPE:is
   CLASS LIST:terminals
      SERVICE TIMES:thinktime

QUEUE:memory
   TYPE:passive
   TOKENS:partitions
   DSPL:fcfs
   ALLOCATE NODE LIST:getmemory
      NUMBERS OF TOKENS TO ALLOCATE:1
   RELEASE NODE LIST:freememory


CHAIN:interactiv
   TYPE:closed
   POPULATION:users
   :terminals-> floppy; .1
   :getmemory->floppy; .1
   :cpu->floppy; .1
   :terminals->disk; .9
   :getmemory->disk; .9
   :cpu->disk; .9
   :floppy->freememory;1/cpiocycles
   :floppy->cpu;1-1/cpiocycles
   :disk->freememory;1/cpiocycles
   :disk->cpu;1-1/cpiocycles
   :freememory->terminals
```

```
QUEUES FOR QUEUEING TIME DIST:memory
   VALUES:1 2 3 4 5 6 7 8
QUEUES FOR QUEUE LENGTH DIST:memory
MAX VALUE:users/2
CONFIDENCE INTERVAL METHOD:replications
INITIAL STATE DEFINITION -
CHAIN:interactiv
NODE LIST:terminals
   INIT POP:users
CONFIDENCE LEVEL:90
NUMBER OF REPLICATIONS:5
REPLIC LIMITS -
   QUEUES FOR DEPARTURE COUNTS:memory
      DEPARTURES:1000
LIMIT - CP SECONDS:50
TRACE:no
```

THE PERFORMANCE ANALYSIS WORKSTATION: AN INTERACTIVE ANIMATED SIMULATION PACKAGE FOR QUEUEING NETWORKS

B. MELAMED

AT&T Bell Laboratories
Holmdel, NJ 07733

ABSTRACT

The advent of low cost high powered intelligent workstations equipped with high resolution displays is gradually changing the practice of performance analysis. Exciting new visual tools are now emerging at research laboratories and in the marketplace.

A typical visual tool combines text and graphics in a user-friendly interface. The interface supports three main functions: system specification (editing icons and parameters), performance evaluation (simulation or analysis), and displaying statistics (time-series, histograms and summary reports). Visual tools reward users with higher productivity, ease of communication and a vastly more pleasing work environment when compared to traditional programming tools.

This paper describes a visual simulation tool dubbed the Performance Analysis Workstation (PAW), currently under development at AT&T Bell Laboratories. PAW enables a user to draw a queueing network model on a CRT screen with a mouse, parameterize the model by filling out forms from a keyboard and then make animated simulation runs displaying traffic flows and evolution of statistics.

1. THE ENTERPRISE OF MODELING

A model is a simplified representation of an entity. A model purports to capture certain behavioral aspects of the modeled entity (see [1-2]), and is only as good as its success in reproducing select behavior of the modeled entity. Examples are: a scale model of an airplane in a wind tunnel; a set of equations describing the state evolution of a queueing network; a computer program describing a protocol across a communication line.

The enterprise of modeling consists of building models, checking for their goodness, and using them to generate predictions. The utility of models lies in their predictive power; the modeler is interested in the value added of new information gleaned from models, and the subsequent scientific and economic benefits engendered by it. Since a model is used to understand and predict the behavior of complex systems, it must combine and balance the opposite requirements of faithfulness and simplicity. Consequently, the process of modeling is often tedious, complex and error-prone. It requires both knowledge and intuition, and often proceeds in cycles of "try and modify", backtracking and refinements. In many cases, modeling is more Art than Science, and good performance analysis is often the domain of experts.

The modeler starts with a conceptual model which is invariably a verbal description of the modeled entity's structure, rules of behavior and numerical parameters. The conceptual model must often be further reduced to a form that can be "solved". Models and their solution methods can be grouped into two broad categories: analysis and simulation. This paper will concentrate on Monte Carlo simulation and the role of workstations in hosting visual computer tools to aid users in simulation modeling.

2. SIMULATION VS. ANALYSIS

A Monte Carlo simulation is a computer program used to generate sets of conceivable system histories. Data collected from those histories are cast into statistics which serve as estimates of system performance. Random phenomena are driven by pseudo random number generators, thus conjuring up the chance experienced by gamblers at that ultimate casino of Monte Carlo.

The statistical approach, so central in simulation, contrasts sharply with analytic methods where mathematical tools (equations, etc.) are used to compute performance measures of sample aggregates (means, probabilities, etc.). An analytic approach rarely addresses transient phenomena or individual histories; rather, it usually deals with systems in steady state.

The trade-off between an analytic approach and a simulation approach can be summarized as follows:

A tractable analytic solution often requires stringent assumptions on the model to be solved. However, if those assumptions can be validated, the analytic method used is likely to require a fraction of the computer time consumed by a corresponding simulation program. Simulation models can be used validly for a far larger class of conceptual models. A lesser degree of abstraction can be employed in their construction because a simulation program can mimic the behavior of the conceptual model more directly. However, a simulation can be extremely costly to run or plainly infeasible due to the computer time or memory necessary to achieve satisfactory statistical confidence.

All in all, it is harder on balance to develop a valid solvable analytic model, and deriving a solution where none is known can require considerable mathematical skill. In contrast, a simulation approach is engineering-like in nature, imposes on balance fewer restrictions on models, and the problem of how to derive an "unknown solution" never arises.

CH2345-7/86/0000/0729$01.00 © 1986 IEEE

3. MODELING WORKSTATIONS

The advent of affordable workstations with ever increasing computational power and rapidly improving graphics has recently rendered the concept of a simulation workstation an attractive and reachable goal. I argue that an animated simulation approach to solving models meshes neatly with a workstation environment where the user interacts with a graphics screen, keyboard and pointing device. Points supporting this argument are:

a) A workstation is a dedicated resource. Model building is inherently incremental, evolutionary and subject to cycles of "try-modify-test repeat." A workstation setting being a dedicated resource can allow the process to proceed swiftly and smoothly, through uninterrupted interaction between Man and Machine. Experimentation is thus encouraged and facilitated.

b) A workstation has graphics capabilities. Model simulations, particularly network models, have fundamental and inherent visual components. These include displaying a pictorial representation of the model and its statistics, and generating animated sequences which represent the model in operation (traffic flows, state changes and statistics evolution).

c) A workstation can support a user-friendly interface. It lends itself to object-oriented programming where icons depicted on the screen are directly manipulated by clicking the buttons of a pointing device. Models that can be manipulated visually appear more natural and concrete to the user.

As a result, the user community of modelers could gain in two professional segments. First, engineers will find modeling easier because of the reduced need for abstraction. For them, a simulation workstation fulfilling a CAE (Computer Aided Engineering) function would be a familiar and accessible tool. Second, mathematically-oriented users who are not proficient with computers could find the burden of interaction alleviated by a good user-interface shielding them from inessential programming detail.

The above discussion must not be construed as advocating the replacement of analytic methods by Monte Carlo simulation, or that analytic methods have no place in modeling. Quite to the contrary. Analysis and simulation are in fact complementary and each approach should be used where appropriate and economical, often on the same problem. A good modeling workstation should ideally use *both* simulation and analysis for cross validation and to obtain well-rounded understanding of the system under study. The point is, though, that a simulation approach has the most to gain from the visual capabilities of a workstation environment.

4. OVERVIEW OF THE PERFORMANCE ANALYSIS WORKSTATION (PAW)

The Performance Analysis Workstation (PAW) is a visual modeling tool with both simulation capabilities and interfaces to analytic tools, currently under development at AT&T Bell Laboratories [3]. A number of visual modeling tools have preceded PAW or are contemporaneous with it. Noteworthy commercial packages include SEE WHY^{TM} (by BLSL Inc.), RESQ^{TM} (by IBM; work is in progress [4]), IDSS^{TM} [5] (by Pritsker and Associates, Inc.), Model Master^{TM} (by GE), PAWS^{TM} (by Information Research Associates) and SIMAN^{TM} [6] (by Systems Modeling Corporation). SIMAN has a particularly nice graphics package called CINEMA^{TM} which allows users to draw a realistic-looking simulation scene complete with color and semantic icons. Recently, PC-SIMSCRIPT was enhanced with visual capabilities by a program called SimVision^{TM} [7].

Hardware and Software

PAW runs on a Teletype Dot Mapped Display 5620 terminal (DMD for short) in conjunction with a host computer running under the UNIX^{TM} operating system. A UNIX host is required for file operations because the DMD has no peripheral memory. The DMD has 1MB memory and is equipped with a keyboard and a three button mouse (a pointing device). PAW is available to the public from AT&T through the UNIX System Toolchest.

PAW is written in C and assumes that the host computer has a UNIX operating system. The user interface is menu driven with pop-up menus appearing on the screen so as to allow the user to make a selection with the mouse. The style of interaction with the user interface is object-oriented. Icons on the screen represent concrete objects (nodes, transactions, statistics, etc.) and the user refers to them by pointing (placing the mouse cursor in an icon) and depressing, releasing or clicking a mouse button (to specify a selection or cause an action). Mouse button functionality is programmed in software.

Human Factors

PAW was designed with the user in mind. A basic design decision was to trade off generality and modeling scope for simplicity and ease of use. PAW is self-teaching and requires little reference to its documentation in the course of a session. To achieve this goal, a special subwindow on the screen is dedicated to communication with the user (see Figures 1-3). Called the Reporter Corner, this subwindow is updated after every key stroke or button action with three messages:

1) PREVIOUS ACTION FEEDBACK: informs the user of the outcome of the most recent action. Erroneous actions are also accompanied by a double beep to draw the user's attention.

2) CURRENT MODE: reminds the user of the current operation mode. This is helpful to novices attempting to navigate the menu tree.

3) EXPECTED ACTION: details a suggested sequence of actions. This is particularly helpful in graphics operations.

In addition to placing a good deal of ready "help" function in the Reporter Corner, PAW will also place needed information in a reminder window, but only when necessary.

PAW extends the user a one step grace period following any text or graphics deletion operations. This allows the user to recover text and graphics deleted by mistake.

Error checking throughout PAW is extensive with detailed error reporting (PAW stores some 16K bytes worth of messages). Error localization is carried out as soon as possible, even in the middle of lexical tokens. Most errors are checked for in the editors as information is being entered, and very little is left to be checked in the simulator.

PAW recovers from user errors with default values inserted whenever missing or erroneous information is encountered. Thus, a model description is maintained consistent and error-free from PAW's point of view. Of course, high level modeling errors cannot be discovered by PAW, but rather by the user observing PAW. Consistency is not guaranteed only when PAW runs out of memory.

5. PAW's WORLD VIEW

PAW's world view is a queueing network. In this regard it is an extension of Jackson, BCMP and Kelly Networks [8-10]. PAW's simulator is a discrete event simulation system operating in a world of queueing networks.

PAW Entities

PAW's world consists in the main of two fundamental entities: nodes (geographical sites in a network) and transactions (jobs, customers, etc.) which circulate among nodes.

A node has associated with it a queue whose positions can accommodate (hold) transactions. A queue capacity can be finite or infinite, but it has to be strictly positive. Queue positions are ordered 1,2,... from head to tail. A block of contiguous positions starting at the head (position 1) can have servers associated with them. There must be at least one server, but obviously the number of servers cannot exceed the queue capacity. Nodes have incoming and outgoing paths so that a queueing network can be thought of as a directed graph.

A special kind of nodes called *environment nodes* represent the "outside" environment of the network; all other nodes are called *regular nodes*. An environment node that feeds into a regular node is called a *source*; an environment node fed by a regular node is called a *sink*. Regular nodes can be freely connected, but any two environment nodes *cannot*. Unlike regular nodes, environment nodes have no queues.

While nodes are static entities anchored to particular network locations, transactions are dynamic entities that circulate and move from node to node. Each transaction has two fundamental attributes associated with it: a *class tag* and a *family membership*. A transaction class tag is a dynamic attribute assigned to the transaction whenever it enters a regular node. Typically, class tags are used to denote the "state" of a transaction during the course of its life in the network. A node and class dependent priority can be specified by the user for transactions. All transactions with the same class tag at a given node will exhibit the same probabilistic behavior as regards external arrivals (from a source), service delays and routing decisions because they all have the same priority and share the probability distributions governing that behavior. A transaction may change class (and consequently priority) on routing to another node.

In contrast, family membership is a static attribute that cannot be changed. PAW keeps track of the total number of each family's members in the system by incrementing or decrementing appropriate counters whenever transactions are created or destroyed respectively. Note that there is no motivation to do the same for classes because transactions with the same class tag but in different nodes may well be conceptually unrelated. Family members, on the other hand, are conceptually related and they can "recognize" each other even when they reside in different nodes and carry different class tags. The family concept is motivated chiefly by situations where a transaction is split (spawns a batch) and the batch members must be eventually joined to "recover" the original transaction (e.g., messages which are split into interleaving packets that must be reassembled at their destination). Family membership can propagate and expand by further splitting.

In PAW, there are no special types of transactions. For example, a transaction modeling a token will have its service time set to infinity with probability one, but otherwise it appears like a normal transaction (a token is a transaction resource that cannot move in the network on its own; rather, another transaction must move it around through the so-called yanking action to be described later). Passive queues can be modeled in PAW, but they do not exist as such, because conceptual "token" transactions are allowed to mingle freely with "non-token" ones. Specialized modeling constructs are embodied in the various types of nodes supported by PAW, to be described next.

6. PAW NODES

The regular nodes in PAW fall into several categories: standard nodes, SPLIT nodes, JOIN nodes and YANK nodes. The list above is ordered in increasing specialization.

Standard Nodes

All PAW nodes are variations on the standard node mechanism. Transactions may be created dynamically at any source node during a simulation run according to a user defined *arrival interval distribution*, or statically at any regular node directly by the user. At arrivals times, transactions may arrive in batches according to a user defined *arrival batch distribution* or singly (batch size one); batch size of zero is also allowed, in which case no arrival occurs.

On arrival at any regular node, a transaction is assigned a user defined *class* and *priority*. It then attempts to seize a server position. If successful, its status changes to *busy* and it is said to join the *busy queue*. A service time is sampled for it by a user defined *service time distribution*, and the transaction will stay in the busy queue until its service time requirement is satisfied. If all servers are occupied (busy), the incoming transaction has its status set to *idle* and it is said to join the *idle queue*. Transactions in the idle queue have no service dispensed to them; rather, they queue up until a server becomes available at which point in time they switch from the idle queue to the busy queue. A user specified mechanism called the *queueing discipline* governs this behavior. A transaction may move from the busy queue to the idle queue if the queue discipline permits preemption; in this case, PAW will automatically keep track of the residual service time of the preempted transaction.

Eventually, the transaction will finish service and attempt to route to another node according to a user defined *routing rule*. If the destination node is a sink node, the routing always *succeeds*. However, if it is a node with finite capacity, the routing may *fail* for lack of queue space. At this point, the transaction can have another shot at an *alternate routing rule*, provided the user had defined it (alternate routing is optional). If the alternate routing fails or if it is undefined, the transaction status changes to *blocked* and it is said to join the *blocked queue*. In this case, the transaction remains in its origination queue and waits there until a free queue position becomes available at its destination node. The server of the blocked transaction remains unavailable throughout the blocking period. Eventually, a queue position may become available at the destination queue. The transaction will then *unblock* by moving to the destination node where it will be assigned a new class and priority. The process then repeats itself.

A transaction is destroyed when it leaves the network by entering a sink node. A transaction attempting to enter a full node from a source is not blocked but immediately destroyed; such transactions are said to be *lost*.

PAW maintains in each node three contiguous disjoint subqueues. These are (from head to tail) the blocked, busy and idle queues. Within each of them, PAW implements queue ordering as follows:

1) Blocked transactions are ordered from head to tail by blocking clock time; that is, those that blocked earlier precede those that blocked later.

2) Busy transactions are ordered from head to tail by residual service time; that is, transactions with shorter residual service time precede those with longer residual service times.

3) Idle transactions are ordered from head to tail according to the node queue discipline within priority sets; that is, higher priority transactions precede lower priority ones.

The queueing disciplines supported by PAW at standard nodes are:

- FCFS = First Come First Served (within priority sets). Also commonly called FIFO.
- LCFS = Last Come First Served (within priority sets). Also commonly called LIFO.
- PS = Processor Sharing. Here the set of transactions with the highest priority share the (single) server. If n are present, their service is being completed at rate $\frac{1}{n}$.
- IS = Infinite Server. Every transaction in the node is guaranteed a server immediately on entering the node.
- PR_SRT_FCFS = Preemptive Resume, Shortest Residual Time, First Come First Served. Same as FCFS, except that preemption is allowed in which case the busy transaction with the *shortest* residual service time is selected for preemption.
- PR_LRT_FCFS = Preemptive Resume, Longest Residual Time, First Come First Serve. Same as PR_SRT_FCFS, except that the busy transaction with the *longest* residual service time is selected for preemption.
- PR_SRT_LCFS = Preemptive Resume, Shortest Residual Time, Last Come First Serve. Similar to PR_SRT_FCFS, except that the idle queue is LCFS.
- PR_LRT_LCFS = Preemptive Resume, Longest Residual Time, Last Come First Serve. Similar to PR_LRT_FCFS, except that the idle queue is LCFS.

SPLIT and JOIN Nodes

SPLIT nodes are essentially FCFS nodes with infinite queue capacity. However, on service completion of a transaction at a SPLIT node, that transaction (called the *parent*) samples a batch size from a user defined *split batch distribution* and spawns the sampled number of transactions (called *children*). If the distinction between parent and children is unimportant, they are collectively called *siblings*. The children may or may not have the class tag or family membership of the parent. All transactions involved in the split event then attempt to route out of the SPLIT node according to their prescribed routing rules.

PAW supports the following SPLIT nodes:

- CL_SPLIT = Class-Split. All transactions involved in the split event are guaranteed to have distinct family memberships. In particular, each child forms a singleton family.
- FM_SPLIT = Family-Split. All transactions involved in the split event (parent and children) are guaranteed to belong to the same family.

JOIN nodes perform roughly the opposite function of SPLIT nodes. The first transaction (in a batch to be joined) to arrive at a JOIN node samples a batch size from a user defined *joined batch distribution*. The transaction class and sampled batch size specify a *join demand* which is then posted at the JOIN node. The first transaction and subsequent ones (which arrive in due time) wait idly until the sampled batch size is met. At that point all batch members are collapsed into a single transaction. The resultant transaction changes status to busy and assumes a user defined class tag; it then proceeds to be served by one of the node's infinite number of servers, before routing out as usual. Busy and blocked transactions are excluded from further batching and join events at the JOIN node. Multiple batching of disjoint batches may take place simultaneously. Joining like splitting may involve transactions of the same class or same class and family.

PAW supports the following JOIN nodes:

- CL_JOIN = Class-Join. Transactions of the same class are batched for joining. Infinite batch size is allowed, but in this case that batch will never join; in effect, the batch will behave like tokens (passive resource).
- FM_JOIN = Family-Join. Transactions of the same class and family are batched for joining. The effective batch size used by PAW is always the minimum of the sampled size and the total family size in the system. Infinite batch size is allowed but it simply means that joining occurs when all family members congregate in the JOIN node and all carry the same class tag.

SPLIT and JOIN nodes can be used to model the following phenomena:

1) Transaction generation: SPLIT nodes provide an alternate means of complex transaction generation. In particular, a SPLIT node can be conveniently used to model the simultaneous arrival of multiple transaction

batches each having a different class. In this case, a generator transaction enters a SPLIT node repeatedly (via a feedback path). On each visit it enters the node with an appropriate class tag and spawns the requisite batch (after spending zero time in service).

2) Assembly operations: Class-JOIN nodes can be used to model factory assembly operations where finished products are assembled from various parts.

3) Synchronization points: Family-JOIN nodes provide a natural way to model synchronization points, especially situations where a parent job spawns subjobs which must all be completed before the parent job is allowed to proceed. The family-JOIN node is necessary to group precisely the subjobs that were spawned by a given parent.

YANK Nodes

Yank nodes are special nodes where a transaction is allowed to yank some other transaction from another node and route it elsewhere. In effect, yanking is an extreme case of preemption followed by rerouting. Unlike preemption, the status of the yanked transaction does not matter; while preemption only applies to busy transactions, yanking applies to blocked, idle or busy ones. Furthermore, yanking does not have the resume aspect of preemptive-resume; rather, the yanked transaction is "restarted" at its destination.

The yanking mechanism is non-trivial and operates as follows. A transaction entering a YANK node may or may not have a *yank targets distribution* specified for it. If none is specified, that transaction is considered a *pass-through* transaction and routes out of the YANK node without delay. Otherwise, that transaction samples a *target node* from which to yank, a *target class* to be yanked and a *batch size* which together specify a *yank demand.* It then posts the yank demand at the target node and attempts to satisfy it. A yanking action is executed as follows. Each member of the target batch is moved from the target node to a destination node. PAW determines the destination node by appeal to the routing rules of the yanked transactions at the YANK node. Each yanked batch member may have its class changed on entry at the destination node where it is considered an ordinary arrival. If a yanked transaction gets blocked, it will stay blocked at the YANK node. A yank event is said to *succeed* if the posted yank demand can be satisfied by transactions on hand; otherwise it is said to *fail.* If the yank event fails and the yanking transaction has no alternate routing defined for it, then that transaction will wait at the YANK node until its yank demand is completely satisfied. If, however, the alternate routing rule is defined for it, that transaction will invoke it without completing the yank action. When a yank event succeeds, only the routing rule is invoked for the yanking transaction.

A YANK node may yank from another YANK node but not from itself. Like splitting and joining, yanking can be programmed for class alone or for family and class; the family to be yanked is implicitly that of the yanking transaction. When specifying a class to be yanked, the user may specify a single class or the keyword "all" meaning "yank regardless of class".

It is common for multiple yanker transactions (possibly at distinct YANK nodes) to attempt to yank the same target transactions from a common target node. Such situations model contention for transaction resources (as opposed to contention for server resources which occurs within a standard node). In particular, if a YANK node targets a JOIN node, then yanking and joining demands may coexist in contention. PAW resolves contention for transactions by ordering the incoming demands first by priority (of the yanking and joining transactions) and then FIFO by the time the demands were posted.

The discipline of satisfying a yanking demand comes in two flavors: first-fit and immediate. In both cases, a demand list is scanned head to tail. If a first-fit yank demand is encountered, it will be satisfied only if its posted batch size can be met by transactions on hand. In contrast, an immediate yank demand may be partially satisfied by any number of transactions on hand, and the corresponding posted batch size is accordingly reduced.

PAW supports the following YANK nodes:

- FF_CL_YANK = First-Fit Class-Yank. First-fit yanking is carried out for target transactions in a prescribed class or regardless of class.
- FF_FM_YANK = First-Fit Family-Yank. First-fit yanking is carried out for target transactions in a prescribed class or regardless of class, but only if they belong to the same family as the yanker transaction.
- IM_CL_YANK = Immediate Class-Yank. Same as FF_CL_YANK except that immediate yanking is employed.
- IM_FM_YANK = Immediate Family-Yank. Same as FF_FM_YANK except that immediate yanking is employed.

YANK nodes can be used to model the following phenomena:

1) Multiple resource possession: Token transaction are placed in a node representing a pool of available tokens. A transaction requiring multiple resources enters a class-YANK node and attempts to yank token transactions from the pool of available tokens to another node holding all unavailable tokens. Eventually that transaction will release the resources by yanking token transactions from the pool of unavailable tokens back to the pool of available tokens. Transactions that cannot find tokens to yank from the pool of available tokens must wait until they become available.

2) Finite queues: Although PAW admits finite queues in standard nodes, it is sometimes required to restrict the entrance of certain transaction classes into a standard node with an infinite queue. For example, in a machine breakdown model, a preemptive-resume node with infinite queue represents the machine, low priority transactions represent the product worked on by the machine and high priority transactions represent breakdowns. The infinite queue is necessary to allow any breakdown transaction to enter the machine node without blocking. However, product transactions can enter the same only when buffer space is available. This can be implemented via the token scheme described above.

3) Time-out: To restrict the time a target transaction may spend at a node, a time-out transaction is created in a family-SPLIT node and proceeds to be delayed, say, in an infinite server node. The time-out transaction then enters a YANK node and attempts to yank the target transaction. If successful, that means that the target transaction has timed out; otherwise, the target transaction has beaten the time-out "clock" and will usually proceed to destroy the time-out transaction by yanking it from the delay node to a sink. A similar setup can restrict the time a transaction is allowed to spend in a set of nodes (subnetwork).

7. PAW STATISTICS

PAW permits statistics collection in the background or in the foreground (in a statistical window). All statistics are displayed up to date on every screen refresh. PAW statistics are packaged in three formats: summaries, time-series and histograms.

A summary maintains eight numerical statistics: number of observations collected, the most recent observation, minimum observation, maximum observation, average, variance, standard deviation and coefficient of variation. A time-series is a pictorial representation of observations and the times they were collected, displayed as a sequence of bars. A histogram is a pictorial representation of an empirical distribution computed from collected observations. Both time-series and histograms also display the respective summary statistics, space permitting (see Figures 1-3). PAW will automatically redisplay updated pictorial statistics whenever the display window is reshaped or refreshed.

Once the user selects the statistics format from a menu, the particular statistical aspect to be collected can be chosen from a menu of fourteen items:

1) Server utilization
2) Total queue length
3) Idle queue length
4) Busy queue length
5) Blocked queue length
6) Total input interval
7) Total output interval
8) Lost arrival interval
9) Node sojourn time
10) Node idling time
11) Node service time
12) Node blocking time
13) Network sojourn time
14) Network cycle time

Utilization and *queue length* statistics are computed as time integrals. The *total input interval, total output interval* and *lost arrival interval* are respectively the time intervals between successive arrivals, departures, and losses at a node; they can be used to obtain traffic rates (e.g. throughputs). *Node sojourn time* is the total time spent by a transaction in a node during a visit. *Node idling time* and *node service time* are the time periods a transaction is idle or busy respectively during a node visit; they are of interest in preemptive resume or processor sharing nodes. *Network sojourn time* is the total time spent by a transaction in the network, while *network cycle time* is the time elapsed between a transaction's departure from a node and its next return to it (useful for collecting "response time" observations). In any given node, statistics can be customized to be collected only for particular classes, as well as for all classes.

8. PAW ARCHITECTURE

PAW software is made up of four components: graphics editor, text editor, Monte Carlo simulator and utilities. The menu mechanism allows the user to quickly and easily walk among components.

The Graphics Editor

The creation of a new model must always begin in the graphics editor. Using the mouse, the user draws the individual nodes of the network, and then specifies the topology by connecting nodes as necessary. "Node drawing" is in fact a simple process because the node icon has been preprogrammed; by moving the cursor, the user merely changes the node size and orientation in the plane. Nodes can be deleted from the screen, in which case all node information is automatically purged from the model. A node icon can also be moved around on the screen. This is done by placing the cursor in a node icon, pressing a button, "dragging" the icon to its requisite place on the screen and then releasing the button. Every node may have a source and a sink. However, all source and sink nodes denote the same "environment" node. A distinct label is generated and associated with each regular node. Node labels can be made to appear and disappear on the screen and can be changed in the text editor.

To reserve screen space for statistical displays, the user sweeps out rectangular areas called *windows*. Statistical windows can also be moved around or deleted. PAW allows the user to bind a statistic to a window (an operation called *pegging*), to clear a window of a statistic or to delete the window altogether. The last two operations never delete the statistic itself; rather, statistics collection is moved from the foreground to the background.

The Text Editor

PAW's text editor is used to parametrize a model and define its statistics. All text data are entered in captioned fields of appropriate forms. The user can scan, enter data or delete data while moving among the form's items in a wrap-around manner. The advantage of form entry is that the user has almost no syntax to remember. On the other hand, this necessarily limits the expressive power of the system. Still, PAW can model a respectably large class of queueing networks.

PAW's text editor checks every input token thoroughly for lexical and some semantic errors. For example, node and transaction labels are checked for validity and uniqueness; routing is checked for discrepancies with the drawn topology; discrete distributions are checked so as not to exceed one, etc. Error localization occurs at the first erroneous character; a double beep sounds and the offending character is rejected or an appropriate default is used.

The Monte Carlo Simulator

PAW's simulation activities are largely controlled from the simulation panel and its nine windows located at middle bottom of the screen (see Figures 1-3):

1) Stop window: when highlighted, the simulator is stopped.

2) Continuous window: when highlighted, the simulator is running in continuous mode.

3) Step window: when highlighted, the simulator is running in step mode.

4) Current window: current simulation time.

5) Interval window: simulation time interval since last screen refresh.

6) Reset window: simulation time to reset all model statistics.

7) End window: simulation end time.

8) Realtime window: real time delay between screen refreshes.

9) Snapshot window: simulation snapshot interval. When zero, every event is animated; when positive, successive snapshots are displayed at the specified time granularity.

PAW supports a variety of run modes and options to suit the requirements of various stages in the modeling process. When a model is first constructed and debugging gets under way, the modeler wants to examine the detailed (microscopic) behavior of the model. As confidence in the model builds up due to modifications and bug removal, the modeler's interest turns to more global (macroscopic) behavior. Finally, when the model is deemed valid, the modeler often makes long production runs and collects final statistics with few or no intermediate snapshots. This last stage rarely requires interaction and is preferably launched in batch mode.

In PAW, the user can set the snapshot window to any nonnegative value. To examine the simulation event by event, the user sets the snapshot window to zero. In this case, screen animation displays the shuttling of transactions among nodes, their creation and destruction and more complicated behavior such as splitting, joining and yanking. For added convenience, the user can elect to step through events or to run the animation continuously, somewhat like a movie. When the snapshot window is set to a positive value, the screen is refreshed every snapshot interval to give a time lapse account of model behavior at the specified time granularity. Again, snapshots can be stepped through or run continuously. Finally, to launch production runs in batch mode, the user can select the upload option and specify an output file. The model on the DMD screen is then uploaded to the host, a batch run is initiated and the final state of the model will be saved in due time in the designated output file. The user can later download the output file from the host to the DMD and look at the final state and statistics. Uploading simulations is particularly desirable when the host has floating point hardware because of the speedup gains (the DMD has all floating point in software).

Uploaded PAW simulations can be run outside PAW under UNIX. In a typical scenario, the user runs the DMD under the *layers* program [11] which supports multiple windows. The user downloads PAW in one window, and launches one or more batch simulations of an already debugged model. To track the progress of a simulation run, the user may interrupt it at any time. PAW arranges to save a model snapshot on the nearest event boundary. The user can then download the snapshot and look at it at leisure. The run itself can be restarted either from the saved snapshot or from the downloaded one, etc. Alternatively, the user may investigate a snapshot's microscopic behavior by running it on the DMD and perhaps later resume it in batch mode.

In a more complicated situation, the user may wish to run replications, step through a model's parameters for a sensitivity analysis, or compare multiple models. To facilitate these kinds of activities, PAW provides a host programming environment that allows the user to access and change model parameters through an interface package. The user writes a control program in C that takes one or more PAW models, modifies them as necessary and calls the PAW simulator to produce the requisite runs. The user is then responsible for processing summary statistics such as confidence intervals, $x-y$ plots or statistical tests. Only when using the host programming environment do users assume the responsibility for any programming bugs resulting from their code; on the DMD, they are only responsible for "high level" errors.

Utilities

PAW provides an array of utilities, mostly involving file manipulation and translation.

In PAW, models are always created on the DMD; but because the DMD lacks peripheral memory, models must be saved on the host. A PAW model is always saved with its full state, including the random number generator seed. Consequently, a PAW simulation can be stopped at any time, a model snapshot saved in a file and later read back into the DMD (or the host) to resume the simulation from where it left off.

PAW models are saved in binary (machine-readable) form for the sake of efficiency. Another set of utilities produces human-readable printouts of PAW models. The user can get a listing of an entire PAW model or save a screen image in a file and print it as a hard copy. Individual statistics can be blown up on the screen and similarly printed.

PAW is envisaged as a single point of contact with multiple modeling and analysis tools for queueing networks. Currently, PAW includes interfaces to two analytical packages: PANACEA (which uses asymptotic expansion to solve large steady state queueing networks [12]), and QNA (which generates an approximate analytical solution for steady state queueing networks based on the first two moments of each modeling distribution [13]). Both tools were created in AT&T Bell Laboratories, but expect different source formats. PAW merely provides translators which take a PAW model as input and generate as output the source files in the requisite format. As usual with simulations, the modeling scope of the PAW simulator subsumes that of PANACEA and QNA by a wide margin.

Most PAW utilities can be invoked either from PAW (in which case they apply to the model on the screen), or from outside PAW (in which case they can be applied to arbitrary files). In particular, the upload facilities and the host programming environment of PAW enable the user to run simulations without being confined to the PAW screen environment. Downloading and uploading can be freely mixed in any combination. Listing, screen hard copy and translation utilities can also be invoked directly from UNIX in order to save the overhead of file transfer between the DMD and the host.

9. INTERACTING WITH PAW

A typical interaction scenario with PAW has a distinct pattern, extending possibly over multiple sessions. The following stages sketch the broad outline of a typical PAW modeling and simulation scenario.

1) The user always begins in the Graphics Editor. At this stage, nodes are created and connected, their labels placed on the screen and statistical windows are swept out.

2) The user enters the Parameters Editor. Form entry is used to specify and edit mostly numerical data describing the behavior of nodes and transactions.

3) The user enters optionally the Statistics Editor to define statistics and display them in statistical windows.

4) The user enters the PAW Simulator to verify the PAW model (i.e., to ascertain that it represents the conceptual model) and sets the windows of the simulation panel. Testing and debugging runs usually start in step mode and progress to continuous mode with the snapshot window set to zero. The user observes sequences of event-by-event animations and notes the flow of transactions, the sequence of clock values and the evolution of select statistics. Later, the user sets the snapshot window to a positive value and proceeds to observe sequences of model snapshots at increasing time granularities. Any discrepancy noted between the PAW model and the conceptual model requires the user to loop back to any of the previous stages to modify the PAW model's topology, parameters or statistics.

5) If a real system gave rise to the conceptual model, then model validation is called for to ascertain that select aspects of the real system are in sufficient agreement with their counterparts in the PAW model. To this end the user launches production runs with few snapshots, or may elect to upload simulation runs to the host to attain additional speedups. The statistical summaries obtained from such runs are then compared to those of the real system. If agreement is not satisfactory, the user may loop back to earlier stages for further rounds of model modification, verification and validation. If possible, PAW's interfaces to the analytical packages may also be invoked to cross-check the simulation and thus aid in validation.

6) Finally, when the PAW model is verified and validated, the user has finally reached the point where the model's predictive power may be used to gain new information. The user can elect to step through model parameters, different topologies, etc., and make production runs to answer "what-if" questions. The host programming environment can be used for these activities as well as statistical post-processing (confidence intervals, hypothesis testing, comparison plots, etc.).

The above scenario sketch only gives an idea of the interaction between the user and PAW. It glosses over the important interactions among analyst colleagues or between analyst and client. Inter-user communication is greatly facilitated by PAW since phenomena can be jointly observed on the DMD screen rather than merely talked about.

10. AN EXAMPLE: A PAW MODEL OF PACKET VOICE

In a transmission system, multiple sources feed streams of packetized voice into a high speed line. The conceptual model is designed to capture the bursty nature of human speech. A voice source is either on (active) or off (silent), corresponding to a customer talking or pausing respectively. When on, the voice waveform is sampled, and a voice packet is generated every 16 ms. A voice burst is assumed to be exponential in length with mean 352 ms. When off, the voice source is silent for an exponential time with mean 650 ms. The time to transmit a packet on the high speed line is .333 ms. The PAW model in Figures 1-3 is designed to model only the traffic flow of packets in the system.

To initialize the simulation model, (see Fig. 1) the user need only introduce the requisite number of voice sources. This is done by putting a set of transactions (in this case 100) with class tag *src* at node INIT. Because PAW allows the user to create transaction batches directly at any regular node, initialization requires just one action. Note that only six transactions are displayed in INIT - the first five and the last one. The gray area separating them is the "fog" covering all undisplayed transactions.

When the simulation is started, each *src* transaction in the family-SPLIT node INIT splits into a family of two siblings. One sibling goes to node SILENCE with class tag *gen*, and the other to node DELAY with class tag *idl*. Class *gen* transactions model the packet generation process by forever shuttling between nodes SILENCE and GENERATOR. When at node SILENCE, they have an infinite service time thus representing a pause in the speech pattern. However, while at the class-SPLIT node GENERATOR, a class *gen* transaction will keep spawning a class *pkt* transaction representing a voice packet every 16 ms. Each *pkt* transaction is routed immediately to the HSL node (representing the high speed line) for transmission before exiting the system. The parent *gen* transaction retains its class tag and simply loops back to GENERATOR for the next packet generation. A class *gen* transaction is delayed (16 ms) before each split, and packet generation by multiple class *gen* transactions proceed in parallel with no queueing interference, because GENERATOR has infinite servers.

The switching on and off of the packetizing process is governed by the sibling of each class *gen* transaction. Initially, that sibling resides in node DELAY with class tag *idl* for the duration of the silence period; again multiple sources co-exist there without interference because DELAY is an infinite server node. When the silence period ends, the class *idl* transaction is routed to the family-YANK node ON_OFF. From the ON_OFF node it seeks and finds its sibling in node SILENCE, and yanks it to node GENERATOR thereby inaugurating an active period during which packets are generated at GENERATOR and sent to the high speed line HSL. The yanker sibling is routed to node DELAY where it assumes class tag *act* and times the duration of the active period. When it is time to switch the line off, the class *act* sibling moves to node ON_OFF to yank its sibling from GENERATOR back to SILENCE. The process will then repeat indefinitely.

Figures 1-3 depict three snapshots in the evolution of the model. They were obtained with the DMD's screen copying utility and show the entire screen (note the Reporter Corner and simulation panel located below the model area).

In Figure 1, the initial state at time 0 is shown with all transactions in the INIT node. Two statistics are collected in the foreground: a time-series of the number of active lines (bottom left) and a histogram of packet delay at node HSL (bottom right). Summary statistics are shown at the right margin of each.

Figure 2 depicts the state just after initialization. All *gen* transactions are in node SILENCE and all their siblings appear as *idl* transactions in node DELAY. This state represents a "cold start", i.e., all channels start in silence mode. To get rid of startup conditions, the reset window has been set to 1000.0; consequently, all statistics will be reset after a warmup period of 1000 time units. A pilot run can be used to gauge a reasonable reset time. Because initialization takes zero simulation time, the simulation clock is still 0. Note that the simulator is in step mode (step window is highlighted).

In Figure 3, the network state has already evolved considerably. The snapshot shows that at time 7185.687, a buildup of active channels is causing some idling time delays of packets at node HSL (see right hand histogram). The time-series on the left shows that 33 channels are now active (and by implication 67 are idle). Although a small queue of *pkt* transactions is present at HSL, a dynamic view of the model (through event-by-event animation) reveals that those queues are intermittent; they build up to moderate size and then disappear periodically. An *idl* transaction is shown in node ON_OFF poised to yank its *gen* sibling from node SILENCE to node GENERATOR. Note that the simulator is stopped and that the second line in each statistics window reflects the fact that all statistics had been reset at time 1000.0.

11. CONCLUSION

The modeling constructs of PAW allow a quick and efficient modeling of a wide variety of systems and phenomena. For example, the split/join mechanism can model message packetizing, product assembly, process forking, job synchronization and more. The yank construct was found to be surprisingly powerful. With it one can model inventories, time-out, flow control, polling, multiprogramming level control, and general phenomena involving token manipulation such as gating, and multiple simultaneous resource possession.

Within AT&T, PAW has been used in dozens of projects and in a wide variety of applications including voice and data communications, computer systems and manufacturing. The response has been positive; users appreciate the ease of use and efficiency of modeling provided by PAW.

PAW is not yet a completely general purpose modeling tool. The fact that it lacks a general programming language, while often a strength, is sometimes a weakness. In particular, predicates are not supported beyond the built-in ones (for example, PAW supports a limited number of state-dependent routing rules). Nevertheless, PAW is adequate for modeling a large class of queueing systems. Where it can be used, PAW is quite effective. It is hoped that future enhancements will further increase PAW's expressive power and consequently its modeling scope.

Acknowledgements

I wish to thank Robert Morris and Harry Heffes for reading and commenting on this paper. Robert Morris participated in the design of PAW and, in particular, in developing the notion of YANK nodes. The PAW model of packet voice is included here courtesy of Harry Heffes.

REFERENCES

[1] Shannon, R. E. "Systems Simulation" Prentice Hall, 1975.

[2] Bratley P., Fox B. L. and Schrage, L. E. "A guide to Simulation", Springer-Verlag, 1983.

[3] Melamed, B. and Morris, R. J. T. "Visual Simulation: The Performance Analysis Workstation", IEEE Computer, Vol. 18, No. 8, pp. 87-94, 1985.

[4] S. S. Lavenberg, private communication, March 1986.

[5] "IDSS Prototype (2.0) Version 4, Users Reference Manual", Pritsker and Associates, Inc., 1983.

[6] Pegden, C. D., "Introduction to SIMAN," Systems Modeling Corp., 1982.

[7] Mullarney, A. "SimVision for PC-SIMSCRIPT", CACI, 1985.

[8] Jackson, J. R. "Networks of Waiting Lines", Operations Research, Vol. 5, pp. 518-521, 1957.

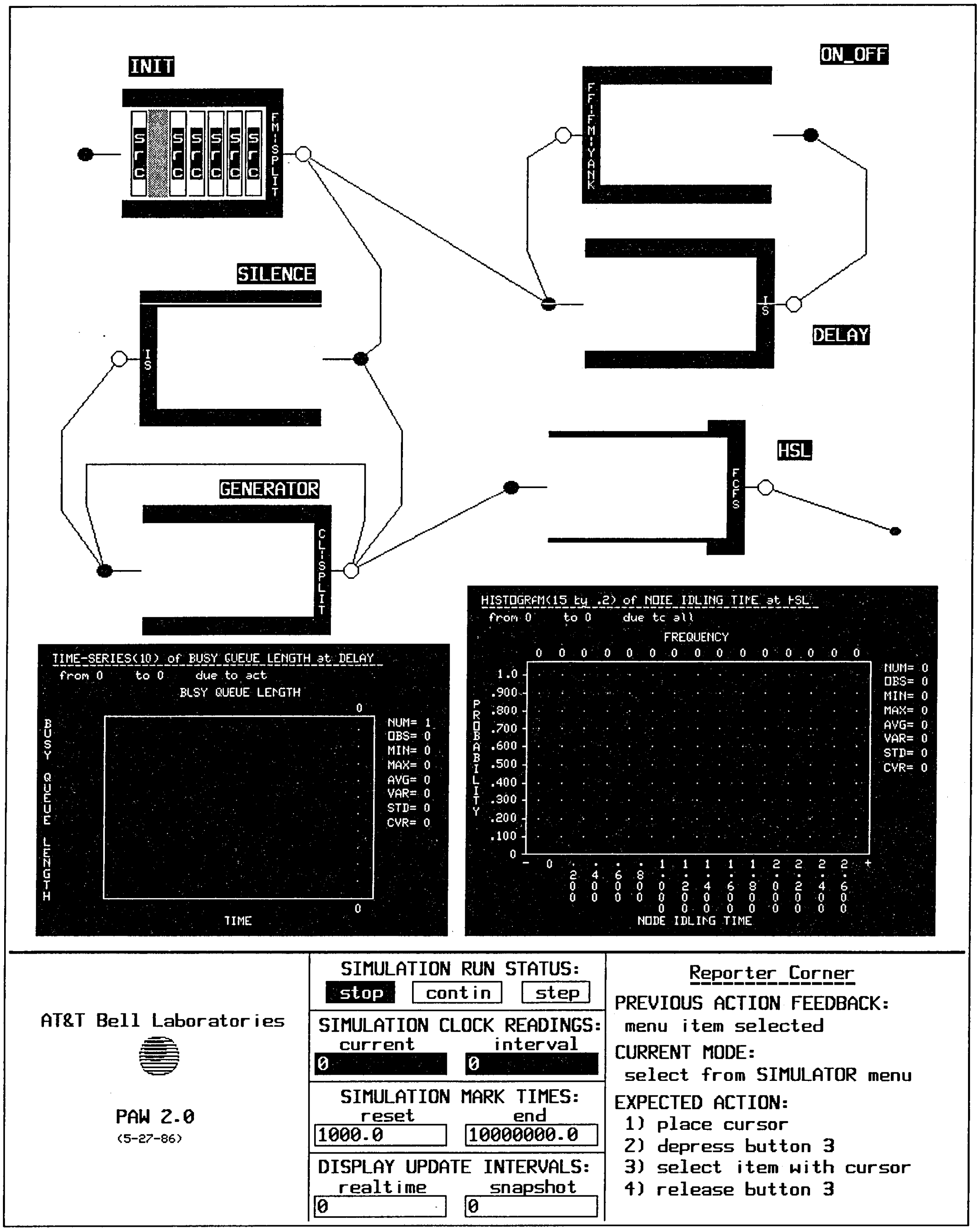
[9] Baskett, F., Chandy, K. M., Muntz, R. R. and Palacios, F. G., "Open Closed and Mixed Networks of Queues with Different Classes of Customers", JACM, Vol. 22, pp. 248-260, 1975.

[10] Kelly, F. P. "Reversibility and Stochastic Networks", John Wiley, 1979.

[11] Kelly, M. J., et.al., "An Intelligent Windowing Graphics Terminal for the UNIX System", EUUG Proceedings, Nijmegen, Netherlands, 1984.

[12] Ramakrishnan, K. J. and Mitra, D. "An Overview of PANACEA: A Software Package for Analyzing Markovian Queueing Networks", Bell System Technical Journal, Vol. 61, No. 10, pp. 2849-2872, 1982.

[13] Whitt, W. "The Queueing Network Analyzer", Bell System Technical Journal, Vol. 62, No. 9, pp. 2779-2815, 1985.



TIME-SERIES(10) of BUSY GUEUE LENGTH at DELAY
from 0 to 0 due to act
BLSY QUEUE LENGTH

NUM= 1
OBS= 0
MIN= 0
MAX= 0
AVG= 0
VAR= 0
STD= 0
CVR= 0

HISTOGRAM(15 by .2) of NOIE IDLING TIME at HSL
from 0 to 0 due tc all

NUM= 0
OBS= 0
MIN= 0
MAX= 0
AVG= 0
VAR= 0
STD= 0
CVR= 0

AT&T Bell Laboratories

PAW 2.0
(5-27-86)

SIMULATION RUN STATUS:

SIMULATION CLOCK READINGS:

current: 0
interval: 0

SIMULATION MARK TIMES:

reset: 1000.0
end: 10000000.0

DISPLAY UPDATE INTERVALS:

realtime: 0
snapshot: 0

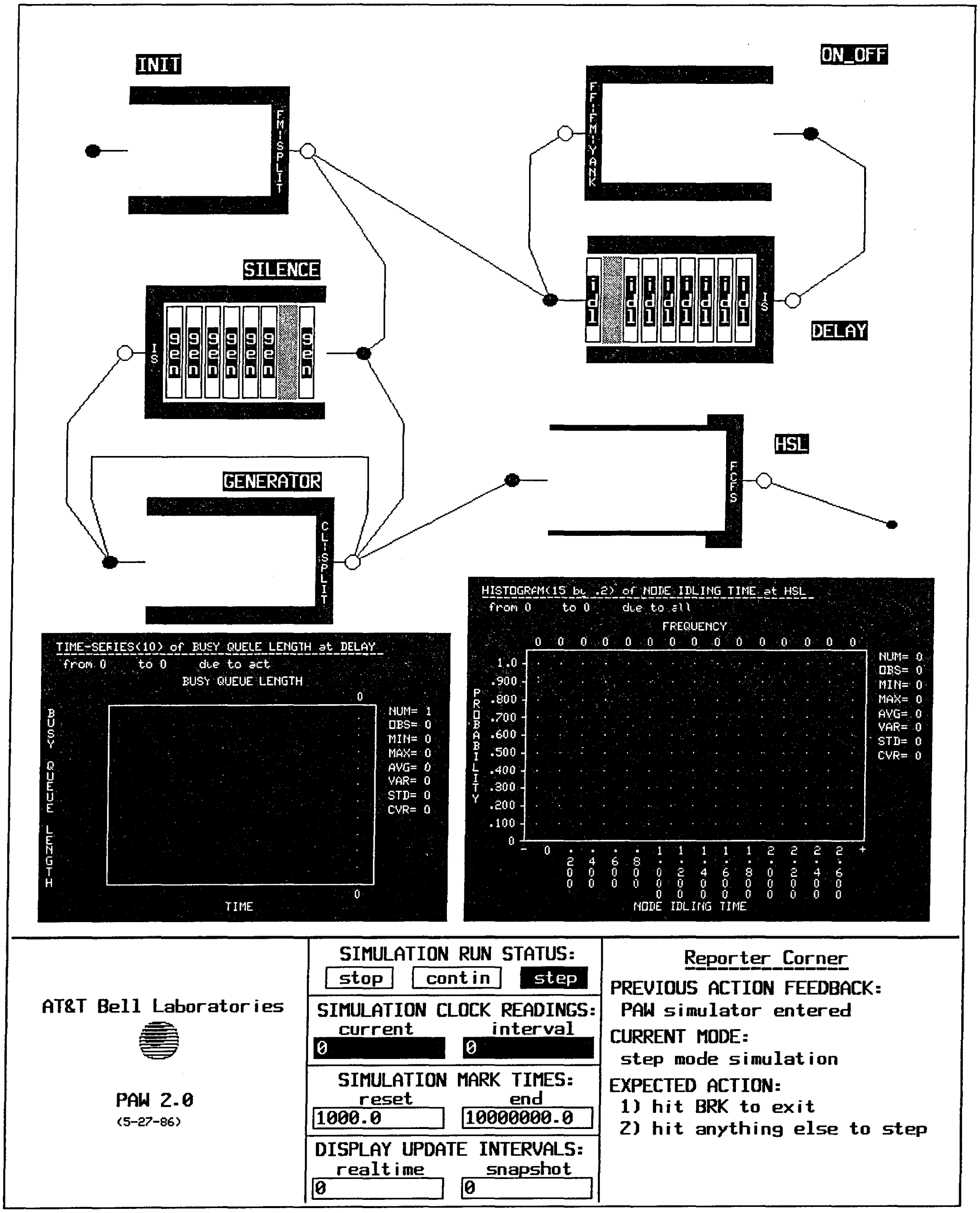
Reporter Corner

PREVIOUS ACTION FEEDBACK:
menu item selected

CURRENT MODE:
select from SIMULATOR menu

EXPECTED ACTION:
1) place cursor
2) depress button 3
3) select item with cursor
4) release button 3

Figure 1: First Snapshot of packet voice model



TIME-SERIES(10) of BUSY QUELE LENGTH at DELAY
from 0 to 0 due to act

```
NUM= 1
OBS= 0
MIN= 0
MAX= 0
AVG= 0
VAR= 0
STD= 0
CVR= 0
```

HISTOGRAM(15 by .2) of NODE IDLING TIME at HSL
from 0 to 0 due to all

```
NUM= 0
OBS= 0
MIN= 0
MAX= 0
AVG= 0
VAR= 0
STD= 0
CVR= 0
```

AT&T Bell Laboratories

PAW 2.0
(5-27-86)

SIMULATION RUN STATUS:

SIMULATION CLOCK READINGS:
current 0
interval 0

SIMULATION MARK TIMES:
reset 1000.0
end 10000000.0

DISPLAY UPDATE INTERVALS:
realtime 0
snapshot 0

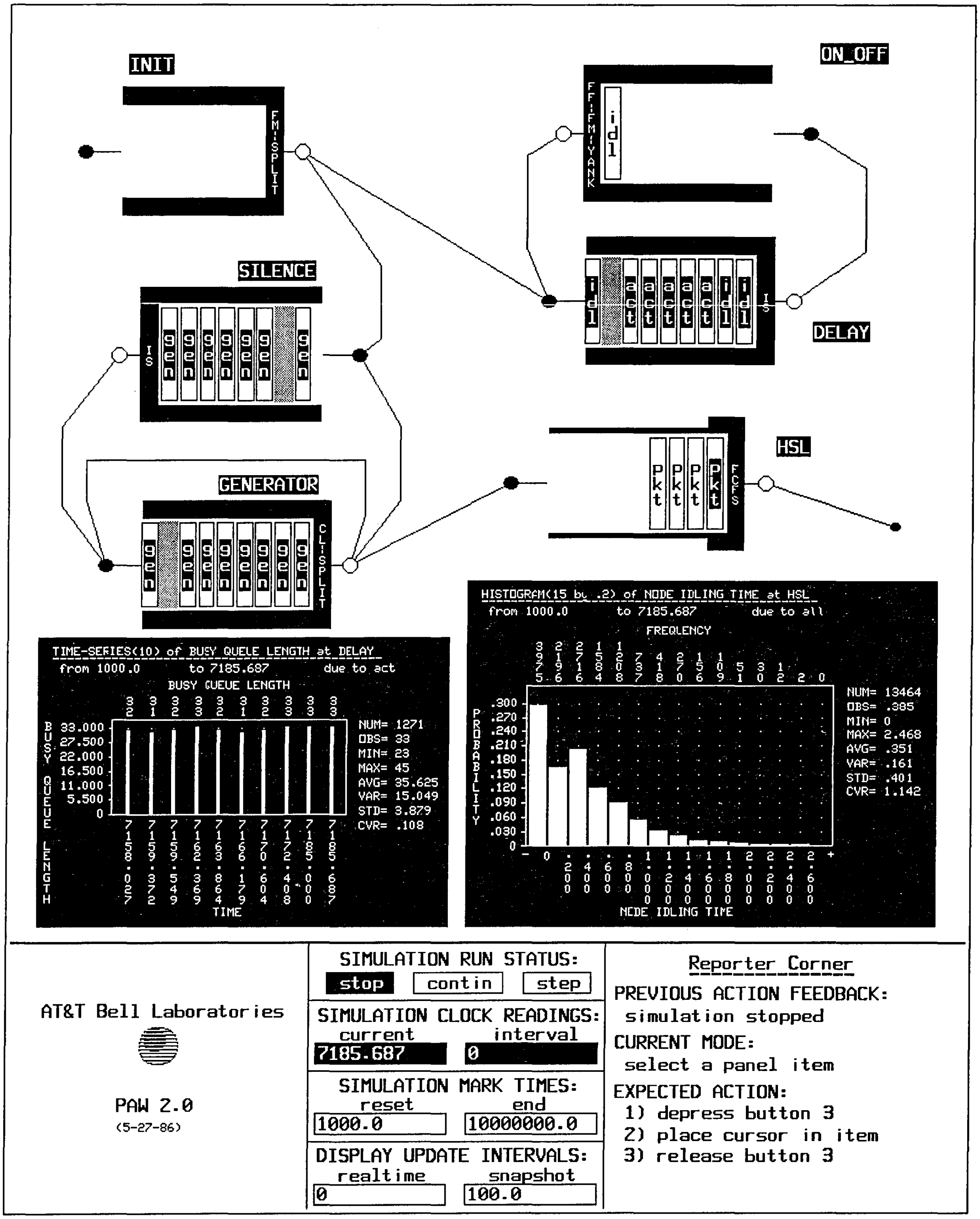
Reporter Corner

PREVIOUS ACTION FEEDBACK:
PAW simulator entered

CURRENT MODE:
step mode simulation

EXPECTED ACTION:
1) hit BRK to exit
2) hit anything else to step

Figure 2: Second Snapshot of packet voice model



TIME-SERIES(10) of BUSY QUELE LENGTH at DELAY
from 1000.0 to 7185.687 due to act

NUM= 1271
OBS= 33
MIN= 23
MAX= 45
AVG= 35.625
VAR= 15.049
STD= 3.879
CVR= .108

HISTOGRAM(15 by .2) of NODE IDLING TIME at HSL
from 1000.0 to 7185.687 due to all

NUM= 13464
OBS= .385
MIN= 0
MAX= 2.468
AVG= .351
VAR= .161
STD= .401
CVR= 1.142

AT&T Bell Laboratories

PAW 2.0
(5-27-86)

SIMULATION RUN STATUS:

SIMULATION CLOCK READINGS:
current: 7185.687
interval: 0

SIMULATION MARK TIMES:
reset: 1000.0
end: 10000000.0

DISPLAY UPDATE INTERVALS:
realtime: 0
snapshot: 100.0

Reporter Corner

PREVIOUS ACTION FEEDBACK:
simulation stopped

CURRENT MODE:
select a panel item

EXPECTED ACTION:
1) depress button 3
2) place cursor in item
3) release button 3

Figure 3: Third Snapshot of packet voice model

AN OVERVIEW OF THE CAPACITY MANAGEMENT PROCESS

Kenneth W. Kolence

Kolence Associates, 3591 Louis Road, Palo Alto, CA 94303

ABSTRACT

This paper provides an overview of the general capacity management processes used to assure reliable on-line response times for large production oriented IBM systems operating under MVS or MVS/XA. The other survey papers in this series focus on the techniques and tools used in these processes, such as analytic modeling, work load management, and software performance engineering. Taken together, this set of papers is an attempt to review the history of capacity management and its current state of the art capabilities. Finally, in this paper, the relationships between the field of capacity management and the academic disciplines of computer science and software engineering are commented on in the context of Ferrari's recent paper, "Considerations on the Insularity of Performance Evaluation".

INTRODUCTION

It is often said that computers have ushered in the Information Age over the last two decades, and this has been reflected in not only what we do, but how we do it. Just as the period we now call the Industrial Revolution was characterized by major changes in the means of organizing business endeavors, so too have computers caused radical changes in the business processes which characterize the modern corporation. While the development of broad corporate data bases ultimately lies at the heart of this new organizational revolution, the outward manifestation is most often in the form of an on-line system. Such systems fit smoothly into the natural pace of work of the business processes only if their response times are both reasonably fast and reasonably consistent.

For major production systems, one of the most consistently vexing problems of on-line systems is how to reliably provide such response times at a reasonable cost. Perhaps because the focus of concern is on production environments, this issue and its means of solution have received relatively little attention in either the academic world or the IEEE and ACM professional literature. However, the problems are of great practical concern, and as such have given rise to a large and vigorous community of professionals dedicated to establishing the relevant theory and practice.

The purpose of this paper is to provide an overview of the history and current state of the art of this field, and to partially address the academic implications of the technologies and viewpoints that have evolved. Other survey papers presented in the Capacity Management sessions at this FJCC Conference consider the major technological issues and provide rich bibliographies of the available literature. (BUZE86, HOWA86, LO86, SMITHB86, SMITHC86)

BACKGROUND AND HISTORY

In this set of papers, the term capacity management has been used to refer to the general collection of processes concerned with assuring on-line response time and batch service time requirements are reliably met at reasonable cost. Both the day-to-day processes and the planning and design processes are included as part of the general capacity management process, since the latter must fully support the former.

The term capacity management was originally intro--duced in the mid-1970s. (KOLE74, KOLE76) This was about the time on-line applications were first being widely placed in use, and slightly before analytic modeling concepts were made available in a generally usable product form. At that time, the major activities were tuning systems to improve utilization and throughput, reducing the run times of batch applications, and equipment planning. The importance of response times, and especially the difficulties in reliably providing required response times, were then only beginning to be fully appreciated. Had the field been named slightly later, a more properly descriptive term, such as service management, would probably have been chosen. Although the tuning activities are still a major aspect of capacity management, the focus is now more on response time and service, along with equipment planning.

Capacity management is almost uniquely identified with large mainframe production environments, and in particular IBM and IBM compatible systems running under the MVS and/or MVS/XA operating systems. Partially this is because these systems are highly multiprogrammed and typically configured with several 308X and/or 309X central processors using a common pool of disks and tapes. Significant cost and throughput benefits accrue when an

CH2345-7/86/0000/0741$01.00 © 1986 IEEE

ɔngoing tuning activity is applied to such an environment. The key reason though is that in modern corporations, these systems usually act as the underlying engines which drive many vital business processes. As such, response time and batch service reliability is of sufficient organizational importance to justify the costs and manpower involved in developing and maintaining service reliability.

As a conservative estimate, well over 10,000 DP professionals in the U.S. are involved in some aspect of capacity management, many if not most on a full-time basis. Worldwide, the number certainly exceeds 15,000.

The growth of capacity management capabilities in this environment was not paced by the equipment vendors. The existence of the potentially large and easily accessible MVS marketplace encouraged the entrepreneurial spirit in many individuals, resulting in a large variety and number of capacity management oriented software products being made available for the MVS environment. Contrariwise, the relatively small and unaccessible marketplaces for other large mainframe manufacturers discouraged independent product development, and much less choice is available to these users. The even smaller budgets for mini and supermini installations, as well as their relative lack of a production orientation, has resulted in an almost complete absence of such products or even capabilities. For example, the UNIX 4.3 system released from a group at U.C. Berkeley contains only one or two of the most basic types of measurement tools. Thus, it is probably quite fair to say that the evolution of the current state of the art in capacity management has been driven by a combination of corporate needs for production response time and service reliability and the entrepreneurial spirit.

Any group of professionals as large as that involved with capacity management will normally be involved in one or more professional societies related to their work. In general, however, these people belong neither to the ACM nor IEEE and have little interest in the broader interests of either group. Because of the homogeneity of interest centered on large IBM operating systems and work loads, the IBM SHARE and GUIDE organizations were more natural loci of interest. Also, in the early 1970s the most universally used products were offered by a company named Boole & Babbage, Inc. A user's group based on these products was founded in 1971. Because the attendees also represented almost all of the users of other products as well, it quickly outgrew the limitations of a single vendor orientation. To reflect this, it was formally incorporated in 1975 as a non-profit organization called the Computer Measurement Group, Inc., or CMG.

At the time of incorporation, CMG member interests were almost solely limited to problems associated with tuning systems and applications, and with charge-back methods. To some extent, this was because available software products were limited to these functions. The more general concepts and methods of capacity management were popularized by a company called the Institute for Software Engineering, Inc., beginning in early 1976. This same company also published a book called "An Introduction to Software Physics: The Meaning of Computer Measurement". (KOLE76, KOLE85) This theory permitted the unification of the many different measurements in use, and postulated the basic structure of the capacity management process to identify where to focus further research in and application of the theory. A variety of courses and publications, and the International Conferences on Computer Capacity Management (ICCCM) were used to inject these concepts into the CMG community.

Almost concurrently with the activities of the Institute, another company called BGS Systems, Inc., was founded to exploit the techniques of analytic modeling in the form of a software product called BEST/1. At that time, modeling was an unproven technique in the eyes of most practitioners, and was initially met with considerable skepticism. However, the combination of an available product to predict response times, the problems of managing on-line systems response times, and the concepts of capacity management quickly resulted in genuine acceptance of both analytic modeling and the capacity management subprocess called capacity planning.

In effect, the development and use of the product provided the empirical proof of the concepts of analytic modeling. This undoubtedly contributed greatly to the relatively widespread academic interest in these techniques.

By about 1980, a relatively balanced emphasis on tuning (or performance management) and capacity planning had emerged at CMG. By this time also, installation management had begun to formally organize capacity planning functions as well as the older performance management activities. Often, this was done at the direct instigation of senior management above the DP group, because of the serious need to provide reliable response times for the major on-line systems used by the corporation.

The planning and tuning activities of capacity management are now generally accepted within the community of large IBM MVS production sites, at least in the U.S. and Europe, and probably all of the other continents as well. This has been reflected in a dramatic growth of the CMG membership and U.S. conference attendance. For example, the 1982 CMG conference attracted 550 people; 950 attended in 1983, growing to 1,200 in 1984 to over 1,600 in 1985. The 1986 conference in December of this year is expected to draw close to 2,000.

In addition to its annual conference, CMG has a large number of Regional Chapters. Typically, these groups meet monthly and present one-day technical programs.

In Europe, a private organization sponsors an annual conference called ECOMA. It is roughly equivalent to a CMG conference. For various reasons, however, local CMG organizations in some of the European countries have begun to form and can be expected to eventually merge with or replace the ECOMA organization. For example, the first U.K. CMG organization was formed in January of 1986. It held its first conference in late May. Over 250 people attended, and the quality of the papers and presentations was excellent. During the same period of time, a less formal meeting of CMG Italy attracted about 100 attendees. Going to the other side of the world, Australia has a very active national CMG with several regional chapters.

The relationships between the various national CMGs is currently quite informal and based mostly on personal relationships. However, the problems to be solved are essentially the same worldwide for everyone who uses large MVS systems. For this reason, one can always expect to see considerable cooperation between the national CMGs, and perhaps some form of international confederation will evolve over the next several years.

Clearly, the vitality of CMG reflects the economic value of the capacity management activities within the sponsoring DP organizations. However, it should also be pointed out that the drive and energy of many individuals have contributed both to the field and to CMG itself. CMG is probably also unique in the degree of support it receives from the vendors of capacity management products, although this is generally in the form of personnel commitments rather than money.

AN OVERVIEW OF CAPACITY MANAGEMENT

The Major Subsystems

The most general definition of capacity management is that it is the collection of processes by which the response time and batch service requirements of applications software are consistently and reliably provided, at all times and at a reasonable cost. In practice, capacity management breaks into two major subprocesses. The first, often called performance management, is concerned with those processes which use existing configuration capacity with existing systems and applications software, and provide real response times and real batch service to real transactions at some load level. That is, performance management operates in the current time frame and is charged with meeting user service requirements with the means at hand. To emphasize it is a major subprocess of capacity management, performance management is often referred to in this paper as the Current Time Frame (CTF) capacity management subsystem.

The second major subprocess is typically called capacity planning. Because the term implies the process is limited to equipment planning, it is considered by the author to be a serious misnomer. More generally, the objective of this second major process is to assure the means required by the performance management process to provide actual service within requirements are in fact delivered at the appropriate points in the future time frame. For this reason and others, this second major subprocess is referred to in this paper as the Future Time Frame (FTF) capacity management subsystem. It should be understood that the FTF subsystem fully includes capacity planning as the term is currently understood.

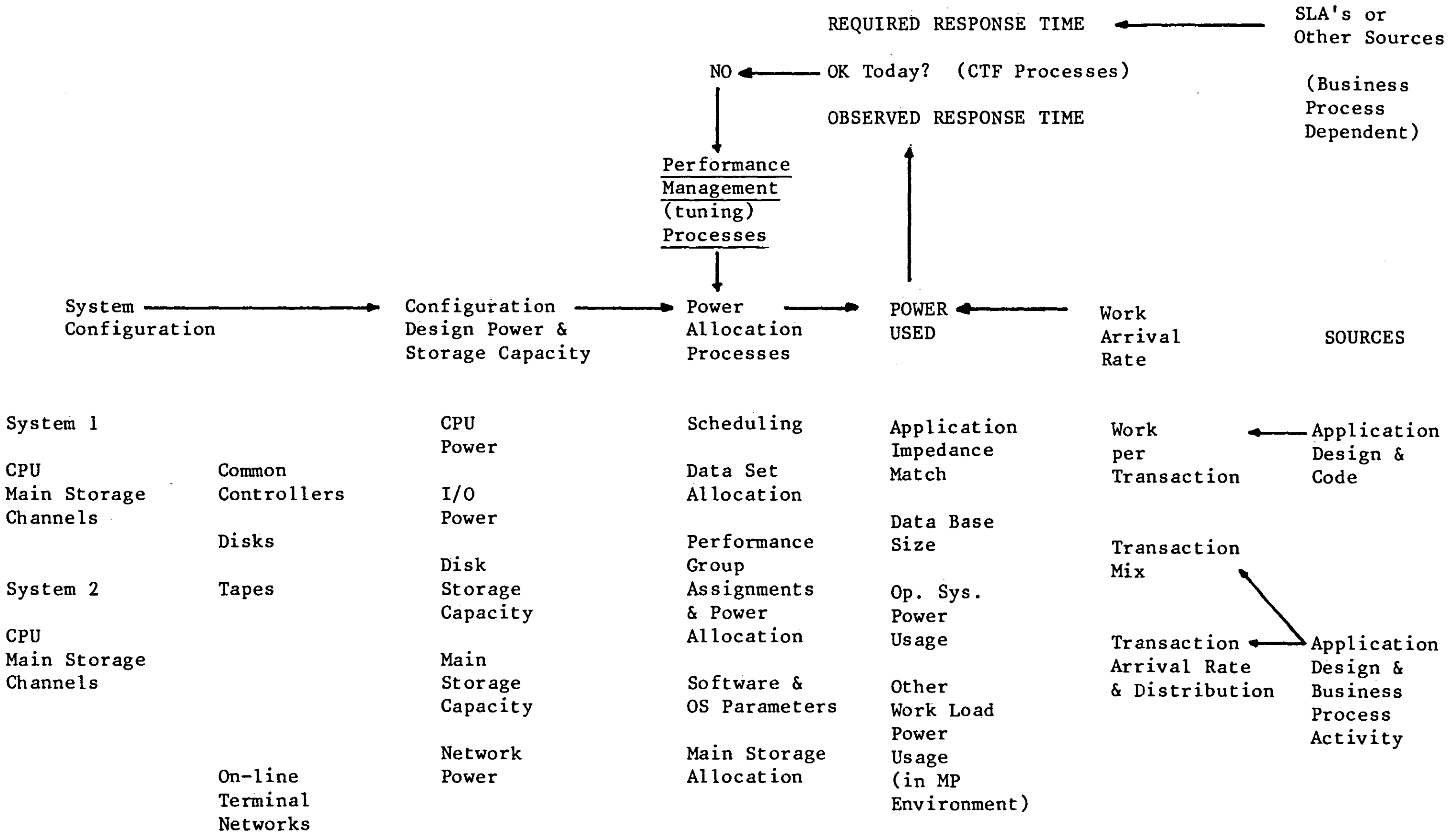
Since physically real configurations, work loads, and service times only exist in the current time frame, the FTF subsystem must develop and use representations of reality to determine what will be required, and then must obviously implement the solutions proposed. As a result, much of the FTF subsystem is concerned with the processes of work load classification, work load characterization, transaction arrival rate and batch volume projection, service and response time projection, and equipment and configuration design. Analytic modeling software products are widely used in this subprocess for the equipment (capacity) planning subfunction.

The emerging discipline called Software Performance Engineering is part of the larger FTF process, and as such is a major reason for discarding the term capacity planning to represent the more comprehensive FTF process. Software performance engineering is concerned with the problem of assuring new systems and applications software have inherently good quantitative properties, both service and utilization/cost. While less well recognized, it is also concerned that such software properly interfaces to the other capacity management processes. Although still in a very early stage, Software Performance Engineering will probably become the major bridge bringing together the capacity management and the computer science and software engineering academic communities.

The responsibility of the FTF capacity management process is to deliver what is needed by the CTF process to provide actual response and service times which meet the required levels. Thus, it is useful to consider the characteristics of a typical MVS installation, and the real processes which result in the observed service characteristics. Figure 1 is a simplified schematic of these characteristics and the basic processes at work. Since the word capacity figures prominently in this diagram, the software physics interpretation of the term will be discussed first.

Capacity and Power

The term capacity generally can refer to either the quantity of available storage (disk, main storage, etc.) or to processing power. The most general definition of processor power is simply bytes per second transferred to a storage medium (including CPU registers), taking into account the full execution time of the processor. (KOLE76, KOLE85) It has been shown in general that processor power defined in this manner is a function of a variety of software and resource allocation parameters in addition to the raw hardware speeds. Thus, the processor powers available from a system depend partially, and sometimes largely, on how these parameters are set.



A Software Physics Analysis of CTF Response Time
Characteristics and Processes

Figure 1

The most commonly used approximations to power are MIPS (Millions of Instructions executed Per Second) for central processors, I/O rates (Number of I/O's per second) for DASD (disk) and tape subsystems, and messages per second for networks. Since power is generally defined as work per unit time, the corresponding units of work in this system of measures are Millions of Instructions Executed (MIE's), Number of I/O's, and Number of network messages. However, on any given CPU it is more normal to measure and refer to CPU work in terms of CPU seconds.

In general, the amount of work required by a unit of systems or applications software to process a given quantity and mix of data is invariant. (However, the work may be quite sensitive to changes in the quantity and/or mix of data.) In particular, the average work per transaction in on-line systems, and the transaction mix in conjunction with the transaction arrival rate, defines the processor powers required to provide any given level of service and response time.

The work per transaction (or per record in batch systems) is a vector quantity, and the ratios of work required by different processors (e.g., CPUs and DASD systems) are fixed for given data. As a result, the application software can only use available capacity in proportion to the work vector requirements. For example, assume one million instructions and 50 I/O's are required to process a transaction. If a ten MIPS CPU is being used, and the I/O system and data set allocation is such that the application can only get 50 I/O's per second, then the application can only use one-tenth of a CPU second per elapsed time second. And, the minimum obtainable response time for the transaction will be close to one second plus the network service time.

In large MVS and MVS/XA systems one typically finds more than one central processor (308X or 309X) with a common pool or network of DASD and tapes. Within the I/O network, strings of drives are attached to controllers or storage directors, and these in turn are attached to one or more channels on one or more central processors. The major reasons for this common I/O pool are reliability and storage accessibility. That is, if a CPU or channel or controller fails, another I/O path and/or CPU can be used to continue processing of critical systems.

The exact manner of connecting I/O devices through controllers and CPU channels is the design of the I/O subconfiguration. In conjunction with the physical drive speeds and storage characteristics, the configuration defines the I/O processing power available overall, and to each of the central processors. In large MVS systems, the overall I/O design capacity is usually considerably less than the sum of the I/O capacities available to each CPU. In Figure 1 the capacity identified as configuration design capacity includes total CPU, I/O, and network power as well as main and secondary storage capacity.

The allocation of design capacity occurs in several ways. First, the scheduling of applications onto various CPUs results in the allocation of each system's total MIPS power and main storage among temporally concurrent applications. In an operational mode, MVS further dynamically allocates CPU power and main storage according to "performance groups and classes", to which each of the applications is assigned.

An important form of power allocation is done by the DASD space allocation process. There are only a limited number of independent or non-interfering I/O paths within the DASD channel-controller-drive network. If data sets from temporally concurrent applications (and the operating system) are allocated space on the same path, interference can occur and DASD service times increased and made more variable. In effect, the I/O power which the application can draw from the design capacity is often significantly reduced. Within a single application, the degree of I/O concurrency between data sets on the same path and other space allocation factors such as head movement and block size can further reduce the actual ability of the application to draw I/O power from the system.

The work vectors for applications require CPU, I/O, and network power be drawn from the system in fixed proportions. In many ways, the problems of allocating processor power to an application correspond to electrical impedance load matching problems. In both instances, mismatching leads to a situation in which less than the design power of a system is usable. In the capacity management context, this means increased unit costs and increased service and response times. It can also mean wide variability in response times when the mismatch occurs as a result of interapplication interference. This is a particularly undesirable situation in most if not all on-line production systems.

The Performance Management Subsystem

The performance management function is usually performed within the computer operations group, often in the group called technical services. They are essentially overconstrained in terms of the means available to them to assure required service levels are consistently provided. At any given time, the equipment and thus the design capacity they use is fixed, the work per transaction is an inherent property of the application, and the transaction arrival rates are determined by the business processes being supported.

Performance analysts are thus limited in capability to adjusting power allocations, and to improving the "impedance match" through data set allocation and placement and by interapplication data set isolation techniques. In addition, they can also improve the available power of the system by adjusting various operating system and application system parameters, particularly those involving main storage management and paging activities.

In general, such activities are referred to as "tuning the system", and indeed performance management is basically limited technically to improving and maintaining the ability of the overall system to deliver as much of its design capacity as possible to the applications.

It should also be mentioned that a considerable amount of effort by performance management personnel is spent in monitoring on-line response times and analyzing problems to determine their causes and possible solutions. Because of the high degree of multi-programming found in MVS systems, along with huge transaction volumes, this is often a very difficult task. It is also not easy to determine the most effective action to take, or indeed even if a given action will completely solve a given problem. As a result, there have been recent proposals that analytic modeling software products be developed and used in conjunction with direct measurements to detect problem causes and quantify potential solution options. In fact, early results have been quite encouraging and a rapid evolution of this new application of analytic models can be expected.

In addition to response times, the performance management group monitors utilizations, work load volumes, and so forth. Various management reports and historical trends are prepared and maintained. Much of the input to the capacity planning function is thus provided by the performance management activity. Typically, these two groups are separate. The day-to-day orientation of performance management and the longer range viewpoint of capacity planners have not been found to mix well organizationally.

Future Time Frame Subsystems

Figure 2 is a schematic of the FTF capacity management subsystem corresponding to the structure of Figure 1.

Service Level Agreements. All of the capacity planning activities clearly require a quantitative definition of what "required response time and batch services times" are. In practice, only certain critical applications (normally, just the on-line systems, but some batch systems are also critical) have formal response time requirements set for them. These are typically stated in the form of Service Level Agreements, or SLAs. These are agreements between the Operations and user departments that a minimum response time will be guaranteed some percentage (typically 95%) of the time.

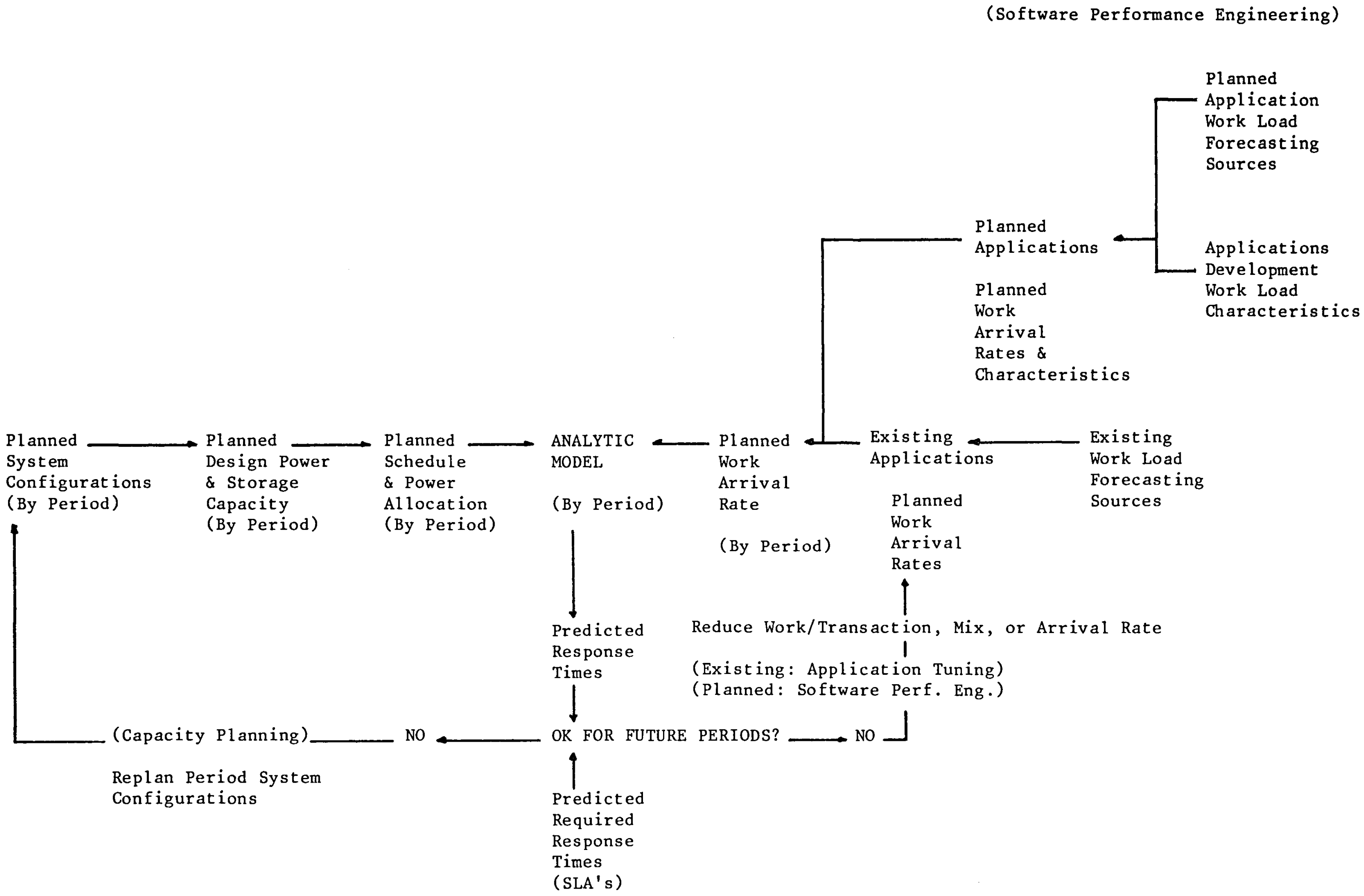
The guarantees in SLAs are generally difficult to fulfill, and may be even more difficult to negotiate. As a result, formal Service Level Agreements and the associated "Service Level Management" systems are far from universal. In the less sophisticated large MVS installations, the problem of establishing required response times for capacity planning purposes is sidestepped by Operations setting response time objectives without formal user consultations. Still, if this is recognized as a primitive form of service level management, then it is reasonable to say that all MVS installations require a service level management activity.

The Capacity Planning Subsystem. The performance management system must match the available capacity to the actual work load to provide the required response and service times. The FTF (Future Time Frame) capacity management subsystem therefore is, among other things, charged with the responsibility of assuring the required capacity is available when needed and in fact can be allocated as required.

To do so requires several things. One is a set of work load projections grouped into applications and other work which correspond to the scheduling of the various physical configurations in the installation; i.e., IMS, TSO program development, production batch, etc. Typically, only peak loads are projected and so a means for identifying and projecting the peaks of various applications is also required. Another requirement is a means of projecting the response and service times to be expected at these peak loads using the existing configurations and also using alternative equipment and configurations. The activities associated with developing these forecasts and determining when and what equipment changes are required are generally called the capacity planning function within a typical MVS installation.

The papers by Buzen, Lo, and Brian Smith review the evolution and current state of the art of the major capacity planning activities (BUZE86, LO86, SMITHB86). While analytic modeling techniques and products have been a boon to capacity planning practice, these papers demonstrate that current capabilities are not yet all that are needed. In particular, a more direct linkage between models and the various power allocation processes would be useful to provide the performance management process with directions on how best to accomplish this in the current time frame.

Software Performance Engineering. The importance of reliable and consistent response times to the successful operation of the basic corporate business is being increasingly recognized at senior non-DP management levels. Considerable pressure is thus being brought to bear for the use of formal SLAs. The importance of the performance management function, which is also usually charged with the resultant day-to-day service management responsibility, is increasing in response to this pressure. Similarly, the importance of the capacity planning function, and especially work load forecasting, is also increasing. However, taken together, both functions are limited in terms of their technical methods of assuring proper response times to simply providing more computing power, more effectively. The other two determining technical factors, work per transaction and transaction arrival rates (in some mix), are currently left largely uncontrolled. This failing probably represents the most significant unresolved area in the field of capacity management.


(Software Performance Engineering)
Planned Application Work Load Forecasting Sources
Planned Applications
Applications Development Work Load Characteristics
Planned Work Arrival Rates & Characteristics
Planned System Configurations (By Period)
Planned Design Power & Storage Capacity (By Period)
Planned Schedule & Power Allocation (By Period)
ANALYTIC MODEL (By Period)
Planned Work Arrival Rate (By Period)
Existing Applications
Existing Work Load Forecasting Sources
Planned Work Arrival Rates
Predicted Response Times
Reduce Work/Transaction, Mix, or Arrival Rate
(Existing: Application Tuning)
(Planned: Software Perf. Eng.)
(Capacity Planning)
NO
OK FOR FUTURE PERIODS?
NO
Replan Period System Configurations
Predicted Required Response Times (SLA's)

A Software Physics Analysis of the FTF Capacity Management Subsystem

Figure 2

Ultimately, the transaction arrival rate characteristics of an on-line application are determined by the level of activity of the corporate business processes being supported. One is thus tempted to say that arrival rates are in fact inherently uncontrollable, and move on to the question of how to adequately project arrival rates given business plans. Certainly this is the basis of one of the major technical thrusts in work load forecasting methodology. However, the arrival rate of logical, as opposed to physical, transactions is a function of the application design, which in turn is a function of the basic business process design. In other words, the actual transaction arrival rate required to support a given level of corporate activity is determined during the design process. Obviously, this is similarly true of the third independent variable determining response time, the work per transaction.

The field of software performance engineering is concerned with assuring that applications and systems software have the quantitative and functional attributes required to meet required response and batch service times, at reasonable cost. It is therefore a process of basic importance within the overall capacity management process, with interfaces to all of the other capacity management subprocesses; e.g., performance and service management, work load forecasting, and capacity planning.

Software performance engineering currently has only a tiny number of proponents and practitioners compared to the other capacity management functions. A major hurdle to the growth of the field is that the practitioners really need to be the applications development and maintenance people within a company, and historically such groups have been outside of the mainstream of capacity management.

An oft-repeated development philosophy is to develop first and then "tune" if necessary. Painful experience has proven this approach is not only grossly wasteful of computing capacity, but much worse, is operationally unacceptable in the context of any company whose business is paced by its computer systems. The basic initial hurdles faced by software performance engineering are thus twofold: (1) to understand the technical methods of designing applications with the desired quantitative and functional capacity management properties, and (2) to encourage the adoption and use of such methods within the application life cycle.

Today, the basic focus of software performance engineering research is on developing an understanding of the technical principles which should be applied in order to reduce the work per transaction characteristics of applications. As with the other capacity management processes, these efforts are proceeding in the classical scientific way by combining empirical efforts with a search for underlying patterns and principles. Good progress appears to have been made in understanding what these principles may be. (SMITHC/UP) However, they can be difficult to apply in the context of applications development systems developed in ignorance of these principles.

For example, information hiding and data base navigation are typical properties of the so-called fourth generation data bases, languages, and systems (such as FOCUS and IBM's DB2). The functional generality of these properties come at a high cost in terms of work per transaction, and in some instances this work may be highly variable. Neither the means to understand a design decision in these terms nor methods for controlling the work per transaction are easily available to the designer, using such systems. So, the problem of encouraging the adoption and use of technical methods of software performance engineering may be more formidable than developing the technical aspects of the discipline.

There are other major obstacles to extending the concepts and techniques of software performance engineering into the applications design and maintenance processes, at least relative to the problem of controlling the response time and work per transaction characteristics. Essentially, they all center on two facts: (1) qualified development personnel are a scarce and expensive resource, and (2) a large backlog of applications requirements exist in a typical MVS installation. While software performance engineering can easily be shown to be highly cost-effective over the life of an application, it is perceived as requiring more personnel and more development time. In many if not most development efforts, especially where considerable tuning is required, this is not true. Nonetheless, this misperception hampers the acceptance of the process.

It should be noted that the other aspects of software performance engineering, such as those concerned with interfacing applications directly to the other capacity management functions, have not been explored in any significant manner. It would appear they could be solved in some form amenable to productization. The history of the growth of capacity management functions corresponds closely with the general availability of product.ı to support functions. The likelihood then is that once such products become available, these other aspects will quickly become important.

Another area of software performance engineering that would benefit from the availability of additional software product tools is applications tuning. In the earliest days (circa 1970), such tools were able to lead quickly and easily to quite significant improvements in most batch applications. However, these tools have not been extended and adapted to current software as well as could have been hoped. Properly upgraded capabilities of this type would be of direct value to the performance management area. They would also identify coding methods and styles which have pos:tive and negative impacts on response time and service. This in turn would lead to enhanced awareness of the value of software performance engineering in the applications development area.

ON THE ISSUE OF INSULARITY

The presentation of this collection of survey papers at FJCC '86 is to a large extent a direct response to, and in support of, Ferrari's recent paper entitled "Considerations on the Insularity of Performance Evaluation". (FERRA86) In it Ferrari observes that the topics of performance evaluation (here called capacity management) have primarily evolved outside of the academic disciplines of computer science and software engineering. He also points out that in general there is a great lack of academic interest in the quantitative concepts and methods exemplified by the field of capacity management. He then argues this is no longer an acceptable state of affairs, both because scientific and engineering disciplines are classically quantitative, and because the quantitative properties of computer-based systems are of great concern in all major projects. Ferrari also fully recognizes that such quantitative concepts and methods cannot be treated as a separate discipline. Rather, they need to be permeated into the very fabric of both computer science and software engineering. Finally, Ferrari suggests several areas of both research and action to assist and support the process of fully integrating capacity management with the academic disciplines.

Ferrari's paper closes with the statement that a small group of people have kept ". . . the esoteric cult of performance evaluation . . . alive but relatively secret for twenty years." While true with regard to the academic community, it is untrue in the context of the community of capacity management professionals. Thus, an anomalous and, as Ferrari argues, perhaps unique situation has developed. A strong and vigorous profession exists, it is deeply wedded to quantitativeness and scientific empiricism, and it is without representation or support in the corresponding academic disciplines.

It would perhaps be easy to argue that this condition has arisen due to the "insularity" of either or both groups, and to a certain extent these arguments would carry a ring of truth. Still, the existence of the thriving professional society, CMG, with a rich technical literature certainly implies openness. Beyond that, CMG's highest award, the A. A. Michelson Award, celebrates the famous experimental physicist and thereby sets his accomplishments in science as a model for the capacity management professional. Moreover, the 1985 award winner was Dr. K. M. Chandy, an academician who, in his acceptance speech, spoke on the same basic issues as Ferrari. So the academic community is in fact represented in and recognized by the CMG community. What then are some of the problems?

The focus of attention of capacity management has been and is large IBM systems under MVS, and such systems are not easily learned nor accessed by academicians. Second, the major topics have been systems tuning, capacity planning, and other production oriented concerns. Again, these are not areas academicians would normally become involved in. Products, often specialized to MVS and MVS application systems, play a major role in the basic activities of a capacity management professional. Although a few academics have been involved in the design of such products, products are not generally transportable to the systems, such as UNIX, which are favored in the universities. Even if they were, the production orientation of most of them would render them useless and uninteresting. Finally, as Ferrari points out, art is fun, but scientific empiricism is often hard, tedious work. The game must be worth the candle, and little in the way of financial, intellectual, or "fashionable" support or reward is available to academicians who venture into this area.

In the author's opinion, part of the solution to dissolving the insularity Ferrari observes is to recognize that the types of problems of interest to the academician and to the practicing capacity management professional only appear to be quite different. There are many instances where great commonality of interest exists. For example, as Ferrari again correctly argues, practice would benefit greatly from improved theory. In the author's opinion, the direction and form of the required theory must relate to the needs of practice as, for example, the general problem of quantitative work load characterization. Such work clearly would, and undoubtedly will, lie at the heart of a discipline of software performance engineering. While other examples can easily be given, perhaps it is simplest to say that a part of the solution to bringing the practical and the academic disciplines together is to recognize where and in what manner their interests overlap.

To be turned into reality, commonality of interest needs to be turned into a commonality of importance and support. This part of the solution involves many things, but the most important is simply to develop proper funding sources and intellectual acceptance and support within the academic community. Opening much broader and meaningful lines of communication between academics and practitioners, perhaps initially through professional society activities, would be important, too. But of the most basic importance, it will take people to make all of these things happen. It is to be hoped that Ferrari's paper, and this set of FJCC papers, will encourage the necessary personal commitments in both communities.

REFERENCES

A. Conference proceeding references in this paper and other papers in the FJCC Capacity Management sessions may be obtained from:

(1) CMG Proceedings:

The Computer Measurement Group, Inc.
6397 Little River Turnpike
Alexandria, VA 22312
(703) 354-3306

(2) ICCCM and ICIM Proceedings:

Institute for Information Management, Inc.
The Pruneyard Tower, Suite 230
1901 South Bascom Avenue
Campbell, CA 95008
(408) 559-6911

B. The other papers in the FJCC '86 Capacity Management sessions are:

BUZE86 Buzen, J. P., "An Overview of Performance Prediction in MVS Systems and SNA Networks"

HOWA86 Howard, P. C., "The Evolving Role of Software Products in Capacity Management"

L086 Lo, T. L., "The Evolution of Workload Management in the Data Processing Industry: A Survey"

SMITHB86 Smith, B. J., "A Survey of the State of the Art and Practice in I/O Subsystem Modeling and Analysis"

SMITHC86 Smith, C. U., "The Evolution of Software Performance Engineering: A Survey"

(Refer to these papers for extensive bibliographies on the corresponding subjects.)

C. References in this paper, other than those above, are:

FERRA86 Ferrari, D., "Considerations on the Insularity of Performance Evaluation", IEEE Transactions on Software Engineering, June 1986.

KOLE74 Kolence, K. W., "Computer Planning and Control", 1974 International Systems Meeting, Association for Systems Management.

KOLE76 Kolence, K. W., "An Introduction to Software Physics: The Meaning of Computer Measurement", Institute for Software Engineering (now the Institute for Information Management), Campbell, CA, 1976.

KOLE85 Kolence, K. W., "An Introduction to Software Physics", McGraw Hill, New York, 1985.

SMITHC/UP Smith, C. U., "Applying Synthesis Principles to Create Responsive Software Systems" (unpublished as of 7/1/86).

AN OVERVIEW OF PERFORMANCE PREDICTION IN MVS SYSTEMS AND SNA NETWORKS

Dr. Jeffrey P. Buzen

BGS Systems, Inc.

ABSTRACT

Analytic models based on the theory of queueing networks are used extensively in capacity planning applications involving mainframe computer systems. The mathematical theory which provides the backbone for such models has been extended recently to deal with SNA based telecommunications networks. This paper surveys the use of analytic models in both host and network capacity planning. Some of the important differences between analytic queueing models for host systems and telecommunications networks are then discussed. Finally, the paper reviews the operation of relevant SNA mechanisms including polling, half and full duplex protocols, slowdown mode, and the parameters MAXIN, MAXOUT and PASSLIM.

INTRODUCTION

The problems caused by an overloaded data center are all too familiar to many users. As workloads grow, response times for online applications can increase dramatically, creating numerous difficulties. These include long lines at automated banking machines, sluggish response for order entry and customer information systems, and reduced productivity among programmers, data entry clerks and word processing personnel.

The process of planning hardware upgrades to avoid such overloads is generally referred to as capacity planning. Although it is an essential function, capacity planning is not an easy task. One problem is that the processing demands on most data centers are continually varying and shifting. In addition, evaluating alternative upgrade strategies has become substantially more difficult as systems have increased in complexity. For example, it is not easy to determine whether a faster CPU, more processor storage, or an additional telecommunications line represents the most cost-effective solution to an overload that is expected to arise three months in the future.

Despite these difficulties, every data center manager does -- in fact -- employ some method for dealing with the capacity planning process. The simplest but least satisfactory approach is to be purely reactive: that is, to wait until a serious problem arises and then react by ordering additional hardware. This creates obvious problems such as disrupting normal business operations and frustrating both external customers and internal users. In addition, management must still address the important technical problem of deciding exactly what hardware is required to resolve an existing overload condition. The simplest approach here is to rely purely on intuition and experience. However, as already noted, it is difficult for human intuition to keep pace with increasing system complexity. As a result, relying on intuition can become a costly trial and error process that generates overexpenditures in certain areas as well as further delays in restoring satisfactory performance.

One obvious way to improve the capacity planning process is to employ predictive techniques that can indicate: 1) how much longer a system can operate before an upgrade is required; and 2) what the most cost-effective upgrade will be at that time. Since response time is the primary indicator of system performance in most installations, predictive techniques employed for capacity planning must provide information about response time. Queueing theory is a branch of mathematics that deals directly with the prediction of response times, and as a result queueing models have become important tools for capacity planning. Before discussing queueing models in more detail, it is useful to present an overview of the major response time components in MVS/SNA systems.

MANAGING RESPONSE TIMES IN HOSTS AND NETWORKS

As shown in Figure 1, the response time for a typical transaction can be divided into three major components. The first,

CH2345-7/86/0000/0751$01.00 © 1986 IEEE

inbound network response time, is the length of time needed for the user's transaction to travel from a remote terminal to the host computer. Host response time, the second component, is the time required to process the transaction at the host. Finally, outbound network response time is the time necessary to transmit the transaction results back to the end user.

In many organizations, responsibility for maintaining acceptable network response time (inbound and outbound) is assigned to the manager of the telecommunications department. The data processing manager is then responsible for maintaining acceptable host response time. Conceptually, this separation of responsibility is appealing because of the clean hardware interface and reasonably clean software interface between the host computer and the network. Furthermore, the substantial quantity of specialized information needed to manage the host computer and the communications network can be separated under this scheme, considerably reducing the complexity of the overall management problem.

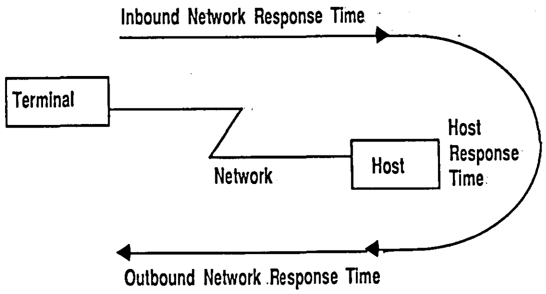


Figure 1: The Components of On-Line Response Time

Despite the advantages of dividing the responsibility for host and network response time, it must be remembered that the end user of an online application sees no such separation. The user is concerned only with total response time (inbound plus host plus outbound) and is satisfied only if the sum of the three components meets expectations.

Traditionally, total response time is controlled by establishing formal or informal "service contracts" between the host data processing department and the network management department. For example, the host data processing department might commit to providing a two second host response time for a particular class of order entry transactions. Based on the assumption that this will be true, the network management department can then configure its telecommunications lines and network processors to meet the total response time requirements of the remote offices that it serves. At the same time, the host data processing department can plan its hardware and software upgrades to meet the two-second requirement as the total load increases. This host oriented planning can be done without concern for network response time.

Although there are several advantages to this division of responsibility, there are certain disadvantages as well. Specifically, without an overall view of host/network interaction, time and effort can be wasted trying to optimize the wrong component. For example, rather than adding higher speed lines to handle increased traffic from a distant city, a more economical approach may be to reduce host response time by increasing the speed of the paging subsystem. Such trade-offs are difficult to see when the responsibilities for telecommunications and host capacity planning are assigned to separate organizations.

Another problem with dividing the responsibility is that certain interactions between host and network work loads may be hidden. For example, as the number of remote users increases, the network traffic and host processing loads increase, and there may be feedback effects resulting from the interaction and interdependencies of the network and host system. These effects can be difficult to account for without direct links between the models used for host and network capacity planning.

A further consequence of the traditional approach to capacity planning is that advances in the analysis techniques of one area may not be fully communicated to the other. The evolution of performance modeling tools during the past five years is a case in point.

PERFORMANCE MODELING

A performance model allows the capacity planner to assess how performance will be affected by changes in a system's external workload and internal hardware and software components. For example, a model can be used to predict the effect on response time if the transaction volume from a remote site increases by 75 percent and a faster host CPU or telecommunications line is installed.

Figure 2 illustrates a very simple queueing model known as the single server queue. It is assumed in this model that a stream of "customers" are arriving at the "server", which is represented by the circle in Figure 2. Each customer requires a certain amount of processing or "service time" at the server. If a customer arrives at a time when the server is idle, the customer receives service immediately. Otherwise, the customer joins the queue of waiting requests, which is represented by the rectangle in Figure 2. When the server completes processing for an individual customer, the queue is examined; if there are customers waiting at this point, one of them is selected and is immediately placed in service.

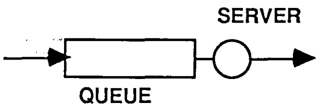


Figure 2
Single Server Queue

One of the most important quantities that analysts consider when analyzing a single server queue (or almost any queueing model) is the response time of a typical customer. Response time includes both the time spent by a customer while waiting in the queue and the time spent receiving service. In general, if server utilization is low (i.e., if the server is idle much of the time), the majority of arriving customers will not have to wait at all in the queue. Thus, average response time should be approximately equal to service time. As server utilization increases, it becomes more likely that arriving customers will encounter a busy server. This will increase queueing delays, which in turn will increase response time. Figure 3 illustrates the way response time typically varies as the utilization of the server increases from 0% to 100%.

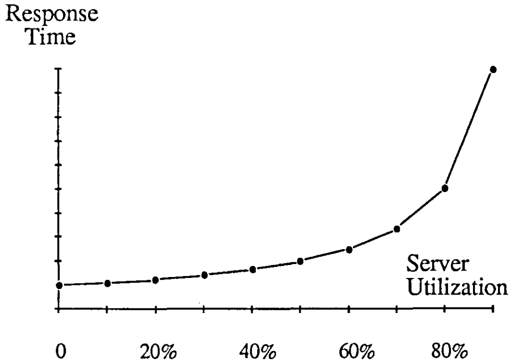


Figure 3
Response Time Function

Note that the curve in Figure 3 has a steadily increasing slope. Thus, the increase in response time that occurs when server utilization changes from 30% to 40% will always be less than the corresponding increase for a 40% to 50% change in utilization, and this in turn will be less than the corresponding increase for a 50% to 60% change in utilization.

Many performance analysts try to make predications of future response time by generalizing upon historical observations of the relationship between previous changes in utilization and their associated effects on response time. For example, an analyst might note that last year utilization increased by 10%, and response time increased by 15%. If another 10% increase in utilization is anticipated next year, the analyst might conclude that there will be another 15% rise in response time as well.

The steadily increasing slope of the response time function implies that this intuitively plausible procedure will almost always underestimate actual response time. The errors that arise when applying simple intuitive reasoning to response time prediction provide one of the principal motivations for the development of queueing theory. The importance of obtaining accurate response time estimates explains why queueing theory is now such an indispensible tool for capacity planners and computer performance analysts.

MODELS FOR MAINFRAME COMPUTER SYSTEMS

When developing a model of a mainframe computer system, the single server queue illustrated in Figure 2 can be regarded as a basic building block which is used many times in the overall model. For example, to model the CPU in a uniprocessor (UP) system, analysts generally use a single server queue in which the "server" represents the CPU itself, the "customers" represent programs that have instructions ready to execute, and "service time" corresponds to the amount of time it takes the CPU to execute these instructions for each program.

Similarly, to model an I/O device such as a disk or drum, an analyst would use another single server queue in which the "server" represents the I/O device, the "customers" represent programs that have I/O operations (reads or writes) ready to be initiated, and "service time" corresponds to the amount of time it takes the device to perform the I/O operation.

In typical computer systems, an arriving program begins by requesting a burst of processing at the CPU. The initial CPU processing request completes when the program generates an I/O request such as a read operation to a data file or a page-in request to a page data set. At this point, the program proceeds to the appropriate I/O device where it will wait in a queue (if necessary) and then perform I/O processing. Once the I/O transfer is complete, the program returns to the CPU queue and begins the next CPU-I/O cycle.

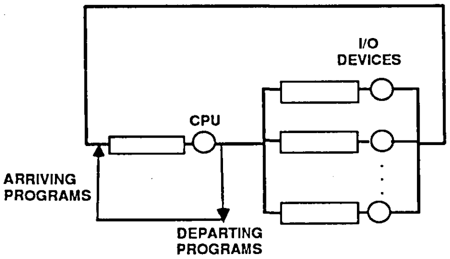


Figure 4
Central Server Model

Figure 4 illustrates the overall flow of programs (customers) in this model. Since the model contains a network of interconnected queues and servers, it is referred to generically as a queueing network model. In particular, Figure 4 presents an example of the "central server" queueing network model [9,10], which has proven extremely useful in analyzing mainframe computer systems. It should be noted, of course, that in order to provide satisfactory analyses of complex MVS systems, the basic central server model in Figure 4 must be extended in a number of ways. The next section of this paper surveys some of these extensions.

MVS MODELING CONSIDERATIONS

One important extension to the central server illustrated in Figure 4 is the representation of multiple customer classes (multiple workloads). The discussion in the preceding section implicitly assumed there is a single class of customers (programs) circulating through the network. This class is represented by a set of parameters that represent the average processing demands that programs make at each server as they circulate. Suppose, however, there are two or more workloads being processed by an MVS system (e.g., batch, TSO, IMS, CICS). In this case a single workload model could be used to approximate performance, but the set of parameters that represent the processing demands at each server would be averaged across all workloads. In addition, predicted response times and throughput rates could only be provided for the single (average) workload. This is clearly unacceptable in cases where separate performance objectives are defined for batch, TSO, IMS, CICS, etc.

To provide this additional detail, the single class queueing network solutions developed by Jackson [22,24] were extended by Baskett, Chandy, Muntz and Palacios [5] to deal with multi-class (multiple workload) networks. Shortly thereafter, the computational algorithms originally developed by Buzen [9,10] for single class models were extended by Reiser and Kobayashi [28] to handle multi-class models. The development of alternative algorithms that provide improved numerical stability in certain cases or improved speed through the use of various approximations continues to be an active area of research [2, 3, 15, 17, 25, 29, 30, 31, 32]. Additional papers on this subject will appear soon in the *Journal of the ACM* and the *ACM Transactions on Computer Systems*.

Another important extension needed for realistic models of MVS systems is an analysis of the delays programs experience before entering memory. Note that the "memory queue" is different from the queue at a CPU or I/O device since memory is not an active server that processes a program for a short burst and then allows the program to proceed to another server. Instead, memory is a "passive resource" that a program must acquire before proceeding to the active job mix and being allowed to compete with other jobs for CPU's, I/O devices, and other conventional servers.

Memory queueing is beyond the scope of the exact solutions developed in [24] and [5]. However, a number of approximations have been developed based on the idea of decomposition. Brandwajn [7] presented one of the earliest analysis of this type. Work by Courtois [18] and Chandy, Herzog and Woo [14] clarified some of the theoretical assumptions involved, and subsequent papers by Menasce [27] and Lazowska and Zahorjan [26] proposed additional approximations to deal with multi-class extensions.

Other problems that must be addressed to develop realistic models of MVS systems include the modeling of preemptive priority scheduling at CPU's and the modeling contention within the I/O subsystem for channels, control units and heads of string. In general, these problems have been addressed by analyzing individual servers or sets of servers in these solutions into complex network models via transformational techniques [1]. The use of operational analysis [11, 19] has facilitated the development of some of these approximations and also provided new theoretical insights into why queueing models work well in practice [33].

It is important to note that the end user of an MVS modeling tool is insulated from these mathematical details in the same way that a programmer using a high level language is insulated from machine language instructions. Thus, the literature on the use of analytic models in capacity planning has shifted from early papers stressing basic feasibility [12] to more recent descriptions of how the models are used to solve everyday problems [6,16,20,21,23].

MODELS FOR TELECOMMUNICATIONS NETWORKS

The basic objectives of a queueing model for a telecommunications network are to predict the response time for various classes of messages: in particular, for "inbound" messages which originate at user terminals and are destined for the host computer system, and for "outbound" messages which originate at the host computer system and are destined for a user terminal. This problem would appear to be well suited for a queueing theoretic analysis since it involves response time prediction for customers that correspond naturally to inbound and outbound messages.

Having identified the customers, the next step is to specify the servers, the service times, and the sources of queueing delay. Consider a very simple case in which a set of user terminals communicate with a host system over a single transmission line as shown in Figure 5.

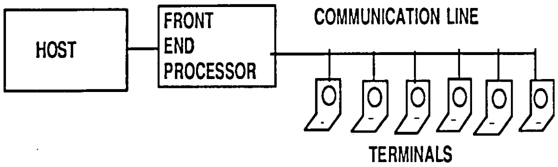


Figure 5
Simplified Layout of a Communication Line

Assume that only one message can be in transmission on the line at any time. In this case the communication line corresponds to the server in a single server queue, messages correspond to customers, and the service time for a customer corresponds to the time it takes to transmit the associated message from the terminal to the host or vice versa.

In this simple case, service time is a reasonably straightforward function of total message length (both data and overhead characters) and the bandwidth of the line (the number of characters per second the line can transmit). Essentially, service time in this case is equal to message length divided by bandwidth.

In reality, queueing models of communication lines are seldom this simple. An important complication arises because of the nature of inbound messages. These messages are prepared at terminals, where they must wait until the line becomes available for the next transmission. Since these messages are not all in one centralized location, it is difficult to arrange things so the "next" message begins transmission as soon as the transmission of the current message completes. In fact, it even takes some special synchronization mechanisms to prevent one message from attempting to use the line at a time when another message is already in transmission.

Note that these synchronization and coordination problems do not arise in the central server queueing model shown in Figure 4. This is because each queue in the central server model contains only local (rather than remote) customers. Thus it is easy for the software that manages each queue to determine exactly when one customer completes a service request, and to dispatch the next customer waiting in the queue without significant delay.

In contrast, the queue of inbound messages waiting for a communication line contains a geographically distributed set of remote customers. Any analysis of such a queue must account for the significant delays that occur between the completion of one service request and the initiation of the next. Because they include queues with remote customers, the mathematical models developed for telecommunications networks differ in important respects from the central server models used to analyze mainframe computer systems.

Systems Network Architecture, or SNA, is the hardware and software architecture employed in most IBM communications networks. Because it must support different types of communications hardware, SNA supports several alternative families of algorithms for managing message traffic between remote terminals and host computers. The next few sections of this paper discuss some of the more important algorithms and architectural concepts incorporated into SNA. The discussion will consider basic functionality as well as implications for performance modeling.

EFFECT OF POLLING

A number of techniques exist for controlling a transmission line to prevent attempts to transmit two or more messages at the same time, and to allow the "next" queued message to begin transmission once the line becomes free. The technique selected for use in SNA networks is known as polling.

To implement the SNA polling algorithm, a front-end processor (e.g. 3705) is placed between the transmission line and the host computer as shown in Figure 5. Conceptually, this processor tests the status of each terminal on the line in succession by sending a test message (a poll) to each. When a terminal receives a poll, it responds by either transmitting a message (if it has one ready) or by transmitting a "null response" indicating that it has no message ready. Once the front end processor has received a response from one terminal, it proceeds to the next terminal on the list. When the end of the list is reached, the polling algorithm in the front end processor returns to the beginning of the list and starts the next polling cycle. The list that controls the polling algorithm is known as the service order table.

Note that the simple queueing model in Figure 2 is no longer directly applicable since the sending of polling messages and null responses takes additional time, thus introducing additional delays between successive message transmissions. Suppose the service time per message is increased to account for these additional delays. That is, suppose the delays are added to the "basic message transmission time" in order to create a larger "effective message transmission time" for use in the model in Figure 2.

At a minimum, the additional delay is the time to transmit and propagate one polling message. However, there could also be several other polling messages, each followed by a null response, before an actual message is finally sent. Moreover, the number of null responses between actual messages will vary as a function of the total load on the line. That is, as the load increases, the expected number of null responses between actual messages will decrease. All these considerations make it difficult to determine the correct "effective service time" to use for the transmission line (the server) when attempting to apply the simple queueing model of Figure 2.

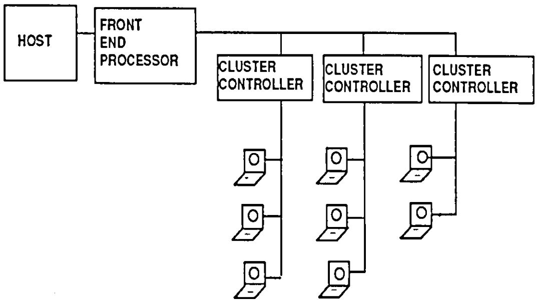


Figure 6
Position of Cluster Controllers

EFFECT OF CLUSTER CONTROLLERS

There are many other factors to consider when developing realistic models of SNA communications networks. For example, Figure 5 is somewhat simplified in that it shows each user terminal connected directly to the transmission line. In fact, a set of terminals is typically connected to single cluster controller, which is then connected to a line as shown in Figure 6.

The point here is that the polling messages generated by the front end processor are, in fact, sent to the cluster controllers rather than the individual terminals. This is more efficient since a single poll can now be used to test the status of several terminals. Hence, there will be less polling overhead and fewer null responses in each complete polling cycle. However, there is now an additional source of delay since the messages queued at terminals on the same controller must be transmitted sequentially. Thus, one of the messages at the controller will begin transmission as soon as poll is received, but all other messages present will experience additional delays prior to transmission. These delays are comparable in nature to the simple queueing delays in Figure 2, but must be combined with the polling delay described above for a complete analysis.

OUTBOUND MESSAGES

The original motivation for polling has just been presented for the case of inbound messages. In principle, outbound messages are simpler to deal with since they are queued together in the front end processor and are controlled by an algorithm executing on that processor. In SNA networks, this algorithm is conceptually similar to the inbound polling algorithm in that use of the transmission line is allocated to messages destined for each cluster controller by cycling through a fixed list (service order table). However, because the queue of outbound messages resides locally in the front end processor, the time to send polls and null responses is negligible. Thus, outbound message traffic is more naturally treated as an ordinary queueing process with local customers of a type illustrated in Figure 2. There are, however, a number of complicating factors, some of which are discussed below.

HALF AND FULL DUPLEX LINES

The transmission line connecting terminals with a host commputer must, in general, be able to handle both inbound and outbound messages. A half duplex line can only handle one message at a time, and thus must switch back and forth between inbound and outbound traffic, further complicating the polling processes described above.

In the case of a full duplex line, there can be one inbound messsage and one outbound message in transmission at the same time. This would appear to decouple the inbound and outbound traffic, allowing each to be analyzed separately. In fact, things are not this simple because there are interactions between inbound and outbound traffic even on full duplex lines.

One obvious interaction arises because of the interference between polling messages and normal outbound messages. Polling messages generated by the inbound polling process are transmitted on the same transmission lines as ordinary outbound messages. Thus, a model of the inbound polling process must account for the delay incurred when a polling message is being transmitted on the outbound message line. Similarly, a model of the outbound message process should include the effect of additional load due to polling messages. Some of the more subtle interactions between inbound and outbound polling processes are discussed below.

EFFECT OF MAXOUT, MAXIN AND ACKNOWLEDGEMENT MESSAGES

Data integrity considerations make it essential that all messages sent on a transmission line be checked for correctness at their destination. If the message passes appropriate tests (e.g., checksum OK), an acknowledgement is transmitted to the message originator. If there is a problem, a negative acknowledgement may be sent to request message re-transmission. Messages will also be re-transmitted automatically after a time-out period if no response is received by the message originator.

Complete messages in SNA networks are divided into a series of fixed length packets or frames (BLU's) prior to transmission. Each frame is tested for correctness at its destination, but to reduce overhead there are provisions for accumulating a batch of test results and sending only one acknowledgement per batch. The number of frames that are accumulated between each acknowledgement is specified by the parameters MAXIN and MAXOUT, which are always greater than or equal to one. MAXIN controls inbound messages, and MAXOUT controls outbound messages.

Suppose a total of MAXOUT frames have been sent to a cluster controller. No additional outbound frames will be sent until an acknowledgement is received. However, the acknowledgement represents an inbound message, and it cannot be transmitted by the cluster controller until the inbound polling process reaches it. Since the inbound and outbound polling processes operate asynchronously in a full duplex line, the outbound process typically experiences a delay at this point while it waits for the inbound process to catch up. In addition, the inbound acknowledgement messages generate overhead that interferes with the normal inbound message traffic. These factors must be included in any complete analysis of full duplex SNA networks.

EFFECT OF PASSLIM

Suppose a very long outbound message (comprised of many frames) is queued for transmission in the front end processor. If the entire message is transmitted when the appropriate entry is reached in the service order table, the line could be tied up for an extended period of time. To prevent this, the parameter PASSLIM is used to limit the number of outbound frames transmitted to a cluster controller each time it is encountered in the service order table. That is, once a total of PASSLIM frames have been transmitted, the polling process moves on to the next entry in the service order table even if there are more frames to send.

PASSLIM also limits the total number of frames that can be transmitted to all the terminals attached to a single cluster controller each time an entry for that controller is encountered in the service order table. For further analysis of PASSLIM, see [34].

It should be noted that an additional level of control is available in SNA since the number of times each controller appears in the service order table is user specified. Thus, if there are three entries for a given controller, the maximum number of outbound frames that can be transmitted to that controller per complete polling cycle is three times PASSLIM. Multiple entries also affect the inbound polling process, but in this case the main consequences are a reduction in polling delays prior to message transmission and an increase in apparent polling frequency.

EFFECT OF PAUSE

Suppose a complete scan of all the entries in the inbound service order table results in very few (or no) inbound messages. This indicates that line traffic was light during that polling cycle. Assuming that line traffic will remain light in the near future, it is not essential to begin a new polling cycle immediately.

The advantage of "pausing" before initiating the next polling cycle is that the front end processor's load will be reduced. This is helpful if the front end processor has other work to do, such as polling other lines for inbound traffic or managing the flow of outbound traffic.

The SNA paramater PAUSE is used to achieve this result. If the inbound scan time falls below the value of PAUSE, an appropriate delay is automatically introduced so that the time between successive scan cycles never falls below PAUSE. Proper setting of the PAUSE parameter is important for good SNA performance [34].

EFFECT OF SLOWDOWN MODE

The front end processor maintains buffers that are used to store messages waiting for outbound transmission to terminals or inbound transfer to the attached host. If line traffic is heavy, the buffers may fill up, making it impossible for the front end processor to accept additional inbound or outbound messages.

When this point is reached, the front end processor will devote all its capacity to clearing out its buffers. In the case of SNA networks, the front end processor is said to enter SLOWDOWN mode.

While it is in SLOWDOWN mode, the network will be unresponsive to users at terminals who are attempting to send new messages to the host. The additional delays that users experience at these times must clearly be considered when making a complete analysis of an SNA network.

CONCLUSIONS

The theory of queueing networks has advanced substantially in the past fifteen years. Much of the impetus for this expansion has come from the practical need to develop realistic models of mainframe MVS systems. With approximately one thousand MVS sites now using analytic modeling tools for capacity planning, it is clear that the gap between theory and practice has in fact been bridged. Recent advances in analytic models for SNA networks have enabled network capacity planners to take advantage of comparable performance prediction tools.

REFERENCES

[1] S.C. Agrawal, J.P. Buzen and A.W. Shum, "Response time preservation: a general technique for developing approximate algorithms for queueing networks", *Proc. 1984 ACM Sigmetrics Conference, Cambridge, MA, pp* 63-77, Aug. 1984.

[2] G. Balbo, S.C. Bruell and S. Ghanta, "The solution of homogeneous queueing networks with many job classes", *Proc. of the International Workshop on Modeling and Performance Evaluation of Parallel Systems,* Grenoble, France, pp 385-417, Dec. 1984.

[3] Y. Bard, "Some extensions to multiclass queueing network analysis." In M. Arato, A. Butrimenko, and E. Gelenbe (eds.), *Performance of Computer Systems,* North-Holland, 1979.

[4] Y. Bard, "A model of shared DASD and multipathing", *CACM 23,10, pp* 564-572, Oct.1980.

[5] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers", *JACM 22,*2, pp 248-260, Apr. 1975.

[6] J.W. Blaylock and S.C. Fisher, "Capacity planning on both Amdahl 470 and Sperry 1100 systems using BEST/1", *CMG'84 Conference Proceedings,* pp 215-222, Dec. 1984.

[7] A. Brandwajn, "A model of a time-sharing system solved using equivalence and decomposition methods", *Acta Informatica 4,*1, pp 11-47, 1974.

[8] A. Brandwajn, "Models of DASD subsystems: basic model of reconnection", *Performance Evaluation 1,3, pp* 263-281, Nov. 1981.

[9] J.P. Buzen, *Queueing Network Models of Multiprogramming,* Ph.D. Dissertation, Harvard University, May 1971.

[10] J.P. Buzen, "Computational algorithms for closed queueing networks with exponential servers", *CACM, 15,9,* pp 527-531, Sept. 1973.

[11] J. P. Buzen, "Fundamental operational laws of computer system performance", *Acta Informatica 7,2,* pp 167-182, 1976.

[12] J.P. Buzen, "A queueing network model of MVS", *ACM Computing Surveys, 10,3,* pp 319-331 Sept. 1978.

[13] J.P. Buzen and A. Bondi, "The response times of priority classes under preemptive resume in M/M/m queues", *Operations Research 31,*3, pp 456-465, May, 1983.

[14] K.M. Chandy, U. Herzog and L.S. Wood, "Parametric analysis of queueing networks", *IBM Journal of Research and Development 19,1, pp* 50-57, Jan. 1975.

[15] K.M. Chandy and D. Neuse, "Linearizer: A heuristic algorithm for queueing network models of computing systems",*CACM 25,2, pp* 126-133,Feb. 1982.

[16] L.K. Chu, "BEST/1 modeling experience and credibility at Security Pacific National Bank", *CMG '84 Conference Proceedings,* pp 494-505, Dec. 1984.

[17] A.E. Conway and N.D. Georganas, "An efficient algorithm for semi-homogeneous queueing network models", *Proc. 1986 ACM Sigmetrics Conference,* Raleigh, NC, pp 92-99, May 1986.

[18] P.J. Courtois,"Decomposability, instabilities and saturation in multiprogramming systems", *CACM* 18, pp 371-377, 1975.

[19] P.J. Denning and J.P. Buzen, "The operational analysis of queueing network models", *Computing Surveys 10,*3, pp 225-261, September 1978.

[20] J. Dwyer, "Modeling large IMS systems using commercial analytic queueing model software products", *CMG '84 Conference Proceedings,* pp 283-289, Dec. 1984.

[21] E.M. Friedman, J.L. Rosenberg and D.A. Sheetz, "XA migration: predicting the performance impact using analytic models", *CMG '85 Conference Proceedings,* pp 358-362,Dec. 1985.

[22] W.J. Gordon and G.F. Newell, "Closed queueing networks with exponential servers", *Operations Research 15, pp* 244-265, 1967.

[23] L.M. Greene, and J.J. Simon, "Establishing performance and capacity planning in a conversion environment", pp 452-458, *CMG '85 Conference Proceedings,* Dec. 1985.

[24] J.R. Jackson, "Jobshop-like queueing systems", *Management Science 10, pp* 131-142, 1963.

[25] S.S. Lam, Y.L. Lien, "A tree convolution algorithm for the solution of queueing networks", *CACM* 26, 3, pp 203-215 Mar. 1983.

[26] E.D. Lazowska and J. Zahorjan, "Multiple class memory constrained queueing networks",*Proc. 1982 ACM Sigmetrics Conference, pp* 130-140, Aug.1982.

[27] D.A. Menasce and V.A.F. Almeida, "Operational analysis of multiclass systems with variable multiprogramming level and memory queueing", *Computer Performance 3,*3, pp 145-159, Sept. 1982.

[28] M. Reiser and H. Kobayashi, "Queueing networks with multiple closed chains: theory and computational algorithms", *IBM J. of Research and Development* 19, 3 pp 283-294 May 1975.

[29] M. Reiser, "Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks", *Performance Evaluation* 1,1, pp.7-18, 1981.

[30] M. Reiser and S.S. Lavenberg, "Mean value analysis of closed multichain queueing networks", *JACM 27,*2, *pp* 313-322, Apr. 1980.

[31] C.H.Sauer, "Computational algorithms for state-dependent queueing networks",*ACM TOCS 1,*1,pp 67-92,Feb. 1983.

[32] P.Schweitzer, "Approximate analysis of multiclass closed networks of queues", *Proc. International Conference on Stochastic Control and Optimization,* 1979.

[33] R.Suri, "Robustness of queueing network formulas", *JACM 30,*3, pp 564-594, July 1983.

[34] A.K.Thareja, S.C.Agrawal, J.P. Buzen, D.E.Hall, R.W.McNitt, E.B. Petrovich, C.Zaiontz, "Impact of network parameters on the performance of single domain SNA networks", *CMG XIV Conference Proceedings,* Dec. 1983.

A Survey of the State of Art and Practice In I/O Subsystem Modeling and Analysis

Brian J. Smith

International Business Machines Corporation
General Products Division
San Jose, California

Abstract

This paper presents a survey of I/O Subsystem Modeling practice as seen in large IBM mainframe installations. It discusses state of the art techniques, but focuses on real problems, real techniques, and discusses the gap between art and practice.

The planning problem for I/O subsystems is a special case of the more general problem of system-wide capacity planning. Basically the problem is to predict the future behavior of a system from observation of its current behavior. There are two subproblems: the capacity planning problem for a particular installation and the system/subsystem design problem. The two problems are related, but the differ with respect to workload characterization. A particular installation can, presumably, measure and characterize its workload. System design problems must deal with typical or average or representative workloads rather than an actual workloads. Furthermore, the potential changes and variations in future workloads make the challange even greater for the system designer.

The approach to capacity planning or system/subsystem design being described involves measurements of existing computer systems to determine workloads. Modeling of the hardware configuration explains the behavior observed in the measurements. There are two distinct parts to modeling computer systems: 1) the system model describing the hardware, configuration, and algorithms controlling the system; and 2) the workload characterization which is imposed on the basic model. A significant factor in this approach is validation of the model -- making sure that the hardware model and workload characterization adequately explain the current state of the system behavior.

One approach to capacity planning or design trade-off studies uses Monte Carlo simulation. The performance analyst creates a model to simulate the detailed behavior of the hardware, drives it with workload descriptions and observes the system behavior. There is one fairly well known, though not generally available, tool based on such a simulation. It is called SNAPSHOT and described in [INGRA85]. Another approach is to use analytic queueing models. CRYSTAL is a modeling tool based on queueing equations. There is a very readable discussion and comparison of CRYSTAL and SNAPSHOT in [INGRA85].

For I/O subsystems, the models typically are based in simple queueing theory, though detailed simulation models work as well. If queueing theory is used to describe the behavior of the I/O subsystem, then the behavior of the hardware is transcribed into the primitives of queueing theory, namely arrival rates and service times. Arrival rates fall into the category of workload description, so the hardware part of the models involve more or less detail about the service time distributions or the size of waiting rooms in the system. The model must include such things as the number of I/O devices, number of data transfer channels, algorithms which control the hardware, such as path selection algorithms and whether or not the I/O devices use rotational position sensing (RPS). Other characteristics of the hardware such as the access times of the I/O devices must also be captured by the model. So the models are essentially collections of equations which describe the behavior (throughput and response time) of the hardware under various workload conditions.

The workload is described in units of work demanded of the system. First is usually the access rate. Some queueing models derive access rates from other input assumptions (such as multiprogramming level), but for most purposes we can assume the access rate for each device is an input assumption. The workload further includes such items as the number of bytes of data to be transferred (blocksize), and perhaps additional parameters involving seek times or other factors of interest. Estimates of these parameters for large IBM mainframe production systems can be found in [SMITH81].

The proportion of the access rate to each device is a very interesting piece of the workload description. Expressed as a probability distribution vector, we call this the I/O skew. Many of the examples in the literature assume a uniform arrival skew -- meaning all the devices do about the same amount of work. Of the input parameters for I/O subsystems, few have as much influence on the answers

CH2345-7/86/0000/0760$01.00 © 1986 IEEE

as the skew. It is fortunate that the arrival skew is routinely reported by most system monitoring reports, unfortunate that real skews are so seldom studied.

Many of the interesting articles in the literature focus on configurations, algorithms in the operating system, or microcode in pieces of the system. For example, multipathing considerations, RPS phenomena, and dynamic path reconnect are current topics of investigation. Cached DASD subsystems is an additional item of current interest. Performance models need to capture the operational characteristics of the microcode and data pathing implementations in order to adequately distinguish one product design from another.

Chapter 10 in the text by Lazowska [LAZOW84], provides a good introduction to I/O subsystem modeling. It discusses RPS phenomena, dynamic reconnect, and includes an introduction to cache modeling, as well. CMG '85 Conference Proceedings include a number of interesting case studies. Two which illustrate current topics are "Analysis of Device Level Select in IBM 3380 Disks" [BUZEN85] and "DASD Performance Analysis Using Modeling" [BERET85].

The text [LAZOW84] shows that analytic techniques are applicable to a broad range of problems in capacity planning and system evaluation. The mathematics has developed impressively, capturing the effects of several queueing disciplines and multiple classes of customers. These techniques have been implemented in a number of packages, such as BEST/1 and CRYSTAL from BGS Systems, MAPS from Amdahl Corp., and CMF from Boole and Babbage. SNAPSHOT and DCAT are IBM internal tools with similar capabilities, and VMPPF is an IBM program for capacity planning in the VM environment.

Simulation technology has also progressed over time. The IBM Research Queueing package (RESQ) is an example of a general simulation strategy with features particularly suited to the analysis of computer systems, capacity planning and system design questions [SAUER80].

After one develops a model of the hardware, the next step is to characterize the workload on the hardware. Workload parameters are usually estimated from software monitors. Examples of these monitors for large IBM systems include RMF for MVS, VM MAP for VM, and DOS/PT. Additional data is sometimes available from key subsystem facilities. For IBM products these include things like IMS Log, Database 2 Performance Monitor, and GTF Trace. Some applications lack adequate software instrumentation, which makes workload characterization difficult. For a good discussion of these issues, I suggest "Establishing Performance and Capacity Planning in a Conversion Environment," by Greene and Simon [GREEN85].

If one reads the articles mentioned in this survey, it becomes clear that capacity planning tools are plentiful and powerful. There is good progress and capability in analytic modeling. If the problems are important enough and too difficult for analytics, simulations can help provide the answers. Workload characterization is made feasible by software performance monitors. The goals and process of capacity planning are well understood, and are handled by several papers in the current session.

Most importantly, complete packages are available to do the job. They extract workload information from a running system, build a model of the configuration, and validate the model and workload characterization against the measured data. The performance analyst can use the model to study changes in hardware and/or workload. This does not mean that all the problems are solved. The "what if" studies still require experience and insight by the analyst. The amount of data involved in capacity planning studies can be enormous. Data entry and data manipulation is a problem. The current tools for for handling such data and system information are epitomized by SAS and MICS [CHEN85]. They foreshadow fourth generation database and data manipulation techniques.

Finally the whole process suggests knowledge-based or expert systems. The proceedings of CMG '85 have a representative collection of papers which capture the current state of this development ([ARTIS85], [HELLE85], [LEVIN85], and STROE85]). The essence of such systems is a combination of rules of thumb for good performance, combined with inferences based on the analysis of queueing models. Nevertheless, there is a considerable gap between what is possible and what is easy to do. [SMITH85] is an excellent dicotomy between the (largely unrealized) potential of the mathematical techniques and the problem of a friendly, flexible user interface.

Facing the Challenge of Other Problems

In the area of system/subsystem design, the difficult problems are a result of the long time frame involved. Real and potential changes to hardware must be evaluated, along with changes in software to exploit new hardware or simply to offer better function. The performance analyst is required to extrapolate far enough into the future to see where the system is likely to break or bottleneck. The job is to effect a change to the system design to avoid such s bottleneck. The system designer must stretch the capabilities of existing systems, thus establishing new rules of thumb for the system. This implies that a knowledge-based system for capacity planning will have to assimilate new rules over time.

Predictions as much as 2-5 years in the future rely on properly characterizing the workload and how it may be changing. From the perspective of a system designer, the problem is predicting the change in the workload. To be precise, the job is to design the change in the workload resulting from a change in the system design. Even simple modifications like a different number or type of devices in the I/O subsystem have potentially large impacts on the workload. As an example, consider a recent problem in capacity planning. What happens to the average seek time in an I/O subsystem when an installation migrates to a new, larger capacity device? It happens that there is nearly enough measurement data to predict the answer to this question, so it is surprising how many studies make simplyfing assumptions only to arrive at partial conclusions. This particular question is studied from real data in "The Impact of 2-to-1 Volume Folding on Seek Time" [MCNUT85].

But I/O subsystem workloads have many more elements in their characterization than seek times. There is connect time, disconnect time, including latency and RPS misses, as well as seek time. There is the access skew to the devices. For cached DASD there are hit ratios, read to write ratios. Even if you measure them in your current subsystem, how do you predict what happens to all of them when you make a change?

Another technique is to hold many study parameters arbitrarily constant while we change a few to study consequences. This specifically denies the possible interactions of several simultaneous changes. All this works well enough for capacity planning and its time frame, but the system design problem, with its longer time horizon, requires that more effort be spent to study possible interactions.

Other problems arise in the characterization and validation of a model. Consider a simple model of an I/O subsystem as a collection of M/M/1 servers. Each device is represented by a server with Poisson arrivals and exponentially distributed service times. Theory then predicts that at 50% device utilization, the response time for a device is twice the service time, or the wait time is equal to the service time. What does the measured data say? Measured data says that many devices have little or no queueing -- that is, the queue lengths are much less than most queueing equations predict. This is most easily understood by imagining the common situation in which exactly one task in the system is using the disk device. No matter how hard the task uses the device, there will be no waiting line when the task wants issues its I/O. Even so, M/M/1 queueing theory provides a lot of insight into the qualitative behavior of I/O subsystems. The problem, of course, is that as computer scientists, we want, expect, even promise quantitative descriptions of the behavior. So we apply the scientific method and try to improve our models.

What can we do to improve the predictions of wait time for I/O devices? We try M/G/1 equations, finite population models of one type or another, separable queueing equations [BUZEN73], even detailed simulations. But in the end there is always some system phenomenon which was not modeled -- which we handle by saying that the models are abstractions, and that the effects they ignore are second order effects. Examples include system design features that exploit parallelism in the I/O subsystem (such as the page/swap subsystem in MVS) or memory and lock contention in databases.

Usually they are second order, but not always. In many ways, the situation is similar to Newtonian physics versus relativistic physics. The simple models are adequate as long as you don't try to extrapolate their results too far beyond the range of current measurements. Trying to use more sophisticated models introduces additional complexity that may make it harder to decide about simple questions at hand. Using more complex models to attack the discrepancies on queue lengths adds extra parameters to the analysis -- parameters like multiprogramming levels, or variance of the service distributions. Often these new parameters cannot be extracted from existing monitors. As mathematically attractive as closed system models with nonhomogenous workloads may be, it is difficult to use an RMF report to apportion paging activity to different workload classes. For I/O subsystem models, it is hard to beat simple queueing theory augmented by heuristics to deal with new or unique features in hardware or microcode.

The problems are hard, but not insurmountable. So it is discouraging to see the simplistic answers that pass for computer science. It is discouraging to see performance studies use superficial techniques to predict disaster, but never check the assumptions against real data. If the workloads are characterized carefully, if the effects of change are given some thought, then the modeling techniques actually suggest ways to avoid disaster. Personally, I believe some performance analysts have more fun predicting disaster, especially if they can appeal to mathematical arguments to make the case that much more plausible.

On the other hand, models may suggest solutions which simply don't work in real systems. Queueing theory can be used to show the profound advantages of uniformly spreading I/O requests across all the I/O devices. But on systems like MVS or VM, the mechanisms which cause skew prevent us from easily modifying the skew toward uniform activity across devices. And in real systems it may not even be desirable. The busiest devices often have the least queueing because only a single task owns and uses the device. The mechanisms which spread activity across volumes would "probably" introduce queueing (wait time) where there was none. The operable word is probably. That is because there is no real program of experimental computer science to address these types of questions. There is not enough data to make scientific hypotheses to be tested by further measurements.

Right now, the mathematical techniques available to the capacity planner or the system designer are ahead of our willingness to collect and analyze data about real systems. We are unwilling to bear the costs. Better measurement techniques are possible, even advocated, but there is no systematic program of experimental computer science where a group is willing to put up money and facilities. Such a facility could be comparable to a physics research program. In part, this is because systems and technology are still evolving too rapidly for the performance engineers to evaluate every trade-off. Ferrari states other reasons for this phenomenon [FERRA86].

The Prognosis

The use of capacity planning techniques is widespread. Data processing shops are known to use sophisticated techniques, and the computer manufacturers apply similar technology to the design and evolution of their systems. Of particular promise are packages which measure a system, characterize its workload, construct a model of the hardware configuration and validate the model against the real system. Such systems are easy enough to use for them to become powerful tools for attacking capacity management problems -- they suggest tuning alternatives in the short term.

Traditionally, much tuning is trial and error. These modeling tools can now be used in a diagnostic mode to show where performance problems might be hiding. They can also use simple deductions based on queueing theory to suggest solutions to the potential problems. Admittedly, there are difficulties in this technique. The models need to be enhanced to deal with more complex workloads. There must be a way to incorporate some common sense into their recommendations. But you see the way this will evolve.

Already we read about expert systems for computer performance. Knowledge based systems to operate, tune, and design systems are receiving much attention. They will undoubtedly be a boon to the performance analyst in time. And as these systems become common and proficient, another development will occur. Systems will incorporate the same techniques to tune themselves. This is a natural evolution of operating systems and I/O subsystems. There are elements of computer systems which already exhibit this trend. The paging supervisor in MVS (VM too), the subsystems which allocate scratch and spool data sets on disk devices, and to some extent, the capabilities of cached I/O subsystems already exhibit the characteristics of self-tuning subsystems. So it is most fitting that the subjects of capacity management and capacity planning are topics of this particular joint computer conference dedicated to "Exploring the Knowledge-Based Society."

Bibliography

ARTIS85 Artis, H.P., "Using Expert Systems for Analyzing RMF Data," CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

BARD80 Bard, Y., "A model of shared DASD and multipathing," CACM 23, no. 10 (October, 1980).

BERET85 Beretvas, T., "DASD Performance Analysis Using Modeling," CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

BUZEN73 Buzen, J. P. "Computational algorithms for closed queueing networks with exponential servers," CACM 16, no. 9 (September, 1973).

BUZEN85 Buzen, J. P. and A. W. Shum, "Analysis of Device Level Select in IBM 3380 Disks," CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

CHEN85 Chen, W., "A Methodology for Modeling M204 On-Line Existing Workloads Using BEST/1 and MICS MVS Model Generator," CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

FERRA86 Ferrari, D., "Considerations on the Insularity of Performance Evaluation," These proceedings of the ACM/IEEE Fall Joint Computer Conference, November, 1986, Dallas, Texas.

GREEN85 Greene, L. M. and J. J. Simon, "Establishing Performance and Capacity Planning in a Conversion Environment," CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

HELLE85 Hellerstein, J, and H. Van Woerkom, "YSCOPE: A Shell for Building Expert Systems for Solving Computer-Performance Problems," CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

INGRA85 Ingrassia, F. J., "Performance of New Online Systems," CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

KLEIN75 Kleinrock, L., Queueing Systems, Volume 1: Theory, John Wiley & Sons, New York, 1975.

LAZOW84 Lazowska, E., J. Zahorjan, G. S. Graham, and K. C. Sevcik, Quantitative System Performance, Prentice Hall, Englewood Cliffs, New Jersey, 1984.

LEVIN85 Levine, A.P., "ESP: An Expert System for Computer Performance Management," CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

MCNUT85 McNutt, B., "The Impact of 2-to-1 Volume Folding on Seek Time", CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

SAUER80 Sauer, C. H., E. A. MacNair, and S. Salzer, "A Language for External Queueing Network Models," IBM Journal of Research and Development, vol. 24, no. 6, November, 1980.

SMITH81 Smith, B. J., "I/O Subsystem Workloads: Measurements and Modeling," 1981 CMG Conference, December, 1981, New Orleans, Louisiana.

SMITH85 Smith, C. U., "Experience with Tools for Software Performance Engineering", CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

STROE85 Stroebel, G. J., R. D. Baxter, and M. J. Denny, "A Capacity Planning Expert System for the IBM System/38," CMG '85 Conference Proceedings, December, 1985, Dallas, Texas.

THE EVOLVING ROLE OF SOFTWARE PRODUCTS IN CAPACITY MANAGEMENT: A SURVEY

Phillip C. Howard, Editor

EDP Performance Review, P.O. Box 9280, Phoenix, Arizona 85068

Abstract

A great many tools have evolved over the years to aid in the capacity management area. In this paper, seven different tool categories are defined and described, including job accounting systems, hardware/software monitors, operations management, communications management, data management, simulation and modeling, and integrated systems. The types and functions of tools are described for each area, including their role in capacity management.

Introduction

There is no shortage of tools to aid capacity planners in doing their jobs, although the sheer numbers of those tools presents its own problem in terms of selection and training. There is no tool which can be said to be the perfect solution for capacity management. Probably those which fall in the general class of simulation and modeling come the closest. That is, they are primarily intended for predicting future capacity requirements based on given workload data, although their predictions are only as good as the forecast data entered into the model and the validity of the model itself. In short, there is no perfect answer to the capacity management problem in terms of software tools, but there are many which can help to make the planner's job easier.

No attempt at capacity management can be successful unless grounded on a firm base of measurement data which accurately portrays the present actual operating environment in terms of both workload and resource utilization. For that reason, a number of tools must be included which are primarily performance measurement-oriented, but which are critical in terms of supplying capacity planners with good measurement data about present operating environments.

Performance measurement or management, as differentiated from capacity management, generally refers to the measurement of existing systems for the purpose of tuning or optimization. However, because capacity management is so dependent on good measurement data, we view performance measurement as a subset of the larger function of capacity management for the purposes of this paper.

The Role of Products

For the most part, the entire capacity management profession has developed from the tools introduced by private companies, mostly entrepreneurial ventures that evolved from advances in the systems programming area. Although the tools market is dominated by IBM MVS offerings, the same capabilities are available for all mainframe vendors, albeit in smaller numbers. Nevertheless, it is probably safe to say that the technology has generally been based on IBM operating system software.

In most fields, academic research is the driving force behind the development of a technology. Except possibly in the area of queueing-based modeling, where the initial work was begun in a university environment, nearly all of the tools were developed in entrepreneurial settings. Several examples come to mind: the contributions of Dudley Warner to the field of hardware monitoring; Ken Kolence's early development of a software monitor and founding of Boole & Babbage; Jeff Buzen's development of the first modeling package for commercial use and his part in the founding of BGS Systems; and Mario Morino's contributions to the collection and archiving of performance data. There are certainly many other examples of individuals and companies which have contributed to the growth of this field.

The principal professional organization in the field, the Computer Measurement Group, has by and large grown up around the tools and technology offered by these types of companies, as opposed to developing around a strict academic discipline. In short, it is the tools and their application, plus the expertise in hardware and software that is found in the many small companies which make up this field, that accounts for much of the growth and success of capacity management.

In this survey, we have covered seven categories of software tools which are in general use in the performance management/capacity management field. Only the simulation and modeling category is really dedicated to the capacity planning problem. Most of the rest are primarily measurement or optimization tools. Both job accounting and software monitors provide almost exclusively measurement capabilities. The operations management group is more oriented toward optimization and making life easier for the operator, but scheduling packages, for example, may have a simulation capability to test the impact of introducing new applications. Both the communications management and data management categories focus primarily on measurement and optimization, but within each there are packages to aid in configuration and space management. The integrated systems typically offer a variety of capabilities, often with an interface to a modeling package.

Job Accounting Systems

Most advanced operating systems keep statistics on internal operations and collect that data in a file or data base. Numerous software packages have been written to process such "job accounting files", both for the purpose of billing resource use back to customers and summarizing the information into meaningful performance reports. It is the use of job accounting files (such as SMF and RMF in the IBM environment) as the primary source of input which uniquely characterizes this category of software tools.

In most cases, the activity data is collected at the termination of a specific event such as a job step, a transaction, a TSO session or other work unit. There are literally hundreds of specific measures or facts which may be collected, depending on the specific system, but the most common are metrics relating to CPU utilization, disk

CH2345-7/86/0000/0764$01.00 © 1986 IEEE

utilization, I/O requests, memory usage, paging activity, spooling statistics, volume mounts, terminal connect time, logons/logoffs, and similar measures. Most job accounting systems have some data base management capability to permit the retention and processing of historical data.

By their very definition, all job accounting systems have some capability for charging usage back to customers, although not all with the same degree of sophistication. In many cases, there is both an on-line and off-line aspect to the system. The on-line capability generally provides users with immediate feedback on the cost of their jobs or sessions and may provide for on-line inquiry into accounting data. The off-line portion produces invoices and other accounting reports. Many systems provide capabilities for user-defined charging algorithms, the input of manual charges and adjustments, comparison to budget data, and cost liquidation for full recovery chargeback situations.

The other major use of job accounting systems is in the reporting of performance data. In addition to serving as a basis for computing charges, the collected data can be processed and summarized to provide a wealth of information on performance and capacity utilization. Data can be sorted and summarized by user, project, shift, department or any other entity which can be identified as the source of work. Since all data is date and time-stamped, reports can be prepared by day, week, month, or any other time period to show workload levels and resource utilization for both a central system and a network. Reports can also be prepared of abnormal terminations and "top 10" or "top 20" lists to identify the largest users.

Although job accounting packages generally don't provide the same level of performance detail as either hardware or software monitors, they have the advantage of not requiring any changes to the operating system or operating environment. For this reason, they often serve as the starting point for users just starting in performance evaluation and capacity planning.

Hardware/Software Monitors

Software monitors differ from job accounting in that they generally have "hooks" into the operating system so that they can collect performance data directly, either from operating system tables and/or "control blocks", or by sampling various internal status conditions directly. Hardware monitors collect data by attaching to "probe points" on the computer system itself, avoiding the CPU and memory overhead that is required by a software monitor.

Monitors of both types provide more accurate measurement data than job accounting information and generally provide a more detailed look at the internal workings of the operating system. They tend to be more system-oriented than job or work unit-oriented, as are job accounting packages, although many of these packages allow the user to see relationships between job class performance and overall system performance.

Hardware monitoring of mainframes seems to be on the decline. In the past few years, hardware monitors have been disappearing from the market, although there are still a number of them to be found in user installations. They have not disappeared altogether, however. There has been a significant growth in the use of hardware monitors for communications networks.

Software monitors may be either sampling-type monitors or event-driven. In either case, they do impose some amount of overhead on the system, requiring a small amount of CPU time, memory space, and a collector device, usually either tape or disk. There is generally an extractor component, which actually does the data collection, and an analyzer component, which produces hard-copy reports from the collected data. Many software monitors also provide for on-line, real-time display of performance data to assist operators in controlling the system. In most cases, operators control the activity of the monitors (sampling intervals, duration of monitor period, areas to be monitored, etc.) through an on-line terminal.

Many different measurements may be taken by monitors, including CPU utilization, I/O and channel activity, paging and swapping, device allocation, disk arm contention, internal system queues, interrupt activity, and storage utilization. Some monitors also provide information about network activities, reporting on such things as response time, TSO activity, transaction rates, and transaction types. They may also provide information about the internal workings of the operating system to aid in decisions regarding page fixing, module assignments. etc.

In addition to raw measurement data, monitors may also detect and report certain kinds of system problems, monitor performance activity against operator-defined performance thresholds, and produce a variety of performance profiles and reports. In many cases reports and displays may be in graphic form, and user-defined reports may be facilitated through a report generation language.

The outputs of monitors are intended to support system optimization activities, and in some cases the monitor itself may take direct optimization actions. In addition, the outputs support capacity planning activities by providing the basic measurement data needed to detect trends in utilization, bottlenecks in configurations, and workload changes. In some cases, the data collected by monitors can be used as direct input to modelling packages.

Operations Management

This category of software packages is really made up of several sub-categories. The common denominator among them all is their support of the operations function in terms of streamlining work flow and optimizing the human/computer interface at the operator control level. There are five sub-categories which are described separately below:

Backup and Recovery - One common problem that all DP installations have is the backup, restoration, and recovery of DASD files and data bases. These packages generally replace mainframe vendor supplied utilities with improved performance and added functionality. The systems provide for the dumping of files to tape, compression of the data, frequency control of backups, automatic JCL creation, and security through encryption and/or passwords. Another type of package facilitates recovery from tape and/or disk errors, and analysis of tape media for quality.

Hardware Management - There are basically two functions performed by this class of package. The first accumulates error statistics from logged peripheral errors, maintaining historical records and providing for reports on potential reliability problems based on threshold error limitations. Potentially bad devices and/or media are identified. The second type of package maintains inventory data on all hardware devices incuding descriptions, leasing/pricing data, dates installed, maintenance contracts and similar information to facilitate management of the inventory of hardware and the preparation of equipment lists and other reports as necessary.

Operator Control - There is considerable diversity among the specific packages in this grouping, but most are directed at an improved operator/system interface. Such packages typically give the operator better control over the running of applications, improved communications with the system, special function keys, new message capabilities, access to and control over spool files, displays of system and job status, shared consoles, and other operational controls. Some of the packages also provide special capabilities for problem or event tracking and reporting.

Scheduling - This is the largest of the five groups and generally these packages are the most sophisticated and complex. Scheduling capabilities provide for the automatic submission of scheduled jobs, taking that task out of the hands of the operations personnel. Such systems include built-in calendars, job history information, priority setting, on-line inquiry into job status, automatic rescheduling, recognition of pre and post run manual activities as well as job dependencies, the prompting of operators for expected events, schedule revisions, and JCL input and maintenance. Most scheduling systems can also be run in a simulation mode to test the effect of new applications. A number of reports are typically provided as well, including daily production schedules, jobs completed and backlogged, performancc against schedule, off-line workstation activities, and job tickets.

Tape Management - This special class of package is intended to optimize and simplify the manual handling involved in tape libraries. Typical capabilities include automatic volume recognition, operator prompts for needed tape mounts, an on-line tape catalog, early device release, security controls, ownership pools, scratch dates, usage history, maintenance of generation data, and identification of tape location (including vault storage). Tape management packages also provide for several types of reports including data set lists, volume serial number lists, scratch lists, daily pull lists, tapes due for cleaning and certification, and volumes out of service.

Communications Management

In the context of performance management and capacity planning, the communications network is often just as important as the mainframe environment. There are several sub-categories of software tools which assist the user in analyzing and optimizing the communications environment.

Configuration Management - There is quite a range of variety among this group of packages, but most are intended to facilitate the management of the physical network configuration. The more elaborate of the packages provide for inventory management of the entire communications network, including problem/change control and tracking. Some packages generate the necessary software tables, parameters, and commands to properly start up the network. Others allow for special configuration capabilities not supported by standard mainframe software, for example automatic speed recognition and code conversions. Various reports may be produced by certain of the products to aid in the management of the network.

Data Manipulation - The few packages in this category generally provide data compression capabilities to reduce transmission times and improve throughput.

Design Aids - This group of products provides assistance to network designers in engineering optimum network layouts. Simulation capabilities provide estimates of service times by priority and throughput rates based on the characteristics of the network, traffic, and applications as specified by the analyst. The more sophisticated of the packages may produce an optimized topological design in hard copy and/or on a display screen. Yet another group of packages provides for an analysis of telephone traffic with recommendations for optimum line usage.

Network Monitors - This class of products includes both hardware monitoring devices as well as software monitors. Like mainframe monitors, they are designed to measure performance characteristics of a network during operation and provide network control personnel with displays and reports of network status, with particular emphasis on the status of lines, response times, outages, and traffic contention. Other types of measurements may include disk I/O activity, message lengths and counts, numbers of transactions by type, buffer use, terminal traffic, error counts, and polling time. Some monitors allow the operator to conduct tests and diagnostics on remote devices and provide alarms when certain performance thresholds are exceeded. Log files and/or a data base of historical network performance data may also be generated.

TP Control Programs - This group of packages have also been called teleprocessing monitors, but should not be confused with the type of monitor discussed above. Control programs provide operating system-like services, specifically improved interfaces between the application programs and the remote devices, including line control and terminal interfacing. They often perform functions which one normally associates with operating systems, including the scheduling and dispatching of TP programs, file handling capabilities, access to data bases, and message switching. These control programs may also collect operating statistics, provide restart capabilities, generate transaction logs, and provide traces of TP activity.

Data Management

With the expanding use of Data Base Management Systems and the growth of data bases, DASD storage has become one of the most significant resources relative to both day-to-day system performance and long term capacity planning. There are a number of tools available to aid the user in designing a data base, tuning and optimizing DASD storage, and handling utility functions such as backup.

Access Methods - There are only a handful of these packages, but they are intended to provide faster processing and access times than standard access methods.

Analysis and Monitoring - This group of tools performs analysis of existing DASD environments, either by examining VTOCs or Directories, or by monitoring DASD activity during program execution (much like a software monitor). The former type report on space usage, free space, track allocation, clustering, etc. The latter provides actual measurements of service time components on disk, CPU times, record counts, etc. Some packages maintain trend information to predict when space will run out and may provide information on such things as pointer problems, defective tracks, workload volumes, and problem areas. Analysis reports may support decisions regarding reorganization of DASD space.

Backup/Recovery - There are a number of products on the market to facilitate the backup and recovery function, most providing faster operation than standard system utilities, plus added functionality. Backup functions may provide for data compression and restructuring of files, compaction of fragmented files, and audit trails. Recovery tools typically produce a log or journal of activities, provide for the back-out of transactions when necessary, and requeue messages after a restart.

Data Base Design - In many cases, the design of a data base has a significant effect on the performance of an application. This group of tools provide the user with information on which to base design decisions. Given various design alternatives and information about the DBMS and the workload, such packages provide estimates of space and time requirements, recommend record design and logical/physical file organization, and may simulate application program logic. Outputs are analytical reports providing information on which users can base design decisions.

Space Management - This group of tools perform an active, rather than a passive, role in terms of DASD space allocation and management. Many of them actually administer direct access storage including such capabilities as archiving and migration of dormant files, sharing of disk areas between partitions, deleting obsolete files and truncating unused space, pooling of unused fragments of space, and dynamically allocating space as requested. Another group of products provide for optimizing the selection of modules for in-core storage to eliminate frequent fetches. In many cases, the products in this category produce reports analyzing the use of storage space, much like the Analysis/Monitoring category.

Utilities - Various DASD utility programs provide specific functions to improve the use of DASD storage including such capabilities as data compression, calculation of optimum blocking factors, conversion of data files from one device type to another, and encryption of data files.

Simulation and Modeling

For the most part, this category consists of queueing-theory-based modeling packages which are used for prediction of future system performance and capacity planning. The models typically allow the user to specify system configurations and workload characteristics and volumes, and produce projections of system response times, throughput, CPU utilization, device utilization, waiting times, and queue lengths. Most packages have interactive capabilities allowing the user to alter the basic inputs to the model and ask "what if" questions regarding future performance and capacity expectations.

Many of the packages accept measured performance data from job accounting or software monitor files and use this as the basis for constructing so-called "baseline" models in order to validate the basic model for the given environment.

In addition to modeling mainframe system operation, there are also several models which can be used to evaluate communications networks, and others specifically designed to predict the performance of applications prior to their development. There are also synthetic workload drivers which allow a user to simulate a real workload on their actual hardware and software configuration and measure the results directly.

A number of years ago, before the advent of queueing-based models, there were several event-based simulation packages which were used primarily for equipment selection decisions. These packages maintained libraries of hardware/software specific data to enable the comparison of different vendors' systems on a given workload. There is only one of those packages, SCERT, which is still being marketed. Otherwise, the queueing-based models are used primarily for capacity planning within a single installation, rather than for comparison of different vendor's equipment.

Integrated Systems

This relatively small group of products generally encompass several of the functional characteristics covered in the other categories, possibly providing capabilities in the areas of job accounting, monitoring, operations management, and modeling or at least an interface to another modeling package. One of the unique characteristics they possess is a performance data base, or a formal and organized approach to the archiving and summarization of historical performance data.

These packages provide capabilities in both performance evaluation and capacity planning, usually with interactive user control over their operation as well as an extensive repertoire of reports.

Summary

Readers might argue that many of the capabilities of software tools discussed above don't really have much to do with capacity management. Technically, this is true; many of them are strictly for tuning and operational enhancement. However, all of the categories should be considered within the context of capacity management, even though some of the specific capabilities may not be applied in that area.

The important point to remember is that any capacity plan must be constructed on a foundation of solid measurement data. All of the various tool categories provide measurement data in one form or another. Furthermore, even optimization and tuning affects capacity planning to the extent that future hardware requirements are reduced as the performance and throughput of existing systems is improved. Certainly, capacity planners must at least know what the possibilities are and when improvement efforts are being applied.

We have not provided any references throughout this paper, not because there aren't any, but because there are too many, few of which really stand out, at least when we restrict the subject matter to tools per se, rather than the larger subject of the tool category itself. In ACR's *EDP Performance Management Handbook*[1], Volume 1, there are bibliographies for most of the areas covered listing numerous references. Similarly, in the Annual Reference Issue of the *EDP Performance Review*[2] a subject index of the literature for the previous calendar year includes coverage of most of the tool categories.

The only book which has been written specifically on the subject of tools in this field is *Computer Performance Evaluation: Tools and Techniques for Effective Analysis*[3], although there is also coverage in the *EDP Performance Management Handbook*[1] of tool usage (Volume 1) as well as detailed descriptions of specific tools in Volume 2 (updated quarterly). The Special Reference Issue of the *EDP Performance Review*[2], mentioned above, provides capsule descriptions of the available tools in ten subject categories.

References

[1]. Howard, Phillip C. (Ed.), *EDP Performance Management Handbook*, 2 Volumes, Loose-leaf, 1986, Phoenix, Arizona.

[2]. Howard, Phillip C. (Ed.), *EDP Performance Review*, monthly, 1986, Phoenix, Arizona.

[3]. Morris, Michael F. and Roth, Paul F. *Computer Performance Evaluation: Tools and Techniques for Effective Analysis,* Van Nostrand Reinhold, 1982, 260 pp., New York, NY.

THE EVOLUTION OF WORKLOAD MANAGEMENT IN DATA PROCESSING INDUSTRY: A SURVEY

T. Leo. Lo

McDonnell Douglas
Aerospace Information Services Company
St. Louis, Missouri 63166

ABSTRACT

Data processing workload management is an essential component of the capacity management (CM) process. Workload management consists of workload characterization, workload forecasting, and workload control. Workload characterization is concerned with the measuring and modeling of production workloads; workload forecasting is a process of projecting future resource usages based on measured or observed statistics; workload control is a procedure to implement and monitor the workloads.

This paper reviews the past ten years (1976-1986) of workload management activities in the data processing industry based on documents and proceedings from the Computer Measurement Group (CMG), the International Conference on Computer Capacity Management (ICCCM), the Computer Performance Evaluation User Group (CPEUG), and the European Computer Measurement Association (ECOMA) conferences. All three areas in workload management are discussed with emphasis on their past and present activities and future direction.

INTRODUCTION

Capacity Management (CM) can be generally described as a process to ensure that the processing capacity can and always will deliver the services to meet DP users productivity goals within their budget requirement. These capacity, service and cost requirements are the integrated mechanism driven by the processing demands (workloads) to provide information for DP management making appropriate decisions. Therefore, capacity planning, service level management, cost management, and workload management, together with the corporate business plans constitute a management decision support system (ALLE83, DITH83, HOWA80, LO84, MULLE85).

Capacity planning is concerned with the management of resources and configuration and the prediction of service impact due to workload changes. Service level management is concerned with the service agreements between service provider and end-users in terms of timeliness, accuracy, reliability, and cost (HOWA80). Cost management deals with the recovery of DP expenses by establishing standard rates for different types of users running in different time zones and applications. Workload management is treated as an external driving force to the capacity-service-cost component due to its vacillating nature. The corporate business plans are also treated as an external factor, but they are more influential than workloads since business plans include many non-DP factors such as business strategy, industry growth potential, competitive pressures, and economic impacts. The conceptual diagram of these relationships is shown in Figure 1.

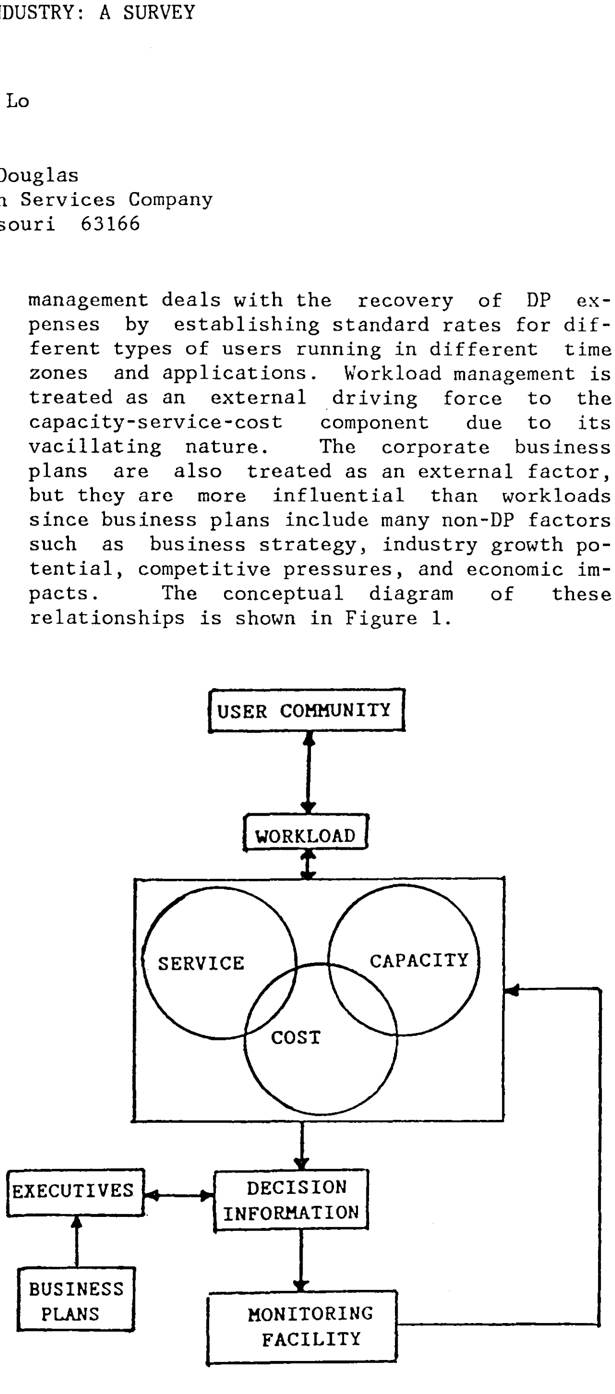


FIGURE 1 - A MDSS STRUCTURAL DIAGRAM

This paper only addresses the workload management portion of the capacity management. It however will conceivably address other areas of CM since they are interrelated. Workload man-

CH2345-7/86/0000/0768$01.00 © 1986 IEEE

agement consists of three key prospective areas: workload characterization, workload forecasting and workload control. Workload characterization is concerned with the measurement and modeling of the production workloads; workload forecasting is a process of projecting future resource usages based on measured or observed data; workload control is a procedure to incorporate the forecasted requirements into the capacity planning process and continue to monitor the implemented workloads for possible future modifications. It is not the intent of this paper to suggest or recommend methods or techniques for workload management. It merely reviews all three areas in detail with respect to how the DP industry has been dealing with them in the last ten years (1976-1986), and what the future directions are. Some of the problems encountered in managing workload are also presented.

The references used in this paper are based on the Computer Measurement Group (CMG) 1976-1985 conference proceedings, the International Conference on Computer Capacity Management (ICCCM) 1979-1985 proceedings, the Computer Performance Evaluation User Group (CPEUG) 1976-1983 conference proceedings, and the European Computer Measurement Association (ECOMA) 1983-1985 conference proceedings. All references are listed in one of the three areas based on the nature of the material: workload characterization, workload forecasting, and workload control which addresses other areas in CM including the workload management.

WORKLOAD CHARACTERIZATION

Workload characterization, or workload classification as some DP analysts call it, is concerned with the measurement and modeling of the production workloads. The purpose of characterizing the workloads is to understand/determine the resource usages and performance behavior for subsequent system tuning, performance evaluation, and/or capacity planning activities. Most workload characterization performed in DP industry is for workload forecasting which is used as an input to the capacity planning process.

The most publicized workload characterization technique is the Cluster Analysis approach. It is a technique which attempts to identify the natural groupings of jobs (or transactions) based on the similarity of the resource requirements. One way to determine the resource requirements is to use the vector form. There are generally five steps to group jobs or transactions.

* Define a resource vector (CPU seconds, storage bytes, disk I/O etc.) for jobs;

* Identify sets of similar jobs (or jobs steps) based on the description of their resource requirements. A geometric distance between a resource vector and a cluster is the measure of similarity. A job can be assigned to a cluster by finding the minimum of the distances between this job and each of the cluster centroid;

* Scaling the resource vectors; scaling is designed to avoid possible mistakes when comparing the geometric distances of a resource vector (e.g. number of tape drives) with another distinct resource vector (e.g. No. of disk I/Os). Scaling can normally be accomplished by normalizing each resource with its robust mean and standard deviation via a Z statistics;

* Provide the description of all the input data and cluster assignments. When the clusters generated from the sample input are used to classify the entire input data, each resource vector is assigned to the nearest cluster based on the geometric distance calculations; and

* Create a workload mix using a mix vector representing the different time intervals (i.e. prime shift, weekends,etc.) for subsequent benchmark generation.

Artis (ARTIS76) presented a case study using this technique for determining the capacity of a batch computer system at the 12th CPEUG conference in 1976. Agrawala and Mohr (AGRAW76) at the same conference presented the result of applying the cluster analysis technique to predict the workload of a computer system. Agrawala & Mohr (AGRAW77A) again presented the results of applying the clustering approach to workload modeling at 13th CPEUG conference in 1977, and discussed the relationship between pattern recognition problems and workload characterization problems at the 1977 CMG conference. Since then there have been many applications of the cluster analysis on computer workload modeling (AGRWA78B, HUGES78, ARTIS79, HARTR79, ELMS80, MOHR80B, LEVY81, SMITH81, FRIED83, LEE83, VINCE84, DOMAN85).

In addition to the application of cluster analysis, there were several articles discussing characteristics and problems of the technique. Mohr (MOHR80A) presented a survey of available workload characterization techniques at the 1980 CMG conference. There were many articles in ICCCM conferences to discuss workload characterization problems (MERLO83, PRICH81, YEN83). There were also many articles discussing the workload characterization and modeling implications/problems at the CMG conferences (HUGES80, MOHR81, SMITH81, WIGHT81, GILMO82, FRIED84, DOMAN85).

Although benchmarking is a natural continuation of workload characterization, it can be treated as a separate topic in workload management. Benchmarking is one of many ways to determine the dynamic nature of a user workload on a given system configuration. Determining the workload characteristics and the basic benchmark measurements can be accomplished following basic modeling techniques. There were so many fine articles on benchmarking at the ICCCM, CMG, CPEUG and ECOMA conferences (e.g. DUJMO80, KAZLA83, PLICH84), that this paper does not attempt to address the benchmarking issues.

The second workload characterization approach is the use of Software Physics concept first introduced by Kolence (KOLEN76) in 1976. It is a study of the quantitative and measurable properties of executable codes and their operands in relation to configurations and class of processors and storage devices. Kolence (KOLEN77) and Hoffman (HOFFM77) both gave a very good presentation on software physics at the 1977 CMG conference. Software physics focuses on a single measurement: software work. User demands, system performance and future requirements can all be represented and quantified through the application of this measure. The basic measurable properties are work (W), time (T), and storage (S). The basic unit is a quantified unit which can be converted to software work per equipment class by means of some scalar quantity. Workload can then be characterized in a vector form which can be normalized as the unit vector form. For example, for a Mth class job, it could have

$$J(M) = W \begin{vmatrix} W(CPU) \\ W(I/O) \\ W(tape) \end{vmatrix}$$

units of software works, where W is the total work performed over a period of time for Mth class job, and W(x) is the measured work for an equipment class, CPU, I/O or tape.

The application of software physics has been greatly promoted by the Institute for Software Engineering, which was changed to the Institute for Information Management later to broaden its services to the information society. Greenacre (GREEN76) presented a general description of the software physics at the 1976 CMG conference in Atlanta; since then more applications were reported at various DP conferences (HOFFM77, DAVID79, KOLEN79, SENSA79, WILLI80, KOVAC81, MEADA83, CLEME85).

In addition to the cluster analysis and software physics approaches, the traditional way of using some statistical methods still accounts for a significant portion of the workload characterization activities in the DP industry. The common practices are: (i) to identify the resources to be tracked, such as CPU utilization, I/O EXCP counts, (ii) to collect resource usage statistics, and (iii) to group resources either by major application class (e.g. IMS, TSO), time zones (8 to 5, weekend), or both. However, this approach is different from cluster analysis and software physics that it uses collective sum of ALL jobs belong to an application class, thus ignoring individual job's influence.

Anderson (ANDER79) presented a technique using statistical regression to determine the resource consumption overhead for each major class of users at the 1979 CMG conference. Davis and Lo (DAVIS81) reported the evaluation results of several methods in determining resource overhead for proper workload characterization at the 1981 CMG conference, and concluded that even with the production workloads, there is no one method that is always superior than the other methods. The Statistical Analysis System (SAS) is probably the most widely used tool for reducing and grouping workload data on IBM systems (e.g. PEDRI78, PHIPP81, BROCK82, and HINTZ83). Again, the data manipulation is a separate topic in capacity management and is also addressed at this conference. Therefore, most SAS related reports were not included in the reference.

In the last six years, the concept of the Natural Forecast Unit (NFU), introduced by Kolence (KOLEN76), has been gradually applied to both workload characterization and forecasting. The NFU approach is to identify and group resource usage patterns based, not on the traditional DP resources, but on the end-user business oriented terms, such as number of checks printed and number of hospital beds. This approach is different than the traditional approaches, including cluster analysis, that it uses a bottom-up approach as opposed to the top-down approach. The NFU approach will be explored further in the workload forecasting section.

WORKLOAD FORECASTING

Forecasting can generally be described as a process of projecting the past into the future. It is concerned with determining what the future will look like. There are three types of forecasting routines that are frequently referenced by the DP industry:

* Short-range forecast - usually is in monthly/quarterly units and extends out over the next few months/quarters, but seldom goes beyond one year. It is sometimes referred to as an "operating" forecasting since it deals with the immediate future operations.

* Medium-range forecast - usually is made on a quarterly or annual basis, and is used to establish the processing capacity for the next year. It is often referred to as the "budgetary" forecasting. Some organizations extend the range to one to three years.

* Long-range forecast - most companies make some attempts in various degrees of sophistication to determine the long-run course of the future. The business direction and national economy trend are the key elements considered. It usually forecasts over the next five to ten years; thus is referred to as the "strategic" forecasting.

Most DP workload forecasting activities deal with the short and medium range forecasts, and very seldom deal with the long range forecast (MAGER79, SHERK84) since the DP workload forecasting tends to become less accurate the further into the future.

Before any forecast is done, a forecast system needs to be established. Three generally used approaches are:

* Model development - based on objectives, various models can be properly chosen. One analyst may use a regression model whereas the others may wish to use a smoothing technique.

* Eclectic research - is a strategy of comparing one method against the others both objectively and subjectively. Tables, graphs, surveys, and models are commonly used to serve the means of comparison.

* Judgement model - it is generally a subjective process, qualitatively but not quantitatively. It usually consists of information or opinion from people who have direct experience on the subject, historical analogy, and the management's direction.

Most DP installations establish their forecast systems using the first two approaches. This paper does not intend to address how to establish a forecast system.

Once a forecast system is established, certain analysis techniques are required to calibrate and validate the data. There are three general approaches that are used by DP analysts and planners: naive method, time series and natural forecast unit.

The naive method generally does not attempt to explain the cause of an event, but simply examines variations over times. In other words, it only uses data on the variables of interest and simply projects the historical patterns into the future. The "rule-of-the-thumb" sometimes are applied to incorporate expert's experience into the data analysis process. In the early DP days when the system accounting data was not fully implemented, the eclectic research and judgmental approaches were used.

The second type of data analysis approach and probably the most common approach is the use of the time series techniques. A time series is a set of observations taken at specific times, usually at equal intervals. It can be described by thinking a point moving to the passage of time. Analysis of such movement is of great value in many respects, one of which is the problem of forecasting future movements. There are in general four types of movements:

* Trend movement - refers to the general direction in which the point moves over a long period of time.

* Cyclical movement - refers to the long range oscillations about a trend line or curve. The cycle may or may not follow exactly the similar patterns after equal time intervals.

* Seasonal movement - refers to the identical patterns which a time series appears to follow during the corresponding months of successive years.

* Irregular or random movement - refers to the sporadic motions of time series.

A majority of the reported DP workload forecast approaches use the time series analysis techniques (DEAGR84, HOFFM82, KULP80, LUIST83, MACKI78, PERLM79, ARTIS80, LINDE80, NIEDZ83, REED81, SHERK84). Some consider the trend only while others consider both trend and seasonality effects. There were also general descriptions on workload forecasting techniques at the CMG (BIASI85), ICCCM (LO80), and CPEUG (MCNEE79) conferences. Although workload forecasting is commonly used to determine an application's future resource usages, it can be used to estimate other types of DP activities such as software development effort (GODWI85), new applications (INGRA85), resource correlation analysis (YEN85), and service level management (HALBI80). In addition to the application of time series techniques, the potential problems facing workload forecasting were also discussed by Allen (ALLEN83), Applegate (APPLE83), Lo (LO85), and Wandzilak (WANDZ84) at the CMG conferences, and by Lo (LO84) at the 1984 ECOMA conference.

The third approach of the DP workload forecasting is the use of the natural forecast unit (NFU) concept. The NFU concept is also referred to as the Key Volume Indicator (KVI) by Sarna (SARNA79). A NFU is a measurable business oriented unit for which DP resource usages are tracked. For example, in an airplane manufacturing factory, the appropriate NFUs could be the number of engineering drawing hours, number of spare parts ordered, and number of training hours, etc. The underline principle behind the NFU approach is that the DP users, given sufficient information, understand their own needs. They probably can not forecast accurate DP resource usages, but can predict their business related growth with a fair degree of confidence.

When a corporation's business plans are established, they can be translated into the corresponding DP resource units (DPUs). Therefore, the NFU approach attempts to link business plans to DP plans. Figure 2 shows a conceptual process which links corporate plans and DP plans.

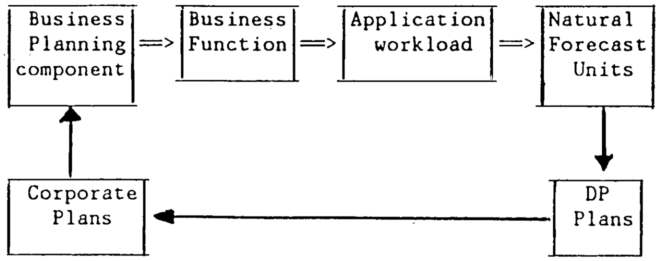


Figure 2 A Conceptual NFU Approach

To Correlate the NFUs to DPUs is not a straightforward process. In addition to the correlation process, the accuracy of the future DPUs is depended on the accuracy of the business forecasts. It is therefore advisable to use multiple forecasted values to represent a range of expectations in order to minimize forecasted errors. The NFU approach requires a computer resource usage measurement package, such as job accounting system, and a statistical method, such as regression models, to correlate business and computer forecasts.

Febish (FEBIS81) discussed the fundamental rules of converting business plans to DP workload forecasts at the 3rd ICCCM in 1981. Waggoner (WAGGO84) emphasized the importance of database design to integrate business and DP plans at the 6th ICCCM in 1984. Liu (LIU85) presented an excellent case study on using NFU dependency analysis for the production CICS workload forecasting at the 7th ICCCM in 1985. Lo and Elias (LO86) discussed the pros and cons of applying the NFU concept in a manufacturing environment (the paper has been submitted to 1986 CMG conference to be held in Las Vegas).

WORKLOAD CONTROL

The workload control is concerned with the implementation and monitoring of the workloads. The implementation involves the data selection, data adjustment, and forecasting technique implementation. Workload monitoring includes data collection and reporting, and data projection analysis.

The essence of workload forecasting is the extrapolation into the future of some structure in the past and present situation, regardless which approach is used. The approach itself, however, can not provide good forecasts unless there are some controls over input data as well as the selection of the forecasting parameters. Therefore, the forecasting parameters must be determined. As indicated before, most DP installations use system accounting data as the forecasting parameters. The length of historical data to be collected is suggested, as a rule of the thumb, at least four times the square root of the number of forecasted periods (LO80); that is, d > 4 * SQRT(f). Others simply use d > f.

There are two major concerns regarding data selection. The first one is the data quality; most DP installations archive a lot of system log data over a period of several years, but may not have the "right" type of data. For example, if the number of sheet metal cut per hour is used as the forecasted NFU, an installation may not have that type of information in its historical database. The second concern is the way data is collected and interpreted. For example, marketing people may forecast their sales revenue based on individual salesman's numbers which generally contain more variances since they always use a % growth over the past year.

To reduce data variances, data need to be adjusted. There are four practical ways for data adjustment. The first one is the adjustment of the time period. Most forecast mechanisms are based on the assumption that data arrive at fixed intervals. An accounting month may not be the same as a calendar month. Therefore, accounting data periods may require adjustment. One way to adjust the differences between a four-week month and a five-week month is to assign weights to make them equal. The second data adjustment is the adjustment for known causes such as holidays, strikes, etc. One example of data adjustment used in DP industry is to determine the peak-to-average resource usages for planning purpose.

The third data adjustment is to adjust the range of the data in order to reduce the variances. For example, the Z statistics is used in cluster analysis (ARTIS76) to reduce the variances between two types of resource vectors with distinct values. The fourth type of data adjustment is also concerned with the reduction of data variances. It transforms the data before carrying out the forecasting process. The common transformations are

$$Y(t) = LOG\ X(t)$$
$$T = X(t) - X(t - 1).$$

The first one is commonly applied to reduce the percentage changes to absolute changes, to reduce the growth model to a linear model, or to change from a multiplicative to an additive seasonal form. The second transformation is frequently used as a device for removing the effect of a wandering mean.

The workload monitoring process involves data collection and reporting, and projection analysis. Data collection and reporting are generally managed by using either vendor supplied system accounting packages, home-grown data collection tools, or commercial packages. Regardless which method is used, a large amount of storage would be required to to store the historical database. Most DP installations keep the current month/quarter data on on-line storage devices and the past months/quarters on either off-line storage or mass storage system. In order to produce reports, most IBM installations use SAS to manipulate their voluminous data. There are many data collection and reporting packages, such as Morino Associates' MICS, Boole & Babbage's CMF, Boeing Computer Services' SARA, and IBM's RMF and SMF, to just name a few.

One of the major concern, from capacity management viewpoint, is how to link the end-user services (e.g. response time and availability for on-line applications) and the processing capacity (e.g. CPU, memory and I/Os). In a large corporation, it is cost prohibitive to measure the entire workload with respect to linking service and capacity. In other words, what end-users see the response time (service) may be different from what the monitor reports simply because delays occurred at the communications network components can not be accurately and/or completely measured. Consequently, the forecasted resource (capacity) usages can not be correlated with the services delivered.

The projection analysis is actually a review process which tracks forecasting parameter statistics, incorporates the forecasts into the management decision support system, and reviews the feedback process. The center of the review process is the management decision support system; the relationship among various components in capacity management has been extensively discussed at the CMG, ICCCM and ECOMA conferences (ALLEN83, DITHM83, HOWAR80. KOLEN77, LO84, LO85, MULLE85, ORCHA85, POLLA84). The overall concept of the review process can be illustrated by the detailed execution flow of the NFU application discussed by (LO86), as shown in Figure 3.

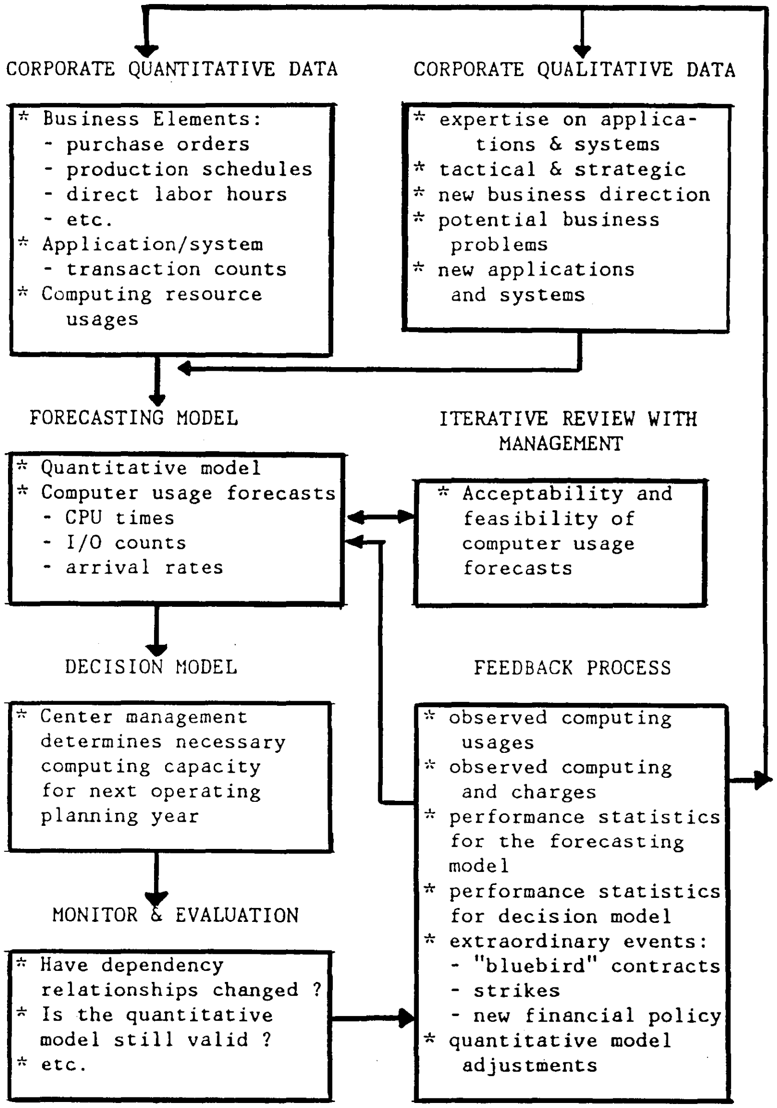

CORPORATE QUANTITATIVE DATA
* Business Elements:
- purchase orders
- production schedules
- direct labor hours
- etc.
* Application/system
- transaction counts
* Computing resource usages
CORPORATE QUALITATIVE DATA
* expertise on applications & systems
* tactical & strategic
* new business direction
* potential business problems
* new applications and systems
FORECASTING MODEL
* Quantitative model
* Computer usage forecasts
- CPU times
- I/O counts
- arrival rates
ITERATIVE REVIEW WITH MANAGEMENT
* Acceptability and feasibility of computer usage forecasts
DECISION MODEL
* Center management determines necessary computing capacity for next operating planning year
FEEDBACK PROCESS
* observed computing usages
* observed computing and charges
* performance statistics for the forecasting model
* performance statistics for decision model
* extraordinary events:
- "bluebird" contracts
- strikes
- new financial policy
* quantitative model adjustments
MONITOR & EVALUATION
* Have dependency relationships changed ?
* Is the quantitative model still valid ?
* etc.

Figure 3 NFU Application Execution Flow

DISCUSSIONS & CONCLUDING REMARKS

It appears that cluster analysis is more popular than software physics probably because of its statistical nature for easier acceptance. The software physics actually requires no more background skill than cluster analysis technique, but has not gained much acceptance probably due to its symbolic formats that are not natural to most DP analysts. One of the concerns on cluster analysis is that if a cluster contains a mix of applications such as TSO and IMS transactions, and short batch jobs, where and how to properly incorporate end-user's forecasted growth. In other words, a cluster's future direction may represent that particular cluster's growth, but an application's forecasts do not break down by clusters. Hence, it may require another layer of processing and analysis effort to translate user's forecasts into each clusters. The concern on software physics is not on its technicality, but on its formality. Performance analysts may be familiar with the hexadecimal numbers but may not be, psychologically, familiar with the Greek-like symbols and its multidimensional vector relationship.

However, both approaches are no doubt better than the traditional intuitive workload characterization approaches. One of the major factors that will make any method popular is through commercialized packages. Although there have been a lot of reported case studies on workload characterization, they are individual applications. There are several cluster analysis programs such as the ISODATA Points program by Ball & Hall from Stanford Research Institute (ARTIS76). The Institute for Software Engineering (now The Institute for Information Management) had partially incorporated the software physics concepts into their DP educational programs and consulting services. For example, they developed a set of programs to measure the relative power of various types of CPUs and I/O devices. However, as far as workload characterization is concerned, there is still a demand for an automated commercial package to characterize the workloads from immense data.

Workload forecasting is probably the most difficult task in capacity management process. It is a collection of individual forecasts which tend to inherit errors from the algorithm used. Another layer of complexity is that each corporation's business plans actually govern the future business direction. Business plans are made based on previous experiences and external influences rather than internal measurable units; hence they tend to produce higher forecasting errors. One good example is that the sale of the high technology product (i.e personal computers) was forecasted to be doubled every year. We have seen a much slower growth rate after only five years of the introduction of the personal computers.

As far as DP workload forecasting is concerned, the dominant techniques are still statistical methods especially the time series techniques. The NFU/KVI concept is an ideal way to incorporate business activities into DP plans, but its acceptance has not been as expected mainly for two reasons. First, there are lack of standards to select proper NFU parameters to forecast since every business preserves its unique characteristics and requires special skill and understanding to select NFU parameters; and second, there is no available commercial package which can readily link the business plans and the corresponding DP resource requirements. Boole & Babbage company is probably one of the leading developers to incorporate the NFU concept into their CMF product lines. Regardless which approach is used, there is still a demand for a complete commercial package to incorporate proper forecasting techniques into an automated process with satisfactory results. Again, a word of reminder: the skill, experience and knowledge of the analyst still play an important role in the workload forecasting process.

The workload control is actually an instrumental measurement and reporting process. There are many automated data collection and reporting packages that provide excellent data manipulation and management capabilities for a DP installation. The only problem is that there is no one complete package that can link the capacity, service and cost information to produce a timely decision support information so that the management can make right decisions at the right time to provide the best services at the most economic means.

In summary, the three pertinent components of the workload management have been discussed. There have been many individually reported case studies applying various techniques in workload management. But the urgent need is not to develop more techniques but to incorporate available techniques such as cluster analysis and the NFU concept into a complete package which provides sufficient and timely information for the analysts and management to properly manage their workload growth in the years yet to come.

TECHNICAL REFERENCES

I. WORKLOAD CHARACTERIZATION

ARGAW76 Agrawala, A.K. & Mohr, J.M., "Predicting the Workload of A Computer System," 12th CPEUG Conference; November, 1976, Washington, D.C.

AGRAW77A Agrawala, A.K. & Mohr, J.M., "Some Results on the Clustering Approach to Workload Modeling," 13th CPEUG Conference; October, 1977, New Orleans, Louisiana.

AGRAW77B Agrawala, A. K. & Mohr, J.M., "The Relationship Between the Pattern Recognition Problems and Workload Characterization Problems," 1977 CMG Conference; December 1977, Washington, D.C.

AGRAW78A Agrawala, A.K. & Mohr, J.M., "A Markarvian Model of A Job," 14th CPEUG Conference; October, 1978, Boston, Massachusetts.

AGRAW78B Agrawala, A.K. & Mohr, J.M., "A Comparison of the Workload on Several Computer Systems," 1978 CMG Conference; December, 1978, San Francisco, California.

ANDER79 E. Anderson, "A Method for the Estimation of Resource Queueing Models," 1979 CMG Conference; December, 1979, Dallas, Texas.

ARTIS76 Artis, H.P., "A Technique for Determining the Capacity of A Computer System," 12th CPEUG Conference; November, 1976, Washington, D.C.

ARTIS78 Artis, H. P., "Capacity Planning for MVS Systems," 1978 CMG Conference; December, 1978, San Francisco, California.

ARTIS79 Artis, H.P., "Methodology for Establishing Resource Oriented Job Class Structure," 1979 CMG Conference; December, 1979, Dallas, Texas.

BROCK82 Brocklebank, J.C., "Analyzing Computer Workload Data Using SAS," 1982 CMG Conference; December, 1982, San Diego, California.

CLEME85 Clements, R. & Kolence, K.W., "Building Workload Profiles to Estimate Practical CPU Power," 1985 CMG Conference; December, 1985, Dallas, Texas.

DAVID79 Davidson, C.M., "Using Software Physics to Evaluate Utilization in a MultiComputer System," 1st ICCCM; April, 1979, Washington, D.C.

DAVIS81 Davis, L.E. & Lo, T.L., "The Evaluation of Methods for the Estimation of System Resource Usages," 1981 CMG Conference; December, 1981, New Orleans, Louisiana.

DOMAN85 Domanski, B., "IMS Workload Characterization: What Comes After Clustering," 1985 CMG Conference; December, 1985, Dallas, Texas.

DUJMO80 Dujmovic, J.J., "Workload Characterization, Benchmark and the Concept of Total Resources Consumption," The 2nd ICCCM; April, 1980, San Francisco, California.

ELMS80 Elms, C.M., "Clustering - One Method of Workload Characterization," 2nd ICCCM; April, 1980, San Francisco, California.

FRIED83 Friedman, E.M., "Workload Characterization for Subsystems: A Case Study for CICS/VS," 1983 CMG Conference; December, 1983, Washington, D.C.

FRIED84 Friedman, E.M., & Rosenberg, J.L., "Workload Characterization for CSCI/VS: the Modeling Implication," 1984 CMG Conference; December, 1984, San Francisco, California.

GILMO82 Gilmore, M., "Characterizing the Performance of A Distributed Processing Product," 1982 CMG Conference; December, 1982, San Diego, California.

GREEN76 Greenacre, G.R., "Applied Software Physics," 1976 CMG Conference; December, 1976, Atlanta, Georgia.

HARTR79 Hartrum, T.C. & Thompson, J.W., "The Application of Clustering Techniques to Computer Performance and Modeling," The 15th CPEUG Conference; October, 1979, San Diego, California.

HOFFM77 Hoffman, J.M., "Software Physics and the SARA System," 1977 CMG Conference; December, 1977, Washington, D.C.

HUGES78 Huges, H.D., "Workload Characterization of Computer Systems," 1978 CMG Conference; December, 1978, San Francisco, California.

HUGES80 Huges, H.D., "A Study of a Procedure for Reducing the Feature Set of Workload Data," 1980 CMG Conference; December, 1980, Boston, Massachusetts.

KAZLA83 Kazlauski, F.A., "Benchmark & Conversion Tool: Test Data Reduction Program," 1983 CMG Conference; December, 1983, Washington, D.C.

KOLEN79 Kolence, K.W., "CPU Power Analysis: Theory and Practice," 1st ICCCM; April, 1979, Washington, D.C.

KOVAC81 Kovach, R. P., "I/O Power Analysis and Reporting," 3rd ICCCM; April, 1981, Chicago, Illinois.

LEE83 Lee, B.K., "Workload Characterization of IMS Using Cluster Analysis," 1983 CMG Conference; December, 1983, Washington, D.C.

LEVY81 Levy, A.I., "A Case Study of Workload Characterization and Configuration Planning," 1981 CMG Conference; December, 1981, New Orleans, Louisiana.

MAEDA83 Maeda, A.R.T., "The Software Physics Update," 5th ICCCM; April, 1983, New Orleans, Louisiana.

MAMRA77 Mamrak, S.A. & Amer, P.D., "A Feature Selection Tool for Workload Characterization," 1977 CMG Conference; December, 1977, Washington, D.C.

MERLO83 Merlo, S., "Workload Classification Problems," 5th ICCCM; April, 1983, New Orleans, Louisiana.

MOHR79 Mohr, J.M., "The Time Varying Nature of Computer Workload," 1979 CMG Conference; December, 1979, Dallas, Texas.

MOHR80A Mohr, J.M., "A Survey of Available Workload Characterization Techniques," 1980 CMG Conference; December, 1980, Boston, Massachusetts.

MOHR80B Mohr, J.M. & Penansky, S.G., "A Framework for Projecting the ADP Workload," 1980 CMG Conference; December, 1980, Boston, Massachusetts.

MOHR81 Mohr, J.M. & Penansky, S.G., "A Forecast Oriented Workload Characterization Methodology," 3rd ICCCM; April, 1981, Chicago, Illinois.

PILCH84 Pilch, J., "Modeling Benchmark Workload," 1984 CMG Conference; December, 1984, San Francisco, California.

PRICH81 Prichard, E.L., "Workload Types and capacity Management Data Requirements," 3rd ICCCM; April, 1981, Chicago, Illinois.

REDDI76 Reddington, D.M., "DP Authority Workload History, Characterization and Forecasting Description," 1976 CMG Conference; December, 1976, Atlanta, Georgia.

SENSA79 Sensabaugh, S.L., "A Systematic Approach to the Support of Software Physics," 1st ICCCM; April, 1979, Washington, D.C.

SMITH81 Smith, B.J., "I/O Subsystem Workloads: Measurements and Modeling," 1981 CMG Conference; December, 1981, New Orleans, Louisiana.

VINCE84 Vince, N., "Clustering Techniques and Their Practical Application," 1984 ECOMA Conference; October, 1984, Munich, West Germany.

WIGHT81 Wight, A.S., "Cluster Analysis for Characterizing Computer System Workloads: Panacea or Pandora," 1981 CMG Conference December, 1981, New Orleans, Louisiana.

YEN83 Yen, E. H., "The Importance of Workload Definition in Capacity Planning and Performance," 5th ICCCM; April, 1983, New Orleans, Louisiana.

II. WORKLOAD FORECASTING

ALLEN82 Allen. L.E., "How to Obtain Accurate Workload Forecasts From Users," 1982 CMG Conference; December, 1982, San Diego, California.

APPLE83 Applegate, T.L., "Decision Roles for Forecast Tracking and Control: Living With Your Forecast as Time Goes By," 1983 CMG Conference; December, 1983, San Francisco, California.

ARTIS80 Artis, H.P. & Boast, D.R., "Estimating Latent Demand for Random Arrival Batch Workloads: A Case Study," 2nd ICCCM; April, 1980, San Francisco, California.

BIASI85 Biasi, O., "Workload Forecasting Methodology," 7th ICCCM; April, 1985, San Francisco, California.

DEAGR84 De Agro, D. & Preston, S., "The Linear Projection Model: An Event Driven Forecasting Model," 1984 CMG Conference; December, 1984, San Francisco, California.

FEBIS81 Febish, G.J., "Converting Business Plans to DP Workload Forecasts," 3rd ICCCM; April, 1981, Chicago, Illinois.

GRANT81 Grant, J.S., "Peak Workload Analysis and Identification," 3rd ICCCM; April, 1981, Chicago, Illinois.

GODWI85 Godwin, W. & Suhler, W., "A timing Estimation Method for Large System Software Development," 1985 CMG Conference; December, 1985, Dallas, Texas.

HOFFM82 Hoffman, L.L., "Workload Forecasting Using Econometric Time Series Analysis," 1982 CMG Conference; December, 1982, San Diego, California.

KULP78 Kulp, R.W. & Melendez, K., "An Application of the Time Series in Computer Performance Evaluation," 14th CPEUG Conference; October, 1978, Boston, Massachusetts.

LINDE80 Linde, S. & Morgan, L., "Workload Forecasting for the Shuttle Mission Simulator Computer Complex at Johnson's Space Center," 1980 CMG Conference; December, 1980, Boston, Massachusetts.

LIU85 Liu, M., "Using NFU Dependency Analysis in Business Oriented Forecasting of Workload Growth," 7th ICCCM; April, 1985, San Francisco, California.

LO80 Lo, T. L., "Computer Workload Forecasting Techniques: A Tutorial," 2nd ICCCM; April, 1980, San Francisco, California.

LO86 Lo, T.L. & Elias, J.P., "Workload Forecasting Using NFU: A Capacity Analyst's Perspective," Submit to 1986 CMG Conference; December, 1986, Las Vegas, Nevada.

LUIST83 Luistro, F.M., "The Cone Theory as Applied to Computer Workload Forecasting," 1983 CMG Conference; December, 1983, San Francisco, California.

MACKI78 MacKinder, C.M., "A Statistical Approach to Resource Control in a Time-sharing System," 14th CPEUG Conference; October, 1978, Boston, Massachusetts.

MAGER79 Magers, J.C. & Fischer, R.C., "Capacity Planning & Long Range Forecasting," 1st ICCCM; April, 1979, Washington, D.C.

MCNEE79 McNeece, J.E., "Computer Workload Forecasting," 15th CPEUG Conference; October 1979, San Diego, California.

NIEDZ83 Diedzielski, V.P. & Cecchi, W.J., "Workload Analysis and Forecasting: A case Study at BKB Corporation," 5th ICCCM; April, 1983, New Orleans, Louisiana.

PERLM79 Perlman, W., "Data Processing Workload Forecasting," 1979 CMG Conference; December, 1979, Dallas, Texas.

REED81 Reed, M.L., "Managing DASD Performance to Satisfy Workload Requirements," 1981 CMG Conference; December, 1981,New Orleans, Louisiana.

SANRA79 Sarna, D.Y., "Forecasting Computer Resource Utilizations Using Key Volume Indicators," 1979 AFIPS Conference Proc. Volume 48.

SHERK84 Sherkow, A.M., "Mainframe Computer Facility Usage Evaluation and Future Requirements Recommendation," 1984 CMG Conference; December, 1984, San Francisco, California.

WAGGO84 Waggoner, W.W., "Effective database Design: A Key to Integrating Business and DP Plans," 6th ICCCM; April, 1984, Washington, D.C.

WANDZ84 Wandzilak, J., "Problems Facing Workload Forecasting," 1984 CMG Conference; December, 1984, San Francisco, California.

WEINT83 Weintraub, M., "Predicting Computer Performance With CMF," 11th ECOMA Conference; October, 1983, Copenhagen, Denmark.

WILLI80 Williams, B., "Capacity Planning through Reduced Scale Implementation of Software Physics," 2nd ICCCM; April, 1980, San Francisco, California.

YEN85 Yen, K. "Projecting CPU Capacity Requirements - A Simple Approach," 1985 CMG Conference; December, 1985, Dallas, TX.

III. WORKLOAD CONTROL

ALLEN83 Allen, L.E., "A Decision Support System for the MIS Executives," 5th ICCCM; April, 1983, New Orleans, Louisiana.

ARTIS78 Artis, H.P., "Capacity Planning for MVS Computer Systems," 1978 CMG Conference; December,1978, San Francisco, California

BECKE85 Becker, G., "Capacity Planning for Applications Still Under Development," 1985 CMG Conference; December, 1985, Dallas, Texas.

DITHM83 Dithmar, H., "Key Ingredients for Successful Capacity Management,." 5th ICCCM; April, 1983, New Orleans, Louisiana.

HAEBI80 Haebig, D.G., "Managing Service Levels Through Workload Scheduling," 2nd ICCCM; April, 1980, San Francisco, California

HARTR83 Hartrum, T.C., "The Application of Multivariate Statistical Techniques to Computer Performance Evaluation Using Simulated Data," 19th CPEUG Conference; October, 1983, San Francisco, California.

HINTZ83 Hintze, J., "Multiple Regression Analysis Using SAS," 1983 CMG Conference; December, 1983, Washington, D.C.

HOWAR80 Howard, P.C., "Planning Capacity to meet User Services," 1980 CMG Conference; December, 1980, Boston, Massachusetts.

INGRA85 Ingrassia, F., "Modeling the Performance of New On-line System," 1985 CMG Conference; December, 1985, Dallas, Texas.

JACKS81 Jackson, R.M., "Workload Characterization and Capacity Planning at a Large IBM Installation," 1981 CMG Conference; December, 1981, New Orleans, Louisiana.

KOLEN76 Kolence, K.W., "The Meaning of Computer Measurement: An Introduction to Software Physics," The Institute for Software Engineering, 1976. Palo Alto, California

KOLEN77 Kolence, K.W., "Software Physics: A Toturial," 1977 CMG Conference; December, 1977, Washington, D.C.

LO84 LO, T.L., "Planning, Service, and Cost Considerations for Management," 12th ECOMA Conference; October, 1984, Munich, West Germany.

LO85 LO, T.L., "Capacity Management: A multidimensional Process," 1985 CMG Conference; December, 1985, Dallas, Texas.

MULLE85 Mullen, J. W., "Capacity Planning: Basic Elements for the Process," 1985 CMG Conference; December, 1985, Dallas, Texas.

ORCHA85 Orachard, R.A. & Domanski, B.E., "System Performance Management and Capacity Planning," 1985 CMG Conference; December, 1985, Dallas, Texas.

PEDRI78 Pedriana, F.L., "Some Practical Applications of SAS to Capacity Planning," 1978 CMG Conference; December, 1978, San Francisco, California.

PHIPP81 Phipps, L.W. & Schiavone, T.H., "Capacity Management Information System Via SAS," 3rd ICCCM; April, 1981, Chicago, Illinois.

POLLA84 Pollack, T., "Capacity management: A Case Study," 1984 CMG Conference; December, 1984, San Francisco, California.

SIMPS83 Simpson-Felix, S., "Implementing an Analytic Model for CICS System," 1983 CMG Conference; December, 1983, Washington, D.C.

The Evolution of Software Performance Engineering: A Survey

Connie U. Smith, Ph.D.
Performance Engineering Services Division
L & S Computer Technology, Inc.
PO Box 9802-120
Austin, TX 78766
(505) 988-3811

Abstract

This paper surveys the approaches to software performance from the 1960's to the present. It points out the breakthroughs leading to the software performance engineering approach (SPE), a comprehensive methodology for constructing software to meet performance goals. SPE advocates building performance into the software architecture (as opposed to tuning code). The paper summarizes the concepts, methods, models, tools, and use of software performance engineering and suggests future trends in each of the areas.

Introduction

Software performance engineering (SPE) is a method for constructing software systems to meet responsiveness goals. With it, one models software requirements and designs, and evaluates whether predicted performance metrics meet the specified goals. If not, alternatives are proposed and assessed. The process continues through the detailed design, coding, and testing stages to develop more precise models of the software and its predicted performance.

Performance refers to the response time or throughput as seen by the users, that is, its responsiveness. Real-time command and control systems must be responsive to be correct. Shneiderman observed that responsiveness is an important human interface factor in all systems [SHN79]. Since it limits the amount of work that can be processed, it also determines a system's effectiveness.

SPE focuses on software development, but it is also an important part of a capacity management program [BER84]. Accurate planning requires data on new applications and data on the effect on resource requirements of growth in current work; SPE models provide both. Three important factors that capacity management should address are in Figure 1.

A passive capacity planning program anticipates growth and reacts to it. The approach advocated in Figure 1 is an *active* capacity *management* program that controls configuration requirements. Growth in workload demands may be inevitable, but growth due to resource requirements of software can be controlled through SPE.

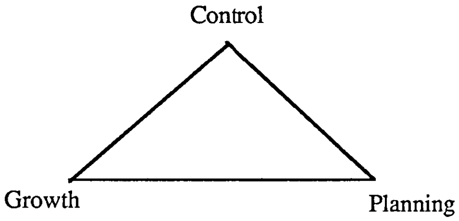


Figure 1. Capacity Management Functions

Many aspects of capacity management are dependent on the execution environment. Other papers in this session are specifically oriented to IBM environments. SPE, though, works in all computer environments. System dependent characteristics are incorporated into the SPE models.

This introduction claims that SPE is a viable methodology. The next section reviews its evolution from the first applications, through the modeling breakthroughs to the tools and methodology that established its viability. Then, recent advances are described, and finally, the future of SPE is predicted.

Early approaches

Performance was typically considered in the early years of computing. The space and time required by programs had to be carefully managed in order to fit them on small machines. The hardware grew but, rather than eliminating performance problems, it made larger, more complex software feasible and programs grew into systems of programs. Some of the software systems had strict performance requirements, such as flight control systems and other embedded systems. Performance modeling and assessment for these systems was expensive and labor-intensive. They used detailed simulation models, consequently creating and solving them was time-consuming. Updating the models to reflect the current state of evolving software systems was also problematic. Thus, the modeling and assessment was cost-effective only for systems with strict performance requirements.

Other systems adopted the "fix it later" methodology. It advocates concentrating on software correctness, deferring performance considerations to the integration testing phase and (if performance problems are detected then) procuring additional hardware or "tuning" the software to correct them. The results have been acceptable until recently. The following figure explains why it no longer works:

Response time was seldom a problem for batch systems. When on-line systems were first introduced, they had modest resource demands, and there were fewer users and other systems competing for the resources. Since then the number of users and on-line systems have grown substantially. The systems now have more ambitious functions, rely on large data bases, and use fourth generation languages; thus, the demand has significantly increased. As Figure 2 shows, an increase in demand on the right end of the curve causes a much higher increase in response time than one on the left. So the likelihood of performance problems was low in the past when demand was low, but they are more likely now.

Unforunately the "fix it later" approach is still used by many system developers. The approach has many disadvantages: it takes time to procure and install hardware and to tune software; testing must be repeated after code changes; and an interim period of poor performance leaves a negative impression with users long after it is corrected [PEAR82]. The rationale for "fix it later" is to save development time, cost and maintenance cost. The

CH2345-7/86/0000/0778$01.00 © 1986 IEEE

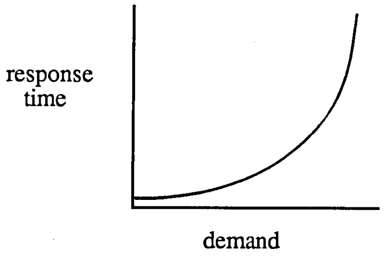


Figure 2. Effect of demand on response time

savings will not be realized, however, if initial performance is unsatisfactory because of the additional time and cost for tuning and maintenance (introduced by efficiency "tricks" for tuning and code entropy resulting from the changes)

Table 1 summarizes the premises that led to "fix it later," the current reality, and the consequences of unacceptable performance. It was a viable approach for batch software, but it is inappropriate for large on-line systems.

Modeling Breakthroughs

In 1971, Buzen proposed modeling systems with queueing network models and published efficient algorithms for solving some important models [BUZ71]. The models are an abstraction of the computer systems they model, so they are easier to create than general purpose simulation models. Since they are solved analytically, they can be used interactively. Since then, many advances have been made in modeling computer systems with queueing networks, faster solution techniques, and accurate approximation techniques [LAZ84, SAU81]. They are commonly used in capacity planning to model large computer systems.

Figure 3 illustrates their use: the model is constructed from information on the computer system configuration and measurements of resource requirements for each of the workloads modeled. The model is solved and the resulting performance metrics (response time, throughput, resource utilization, etc.) are compared to measured performance. The model is calibrated to the computer system. Then, it is used to study the effect of increases in workload and resource demands, and of configuration changes.

The models were used primarily for capacity planning. For SPE they were sometimes used for feasibility analysis: request arrivals and resource requirements were estimated and the results assessed. More precise models were infeasible since the software could not be measured until it was implemented.

The second SPE modeling breakthrough was the introduction of analytical models for software [BOO79, SAN78, SMI79]. With them, software execution is modeled, estimates of resource requirements are made, and performance metrics are calculated. They yield an approximate value for best, worst, or average resource requirements. They can also be used to derive an estimate for response time; they can detect response time problems, but since they do not model resource contention they do not yield precise values for predicted response time.

The third SPE modeling breakthrough was combining the anlaytic software models with the queueing network system models to more precisely model execution characteristics [BGS83,SMI80]. Figure 4 shows

Table 1. "Fix it Later" Approach?

Premises:	**Reality:**	**Consequences (of unacceptable performance):**
Performance problems are rare	The number of on-line software systems, their size, and their complexity have increased; many large systems cannot be used initially due to performance problems	Requires tuning and/or hardware
It's too expensive to build high-performance software	Data on time and cost of building-in performance is out of date; methodologies and tools have dramatically reduced costs	Testing is slower so cost is higher; maintenance costs (for "tuned" code) are higher
Tuning can be done later	Problems are usually due to fundamental architecture or design problems rather than inefficient coding	Code tuning yields modest improvements; large gains require major revisions. Very expensive (often infeasible) to change fundamental design choices
Efficiency implies "tricky" code	Acceptable performance is required; it can be designed-in early	"Tricky" code may be the only option for achieving goals late in life cycle
Hardware is fast and inexpensive	No installations have unlimited hardware budgets. Advanced planning and justification are needed for procurements. Software demands may exceed all hardware capabilities	Lose control of equipment purchase and maintenance budgets

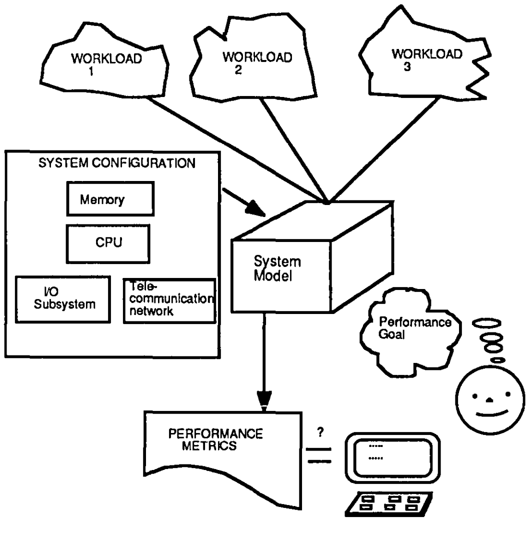


Figure 3. Conventional Performance Modeling

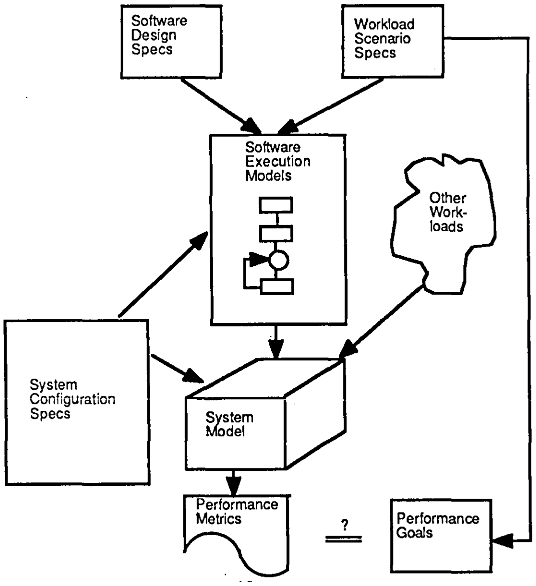


Figure 4. SPE Modeling

the combination of the conventional model used for capacity planning and the software model for systems under development. They not only model more precisely the execution, but also show the effect of new software on existing work and on resource utilization, and identify computer device bottlenecks and the parts of the new software with high use of bottleneck devices.

By 1980, the modeling power was established and many software tools were available [BGS82, BGS83, IRA75, QSP82] (note: the products and publications have been updated since 1980). Thus, it became cost-effective to model large software systems early in their development.

SPE Methodology

Early experience with a large system under development confirmed that sufficient data could be collected early in development to predict performance bottlenecks [SMI82]. Unfortunately, despite the predictions, the system design was not modified to remove them and upon implementation (approximately one year later) performance in those areas was a serious problem, as predicted. The modeling problems were resolved, but that was not enough to prevent problems.

The SPE methodology in Figure 5 was proposed [SMI81] and later updated [SMI84]. Key parts of the methodology are methods for collecting data early in software development, and critical success factors (Figure 6) to ensure SPE success. The methodology also addresses compatibility with software engineering methods, what is done, when, by whom, and other organizational issues. SPE remains an art--the problems are not technical, they require human communication and interaction techniques that can be learned.

SPE Use

Lately, many papers have been written reporting success in applying SPE. A special issue of the *Computer Measurement Group Transactions* on experience with SPE appeared in fall 1985 [CMG85]. It contains an extensive bibliography on SPE reporting experiences, methodologies, modeling techniques, tools, software measurement techniques, and program improvement techniques. The US Army Information Systems Engineering Command developed a performance engineering handbook for their software engineers. Many conferences now include sessions on software performance engineering; and seminars are offered to teach SPE techniques.

Recent Advances

The purpose of SPE is to support the development of software systems that will be responsive to users when they are initially implemented. Performance problems have been detected and corrected early, before implementation. Since ideas and designs must be formulated before they can be modeled, problems still tend to be present in the initial formulation. To prevent this, a set of formal, general principles for performance oriented design was developed [SMI86]. Software architects who are experts in formulating requirements and designs for large, high-performance systems use intuition to develop their systems. The general principles formalize that expert knowledge. They can, thus, transfer the expert's intuition, developed through years of experience, to software architects with less experience in building responsive systems.

With this advance, the evolution of SPE can be depicted with the chart in Figure 7.

Other recent advances offer incremental improvements to the areas in Figure 7.

The models have evolved. Complex execution characteristics (such as memory usage, locking resources, parallel execution, etc.) cannot be solved easily with analytic queueing models. It is easier to use a similar model structure to the queueing network models, but use simulation-based solution algorithms that execute much quicker than the old, detailed simulation models [IRA83]. Petri net models have also been used to model parallel execution of software and hardware [BAL85, SMI85].

The user interfaces of tools have improved; graphics devices provide visual feedback to designers [FRA85, IRA85,LS86]. SPE has been extended to new applications such as point of sale systems [AND84], and software/hardware codesign [FRA85].

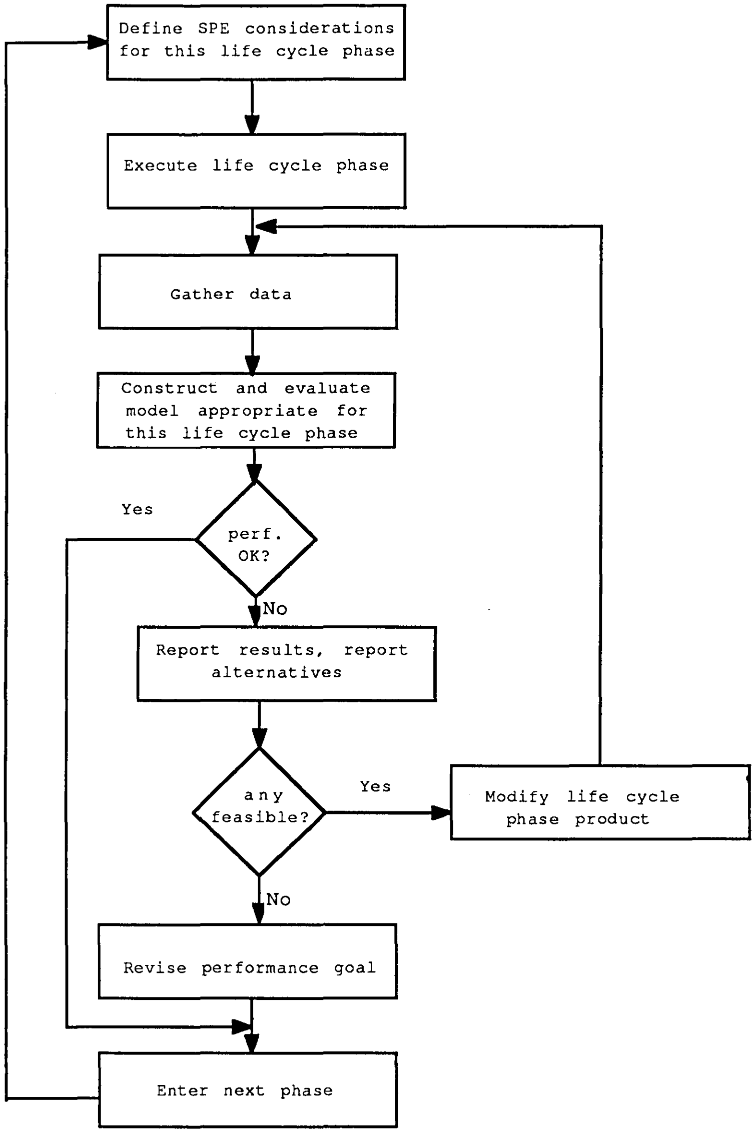

Define SPE considerations for this life cycle phase
Execute life cycle phase
Gather data
Construct and evaluate model appropriate for this life cycle phase
perf. OK?
Yes
No
Report results, report alternatives
any feasible?
Yes
Modify life cycle phase product
No
Revise performance goal
Enter next phase

Figure 5. P. E. Methodology

Project management:

commitment -react to significant results

scheduling - include sufficient time for SPE (and
for possible modifications)

control - performance consciousness

credibility of results (analyst training and backing)

Data requirements:

representative workload scenarios

precise estimates of critical resource usage

best and worst case results:
focus attention on problems rather than models
for desirability evaluation

Organizational issues:

outside expert or member of design team?

cooperative effort

ongoing CPE provides better data

Performance analyst responsibilities:

timely results

previous experience (chicken & egg problem?)

quantitative data for alternatives

project visibility

Figure 6. Critical Success Factors for Effective SPE [SMI84]

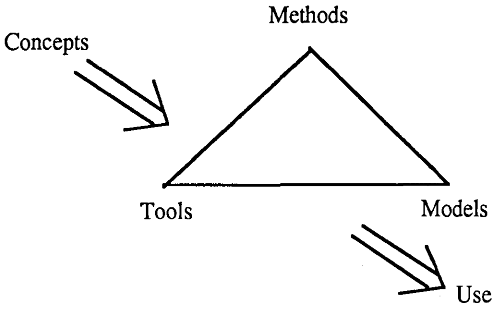


Figure 7. SPE Evolution

The Future of SPE

Advances in SPE are likely to continue at a rapid rate. Table 2 summarizes some predicted advances in each of the five areas identified in Figure 7, and adds a new area: verification and validation of SPE models.

There are many exciting challenges for researchers. Most require skills in multiple areas (e.g., software engineering and performance modeling, computer architecture and software engineering, etc.). For practitioners, many systems under development need SPE. Each will provide new modeling challenges and many learning opportunities. The support tools available in the future will be fun to use and will reduce the amount of time spent "pushing numbers."

Table 2. Future Directions

Area	Improvements
Concepts	General software design principles to incorporate other quality attributes such as reliability, testability, maintainability, etc.
Methods	Formal integration of SPE and software engineering
	Models, tools, and methods for assessing the other quality attributes early in development
	Official organization positions for "performance engineers" with continuing education to improve the human interaction skills
Models	New models to evaluate extensive parallel and distributed processing
	Models for new computer architectures
	Models of transient behavior to study periodic behavior or unusual execution characteristics
Tools	Computer-aided design (CAD) tools to support the designer in automatically performing design assessments while formulating designs, and to integrate the models with the design evolution
	Expert systems to automatically detect and suggest design improvements
	Effective graphical techniques for reporting SPE results
	Software measurement tools to capture, reduce, interpret, and report data at a level of detail appropriate for designers
	A software performance data base to store evolutionary design and model data and support queries against it
Use	Rapid increase in application to new, large systems
	More literature documenting experience with SPE
	New domains, such as predicting performance of VLSI chips early in design
Verification and Validation	Techniques for including instrumentation in software designs
	Techniques for calibrating software models, automatically relating predictions to measurements, and studying discrepancies

References

[AND84] Gordon E. Anderson, "The Coordinated Use of Five Performance Evaluation Methodologies," *CACM*, 27,2, Feb. 1984, 119-125.

[BAL85] G. Balbo, S.C. Bruell, S. Ghanta, "Modeling Priority Schemes," *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Austin, Aug. 1985.

[BER84] Margaret E. Berry, "The Best of Both Worlds: An Integrated Approach to Capacity Planning and Software Performance Engineering," *Proc. Computer Measurement Group Conf. XV*, San Francisco, Dec. 1984, 462-466.

[BGS82] BEST/1 Modeling Package, BGS Systems, Inc., Waltham, MA, 1982.

[BGS83] CRYSTAL Modeling Package, BGS Systems, Inc., Waltham, MA, 1983.

[BOO79] T. L. Booth, "Use of Computation Structure Models to Measure Computation Performance," *Proc. Conference on Simulation, Measurement and Modeling of Computer Systems*, Boulder, August 1979.

[BUZ71] J. P. Buzen, "Queueing Network Models of Multiprogramming," Ph.D. Thesis, HarvardUniversity, Cambridge, MA, 1971.

[CMG85] "Special Issue on Software Performance Engineering," *Computer Measurement Group Transactions*, 49, Sept. 1985.

[FRA85] G.A. Frank, C. U. Smith, J.L. Cuadrado, "Software/Hardware Codesign with An Architecture Design and Assessment System," *Proc. Design Automation Conference*, Las Vegas, 1985.

[IRA75] User's Manual for the CADS System, Information Research Associates, Augstin, TX, 1975.

[IRA83] Performance Analyst Workbench System (PAWS), Information Research Associates, Austin, TX, 1983.

[IRA85] GPSIM: Graphical Programming for Simulation, Information Research Associates, Austin, TX, 1985.

[LAZ84] E.D. Lazowska, J. Zahorjan, G.S. Graham, K.C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice Hall, Inc., Englewood Cliffs, 1984.

[LS86] Graphical Queueing Network Analysis (GQUE) User's Manual, L & S Computer Technology, Inc., Austin, TX, 1986.

[PEA82] S.W. Pearson, Tutorial on Information Systems Effectiveness, 18 Computer Performance Evaluation Users Group, Washington, DC, Oct. 1982.

[QSP82] A Modeling and Analysis Package (MAP), Quantitative System Performance, Seattle, WA, 1982.

[SAN78] J.W. Sanguinetti, "A Formal Technique for Analyzing the Performance of Complex Systems," *Proc. Computer Performance Evaluation Users Group 14*, Boston, October 1978.

[SAU81] C.H. Sauer, K.M. Chandy, *Computer Systems Performance Modeling*, Prentice Hall, 1981.

[SHN79] B. Shneiderman, "Human Factors Experiments in Designing Interactive Systems," *IEEE Computer*, 12,12, Dec. 1979, 9-20.

[SMI79] C.U. Smith, J.C. Browne, "Performance Specifications and Analysis of Software Designs," *Proc. Conference on Simulation Measurement and Modeling of Computer Systems*, Boulder, August 1979.

[SMI80] Connie U. Smith, J.C. Browne, "Aspects of Software Design Analysis: Concurrency and Blocking," *Proc. Performance 80* also in *ACM Performance Evaluation Review*, 9,2, Summer 1980.

[SMI81] Connie U. Smith, "Software Performance Engineering," *Proc. Computer Measurement Group Conference XII*, Dec. 1981, 5-14.

[SMI82] Connie U. Smith, J.C. Browne, "Performance Engineering of Software Systems: A Case Study, *Proc. National Computer Conference*, Vol. 15, Houston, June 1982, 217-224.

[SMI84] Connie U. Smith, "Effective Implementation of Software Performance Engineering," *Proc. European Computer Measurement Association 12*, Munich, Oct. 1984, 241-245.

[SMI85] Connie U. Smith, "Robust Models for the Performance Evaluation of Software/Hardware Designs," *Proc. Int. Conf. Timed Petri Nets*, Torino, July 1985, pp. 172-180.

[SMI86] Connie U. Smith, "Applying Synthesis Principles to Create Responsive Software Systems, " Duke University Technical Report CS 1986-11, submitted for publication, 1986.

COMPUTER DESIGN ARENA

Fault-tolerant Computing

TRACK CHAIR: Prof. John Meyer
University of Michigan

VLSI Design and Test: Theory and Practice

TRACK CHAIR: Mr. Jerome Kurtzberg
IBM T. J. Watson Research Center

Computer Graphics

TRACK CHAIR: Prof. Michael Wozny
Rensselaer Polytechnic Institute

PERFORMABILITY ANALYSIS OF OPERATION MODES OF CONFIGURABLE DUPLEX SYSTEMS

Balakrishna R. Iyer, Daniel M. Dias and Philip S. Yu

IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598

Abstract

Configurable duplex system structures, like the IBM 3084, are capable of operating as one single system (single image mode) or as two independent systems (partitioned mode). The user installation has a choice of operation modes. The single image mode of operation is more tolerant to hardware failures because of component redundancies, but on operating system failure all processors are lost. In the partitioned mode the outage of any hardware component or operating system brings down the affected half of the duplex system. The problem is to select the mode of operation. Since multiple performance levels, derived from a model, are involved a composite performance reliability measure like performability is required. Areas based on the hardware and software failure rates are demarcated, where one of the operation modes is better than the other. The sensitivity of the choice of operation mode to the the goals of the installation (choosing on the basis of average or percentile) is also demonstrated. The use of a steady state performability measure instead of transient performability is shown to distort the choice of operation mode. Hence, simple to compute tight upper and lower bounds for the hard to compute transient performability are derived.

1. Introduction

The demand for higher system availability and greater processing power has been growing rapidly in various installations. A general trend of commercial computer system architectures is towards multiple processor systems with (a) fault tolerance and (b) horizontal growth as major design goals. Both goals can be achieved by replicating functional units and providing proper recovery software. In the IBM 308X processor family, some 3081 models are field upgradable from a two processor system into a four processor 3084 system. Boos (1985) describes an interesting feature of the 3084 system in that it can be configured either as a 4-way tightly coupled multi-processor system or as two separate 2-way tightly coupled multi-processor systems, each of which can be used to service half the total workload.

A simplex system consists of various functional units and operates under a single operating system. A configurable duplex system is obtained by taking two simplex systems and adding interconnections between their functional units. A configurable duplex system may be configured to operate in two different modes - PARTITIONED or SINGLE IMAGE. In the partitioned mode the duplex is configured as two separate (simplex) systems each with a single instance of each functional unit. In the single image mode the duplex runs as one system with dual copies of every functional unit. We consider the problem of selecting the operation mode for a configurable duplex system structure, like the IBM 3084. A study of different configurable duplex system structures can be found in Iyer (1985).

The user installation has a choice of operation modes. The single image mode of operation is more tolerant to hardware failures because of component redundancies, but on operating system failure all processors are lost. In the partitioned mode the outage of any hardware component or operating system brings down the affected half of the duplex system. The problem is to select the mode of operation based not only on the the available statistics about the failure rates of the system and the projected performance levels during periods of degraded operation but also based on the installation requirements. For instance, if system availability is a critical installation requirement, as in online ATM systems, then a 95'th percentile metric related to availability may be appropriate. In throughput oriented installations, like those doing check processing, average system performance may be the preferred metric.

Since multiple performance levels are involved reliability or availability analysis based on a binary valued function is inadequate. The composite performance reliability measure **performability** (Meyer (1980)), which can be used both as a percentile oriented metric and a throughput oriented metric, is required for the problem. Performability may be viewed as the total accumulated reward of a system during its mission time - the interval during which the system is required to give uninterrupted service. Examples of mission times are a) 8 a.m. to 5 p.m. for office systems, and b) the blast-off to earth landing time for a computer system aboard a space vehicle. The normalized capacity of a system is defined as the work per unit time (e.g. transaction rate in a database system) that the system can sustain during the system mission time normalized by the maximum work rate. Clearly the available capacity at any instant depends upon the number of functioning processors at that instant. Normalized capacity is a special case of performability. We refer to the work of Furchgott and Meyer (1984), Kulkarni et. al. (1984) and Goyal and Tantawi (1984) for recent investigations into applications and analysis of performability.

The simplex and duplex system structures and the two duplex system operation modes are described in Section 2. In Section 3, the partitioned mode operation and single image mode operation are compared using performability measures. This comparison is found to be sensitive to the metric chosen (e.g average or tail of the distribution of performability) and also to other system parameters. Usage of steady state performability instead of transient is shown to lead to distortion of choice between single image and partitioned modes. In Section 4, we focus on the difference between transient and steady state performability measures and derive easily computable analytical upper and lower bounds for their difference. An approximation is also provided. The bounds themselves are shown to be the basis of good approximations to compute transient average performability starting from the steady state performability measure. Conclusions are drawn in Section 5.

2. System Structures

2.1. Simplex System Structure

The computer system structure envisioned in this paper is depicted in Figure 2.1(a). The configuration is similar to the configuration of the IBM 3081 processor complex described by Pittler et. al. (1982) and

CH2345-7/86/0000/0785$01.00 © 1986 IEEE

Gustafson and Sparacio (1982). The processor (P) executes instructions that reside in the memory (M). Memory access is coordinated by the memory controller (MC). Both the memory controller and processor are connected through the system controller (SC) which serves the purpose of a switch and may be used as a controller for a processor buffer (cache). The system controller is also attached to the external data controller (X) which contains the intelligence required to address the control units which communicate with the disks, tapes, printers, terminals and other peripheral devices for data. Hardware functional units of the above kind are typically subject to frequent transient failures. Each hardware unit described above is assumed to have comprehensive error detection, correction and fault isolation capabilities similar to those described by Bossen and Hsiao (1980,1982) and Tendolkar and Swann (1982). The processor.controller (PC) coordinates such retry activity and maintains a log of functional unit errors. It interfaces to each functional unit through the interface sensor sets (I) over links that are typically of much lower bandwidth than those used between the other hardware units for data transfer. Reilly et. al. (1982) describe the processor controller for the IBM 3081. Operating system software (SW), running on top of the hardware configuration, is considered as another component. We will use the terms component and functional unit interchangeably.

The simplex system is available only if all the hardware and software components of the system are operational. (Discussion of power and thermal unit failures is beyond the scope of this paper.) Basic system availability is pictured by a coherence diagram in Figure 2.1(b). The system is considered to be available if there is an end-to-end path in the coherence diagram that contains non-faulty components. Memory is assumed to be protected by error correcting circuitry and most memory failures are masked out. Remaining memory failures and memory controller failures are lumped together as memory controller failures. There are two such types of failures considered. The first type consists of failures for which the system may be successfully re-IPL'ed (IPL is used to abbreviate the 'initial program load' function) after bypassing the failed portion of memory or memory controller. The system now operates with less memory than before. It is assumed that such loss of memory is usually small and therefore the impact on performance may be ignored. The second type of failure is more serious and knocks off the entire memory until a hardware repair action is taken on the memory modules or controller. The ratio of failures of the first type to the second is assumed to be $r/(1-r)$. Both types of failures cause system outage. Under the assumption that the system is subject only to single component faults, Figure 2.1(c) gives a transition diagram representation of the system availability process. L_P , L_{SC} , L_X , L_{MC} , L_{PC} and L_I are assumed to be the permanent hardware failure rates of the processor, system controller, external data controller, memory controller, processor controller, and the interface sensor set, respectively. The average repair time for such a functional unit failure is assumed to be $1/M_R$. L_{SW} is the failure rate of the operating system running on the simplex system induced by (a) program logic errors and (b) transient or intermittent hardware failure. These do not include operating system failures attributable to its inadequacy in dealing with permanent functional unit failures - these are accounted for by the coverage probabilities. Operating system failure is recovered by a system re-IPL which takes time $1/M_I$. Link failures are assumed to be negligible and are not modelled. There are two modes of failures of the interface sensor set modelled here. In the first we consider the loss of the sensor link to a single component in the system. The failure is incorporated into the individual component failure rate. The second mode of failure, referred to as the common mode of failure, models the loss of all sensor lines from the sensor set. This is modelled by the failure rate L_I.

2.2. Duplex System Structures

The duplex systems may be run in either partitioned mode or single image mode, described below.

Partitioned Mode Operation:

In the partitioned operation mode, depicted in Figure 2.2.1(a), there is no coupling of the two systems running on the two halves of the duplex. In a transaction processing environment, two separate database systems with disjoint sets of databases can be running on each half of the duplex. The advantage of this mode is that failures on one side do not affect the other side. Such failures include both hardware and operating system failure. The disadvantage, though, is that failure of any hardware component leads automatically to the outage of half the system. In Figure 2.2.1(b) these availability considerations are depicted in a coherence diagram. Recall that the system is considered to be available if there is an end-to-end path in the coherence diagram that contains only non-faulty components. Perfect fault recovery coverage is assumed for parallel pairs in the coherence diagrams. In the next section, the impact of imperfect recovery coverage is considered for parallel pairs. We pick throughput as the reward to signify performance levels in different configurations. Each state is associated with a single throughput value, thus ignoring the transient effects of on throughput immediately after a reconfiguration. For systems where the jobs to be processed take small time compared with the time between reconfigurations (e.g. transaction processing systems), our assumptions are realistic. Assuming a (normalized) reward rate of unity for a fully operational partitioned mode system (both systems running) the reward rate for partitioned mode during component failure (only one system running) becomes 0.5. We do not model occurrence of failures during the repair of another failure. Figure 2.2.1(c) is a transition diagram indicating state transitions on failures with the states labeled with the associated reward rates. The failure rates assumed in Figure 2.2.1(c) reflect the fact that there are two of each component in the duplex system architecture. Each operating system is assumed to have the the same failure rate L_{SW} as the operating system for the simplex system. System loss due to either software failures and partial memory failure may be recovered by a re-IPL. Other hardware failures require a hardware repair action.

Single Image Mode Operation:

In this mode of operation, two simplex systems are assumed to be connected through their system controller's with connectivity further enhanced by having each memory controller connect to both system controllers and by having each processor controller connect to both sets of interface circuits, as illustrated in Figure 2.2.3(a). This interconnect architecture is similar to the IBM 3084 interconnect architecture described by Boos (1985), where two tightly-coupled (i.e. dyadic) processors replace the single processor we have assumed for each P in Figure 2.2.3(a). The impact of component failure on this mode of operation is shown in Table 2.1. The rows in this table indicate the effect of component failure. The columns labelled P, S, M, X, show the number of processors, system controllers, memory systems and external data controllers that remain active for each type of component failure. In the last two columns of this table we indicate the transitions experienced due to covered and uncovered failures for each kind of component failure. For uncovered failures there is an outage. Re-IPL (average time $1/M_I$) restores the failed system to a level of degraded performance. A hardware repair action starts simultaneously. At the end of the repair action the system returns to a state of normal performance. We assume that the time for the repair action to complete after the end of the IPL action is given by $1/M_{R-I} = 1/M_R - 1/M_I$. The failure of memory or memory controller leads to an instantaneous loss of half the memory and is assumed to be catastrophic and causes system outage. For covered failures of memory the resultant outage is recovered by a re-IPL. For other component failures, degraded performance results until the end of the repair action (average repair time $1/M_R$). Markov transition diagrams in Figure 2.2.2(c) illustrates these transitions and changes in rewards, graphically.

We make the following additional observations about this mode of operation. (i) Only one operating system is run on the duplex system and hence system software failure leads to system outage for the duration of an IPL. The exact relationship between the operating system failure rate of the simplex machine and the duplex machine running in single image mode is unknown. On the duplex machine we have one operating system controlling twice the number of resources as in a simplex machine. In the absence of experimental evidence we assume for most of this paper that the software failure rate for single image mode operation is twice that for the operating system of the simplex machine. One parametric study varying the ratios between the software failure rate of the single image mode with that of the partitioned mode is conducted in the next section. (ii) Only one processor controller need be available for normal system operation. Since the processor controller is used mainly for error logging and retry operations, the effect of having two or one processor controllers does not have any effect on system performance. Hence it is not included in Table 2.1. (iii) Jobs are assumed to be picked up by any free processor for execution. It is in this way that the single image mode recovers from processor failure. (iv) We assume that there are dual access paths to all I/O devices - one path through each external data controller. Every I/O request is sent to both system controllers and from there to both external data controllers. The external data controller that can first satisfy the I/O request does so. Thus single image mode operation can continue in the face of an external data controller failure. (v) It is assumed that the software system has no capability to recover from a system controller failure. (vi) Both interface sensor sets are crucial to the operation of the system because failure of any one of them brings down half the coupled system, and since half the memory is lost we assume that there is a system outage. .Furthermore, on processor loss, memory loss or external data controller loss the system may be reconfigured leaving out only the lost component. Software failures of the operating system lead to complete system outage until the system is re-IPL'ed.

3 Performability Analysis

3.1 Methodology

Traditionally reliability and performance analysis of computer systems have been carried out separately. Computer system states have been traditionally classified into 'up' and 'down' states. In a reconfigurable duplex system there are a class of states in which the computer system is operational but not at peak performance, due to loss of some component. It is important to further refine the concept of availability to reflect more clearly the performance of the system in these states.

Meyer (1980) first defined performability in terms of the state of the system X_s (at time s). The state dependent function $f()$ defines the reward rate in state X_s. Y_t, the performability of the system over the mission time $(0,t)$ is defined as the distribution of the accumulated reward

$$Y_t = \int_{s=0}^{t} f(X_s)ds.$$

When the states space is discrete and finite and $X_s \in (0,1,2,..,N)$ then

$$Y_t = \sum_{i=0}^{N} f(i)\tau_i$$

where τ_i is the time spent in state i during the system mission time. We pick $f_{\max}$ to denote the largest reward in the system states and define normalized capacity as $Y_t/(t f_{\max})$, and steady state normalized capacity as $\lim_{t\to\infty} Y_t/(t f_{\max})$. If we let $f(i)$ denote the throughput of the system in state i then normalized capacity is the normalized work output by the system per unit time over the course of the system mission.

Both the mean $E(\frac{Y_t}{t})$ and the distribution $P(\frac{Y_t}{t} < \hat{y})$ metrics are of interest in this paper.

We quote the following result due to Iyer et. al. (1984) that sets forth an analytical method to compute the moments of the performability distribution. They assume that the time to failures and repair times are exponentially distributed. All vectors shall be understood to be column vectors. Let $\mathbf{m}_n(t)$ be the vector with entries $m_{i,n}(t)$ where $m_{i,n}(t) = E(Y_t^n \mid X_0 = i)$ is the n'th moment of performability starting from state i. Note that $m_{i,0}(t) = 1$. The n'th moment of performability is given by

$$\mathbf{m}_n(t) = \sum_{i=0}^{N}\sum_{j=0}^{n} \mathbf{v}_n(i,j) e^{\lambda_i t} t^j,$$

where, the λ_i 's are the eigenvalues of the generator matrix of the Markov process X_s and the vectors $\mathbf{v}_n(i,j)$ are computable by recursions.

Programs were developed to compute all n moments of performability based on the above result and used to generate transient average normalized capacity for the different system configurations described in Section 2. For the Markov models considered in this paper, the steady state normalized capacity is derived through the steady state probabilities derived by using balance equations and the normalization constraint.

No general closed form solution is known for the distribution of performability. Kulkarni et. al. (1985) have recently used a numerical Laplace transform inversion procedure to obtain the distribution of performability for Markovian models. Donatiello and Iyer (1985) have recently shown that even in the case of a simple two state Markov process the distribution of performability is composed of modified Bessel functions. We use simulation to generate the performability distribution. The simulation assumes that the time to failure is exponential and uses both the exponential and deterministic distribution for repair times.

3.2 Comparison of Operation Modes

In this section we investigate the choice of single image or partitioned operation modes. The choice is found to be sensitive to the hardware and software failure rates, and other factors. We compare the use of steady state measures as approximations to transient measures and show that conclusions drawn by using the former may lead to anomalies. This observation motivates the approximation technique for the transient average normalized capacity proposed in the next section.

Instead of differentiating operation modes on the basis of the reward rate during normal operation we focus on the difference in the impact of component failures on different system structures and modes of operation. System operation is quantified by the accumulated reward. The reward rate may signify the throughput of the system under a response time constraint or may reflect other performance related variables as described by Donatiello and Iyer (1984). The choice of the reward rates is a function of the computing system environment. For example, in a high availability environment a severe penalty may be the reward for down states.

The performance level of each operation mode is different for each state and can be derived from some lower level model. For example, in the Appendix a model is presented to evaluate the system throughput for a transaction processing environment. The impact of each type of component failures is captured in the modelling. To evaluate the effect of external data controller or system controller failure, IO paths to storage devices are explicitly modelled. In the partitioned mode, the two halves of the duplex are assumed to run different database systems with disjoint sets of databases. For an equitable comparison with the partitioned mode, the same types of transactions and databases are assumed in the single image mode. The performance level derived for the single image mode of operation for each state is summarized in columns 6 and 7 of Table 2.1.

There are two coverage probabilities involved in this study. One is the probability r that a memory can be reconfigured after either memory or memory controller failure to continue system operation. This coverage probability is usually a characteristic of the memory hardware design. We set $r = 0.75$. Then there is the coverage probability that the system does not suffer an outage on the failure of one component of a parallel pair in the coherence diagram. This coverage probability is determined by the functionality of the software to continue operation in the face of component failure. In an ideal system this probability should be unity. However in practical systems limitations of software (due to which every possible error state is not covered) cause the coverage probability for each type of failure to be smaller than unity. In this study we set coverage probability for all such components to the same value c and vary c in our studies.

Figure 3.2.1 shows the transient average normalized capacity of the single image and partitioned modes of operation as a function of the coverage probability. Software failure rate is assumed to be 0.5 per week. For these parameter values, partitioned mode operation exhibits smaller performability as compared to the single image mode operation. The comparison is sensitive to the software failure rate, as indicated in Figure 3.2.2(a). For higher software failure rates partitioned mode operation exhibits greater performability. This is because while partitioned mode loses one half of its processing capability on software failure, the single image mode loses its entire processing capability. In contrast single image mode has greater performability than partitioned mode for large hardware failure rates, because of the ability of the single image mode to use redundant components to mitigate the degradation in performance during hardware component failure. This is illustrated in Figure 3.2.2(b). In Figure 3.2.3(a) steady state and transient average normalized capacity for the single image and the partitioned mode operation are plotted, assuming that software failure rate is 0. Transient average (curves with labels subscripted with a T) is always greater than the steady state (curves with labels subscripted with an SS) because for smaller mission time the probability of a failure within the mission is smaller and because we always start in the normal system state for transient performability. The single image normalized capacity (curves labelled SI) is better than partitioned (curves labelled P). When the software failure rate is set to 1 per week (Figure 3.2.3(b)) the partitioned system has greater normalized capacity - indicating the sensitivity of the comparison to software failure rate.

As pointed out earlier the exact relationship between the software failure rate in single image mode and in partitioned mode is unknown. An expansion factor of 2 for the software failure rate in single image mode over the simplex software failure rate has been so far assumed. We explore the sensitivity of the comparison between the partitioned and single image modes to the expansion factor in Figure 3.2.4. Each half of the duplex in partitioned mode is assumed to suffer a software failure rate of 1 per week - that for the simplex. We vary the software failure rate expansion factor of the single image mode from 1 to 2. For low software failure rate expansion factors the single image mode has greater performability than partitioned mode because of the ability of the single image mode to partially cover hardware component failure. On the other hand partitioned mode always loses half the system. As the single image mode software failure rate expansion factor increases the relationship between the performabilities in the two modes is reversed exposing the greater impact of a software failure on single image mode of operation.

Since the difference between steady state and transient normalized capacity is of the order of less that 1%, one may be tempted to use steady state normalized capacity itself to approximate the transient normalized capacity as the comparison metric. Unfortunately, such an approximation leads to anomalies shown in Figure 3.2.5, over a wide parameter range. In Figure 3.2.5 we vary both the software failure rate and the coverage probabilities and find that in region A, single image mode operation is superior to the partitioned mode operation when compared on the basis of transient normalized capacity. The relation is reversed in regions B and C. Comparing on the basis of steady state normalized capacity, single image is better in regions A and B and partitioned mode better in region C. In region B the single image is better for the steady state but worse for the transient. The reason is that steady state favors single image because even though uncovered failures cause system outage for the single image, a re-IPL following the outage puts the single image into a state with reward rate greater than 0.5 in some cases and at worst 0.5 in the remaining, as in Figure 2.2.2(c). For short mission times the single image does not derive the full benefit of being in the intermediate stage and the penalty of the 10 minute system outage is amplified. Hence, even though the using steady state instead of transient performability constitutes less than 1% error, conclusions drawn by the use of steady state performability may not be the same as using the results of transient analysis. Therefore, in section 4 we derive a quick way to approximate the transient normalized capacity from the steady state normalized capacity.

In Figures 3.2.6 and 3.2.7 the 99'th percentile of normalized capacity has been plotted for the single image and partitioned modes for software failure rates of 0 and 1 per week, respectively. Both exponential and fixed repair time distributions are considered. The jagged nature of the curves are due to inaccuracies inherent to simulation. We observed from simulations that the fixed repair time distribution has a smaller transient average normalized capacity than the exponential. This is because in the exponential case - long repair times tend to get cut off by the end of the mission; hence the expected outage time in the exponential case is smaller than that in the fixed repair time distribution case. However, there are some missions with large outage times, for the exponentially distributed repair times (as expected), in comparison to the fixed repair time cases. Thus, the tail of the transient normalized capacity is longer for the exponentially distributed repair times cases than for the fixed repair time cases. This phenomenon can affect the higher percentiles. In Figures 3.2.6 and 3.2.7 the effect is seen for the partitioned 99'th percentile capacity, where the exponential case is worse than the fixed. From Figures 3.2.6 and 3.2.7 we find that the single image mode of operation exhibits better 99'th percentile transient normalized capacity for both software failure rates of 0 and 1 per week. In the previous figures we had plotted average normalized capacity and found that the comparison between the two modes was affected by the two failure rates. (This implies that we have to be careful about the choice of metric for comparison of the modes of operation of the two systems - because they give different results.) The choice of average or percentile of the performability distribution is of course determined by the goals of the system installation.

4. Approximations for Transient Performability

The previous section presented results of the transient analysis and simulation of normalized capacity for the two operation modes, and compared these with the steady state results. We note that while the steady state analysis is straightforward, transient analysis is complex. The complexity of transient analysis arises because it requires the solution of an eigenvalue problem as illustrated in Theorem 3.1, and involves evaluating time constants and coefficients of the transient terms. The solution procedure can lead to numerical problems. Although the difference between the transient and steady state average normalized capacity can be small the use of steady state analysis may give rise to anomalies, as discussed in Section 3. Thus, the question arises as to when transient analysis is required. In this section we develop simple upper and lower bounds and an approximation for the difference between the transient and steady state average normalized capacity. These bounds can be used both to determine the deviation of steady state normalized capacity from transient average normalized capacity and as an approximation to the transient solution. We illustrate the use of these bounds by applying it to the operation modes of Section 3, and observe that the bounds are tight. The approximation lies between the bounds, and appears to be a good one.

For ease of presentation, we first consider a simple two state case, with states denoted by 1 and 0, and with rewards 1 and 0 in these states respectively. The high reward state typically represents the "up" state, and the low reward state the "down" state. Thus, the system starts in the 1 state at the start of the mission. Furthermore, failures are rare. That is, on the average, the system spends most of the mission time (t) in the 1 state.

For computing the steady state average normalized capacity, the time axis can be divided into intervals, each of length equal to the mission time t, and averaging over this ensemble. For a Markov failure and recovery process (i.e. an exponentially distributed time in each state), the steady state average normalized capacity $\overline{m}$ can be computed as,

$$\overline{m} = \pi_1 m_{1,1}(t) + \pi_0 m_{0,1}(t) ,$$

where, $m_{x,1}(t)$ is the transient normalized capacity when starting in state x at the beginning of the interval, and π_x is the probability of being in state x at the start of the interval. For the transient normalized capacity, we are typically interested in the performability given that we start in the "up" (1) state. The difference between the steady state normalized capacity and the transient average normalized capacity when starting in the "up" state is

$$m_{1,1}(t) - \overline{m} = (1 - \pi_1) m_{1,1}(t) - \pi_0 m_{0,1}(t) .$$

Since $\pi_0 + \pi_1 = 1$,

$$\begin{aligned} m_{1,1}(t) - \overline{m} &= \pi_0 (m_{1,1}(t) - m_{0,1}(t)) \\ &\leq \frac{\pi_0 t_0}{t} , \end{aligned} \tag{4.1}$$

where t_0 is the mean "down" time. This last inequality is derived as follows. Figure 4.1(a) shows that the average normalized capacity, $m_{1,1}(t)$, starting in the "up" state is monotonically decreasing. Figure 4.1(b) shows the case of starting in the "down" state, and staying in the down state for time $\hat{t}$. Following this down time, the average normalized capacity for the rest of the mission time is $m_{1,1}(t - \hat{t})$. The average reward when starting in the "up" state, is the area marked as A and bounded by **adef** in Figure 4.1(b), while the corresponding average reward starting in the "down" state is marked as **B** and bounded by **bcjf**. The difference $m_{1,1}(t) - m_{0,1}(t)$ is (Area A - Area B)/t. An upper bound on this difference is merely (Area C)/t, where area C is bounded by **ghej**, or $\hat{t}/t$. Now, the mean value of $\hat{t}$ is,

$$\int_0^t \frac{x}{t_0} e^{-x/t_0} dx + \int_t^\infty e^{-x/t_0} dx = t_0 - t e^{-t/t_0} < t_0 \tag{4.2}$$

Thus, $(m_{1,1}(t) - m_{0,1}(t)) < t_0/t$, giving inequality (4.1). The bound can be improved by using equation (4.2).

This bound is easily generalized to a system with more than two states. For a system with (N+1) states denoted by 0,...,N , and state N as the "up" state,

$$\begin{aligned} m_{N,1}(t) - \overline{m} &= m_{N,1}(t) - \pi_N m_{N,1}(t) - \sum_{i=0}^{N-1} \pi_i m_{i,1}(t) \\ &= \sum_{i=0}^{N-1} \pi_i (m_{N,1}(t) - m_{i,1}(t)) \\ &\leq \sum_{i=0}^{N-1} \pi_i \frac{(f_N - f_i) T_i}{t} \end{aligned} \tag{4.3}$$

where, π_i is the probability of being in state i at the start of the interval, f_i is the reward in state i, T_i is the mean time to get from state i to state N , and t is the mission time. The final inequality in equation (4.3) follows by an extension of the argument used in deriving equation (4.1), and assumes that the reward in intermediate states in returning from a down state i to state N is a non-decreasing reward function.

The bound in equation (4.3) can be tightened by considering the rewards and times spent in intermediate states between states i and N. Suppose that the system (given that a transition occurs) goes from a failure state i to state N through intermediate states, with rewards as indicated in Figure 4.2. The concomitant loss in performability is indicated by the area A in this figure, and will be denoted by Δ_i for the loss in state i. Suppose that the system goes from state i to state j, $0 \leq j < N$, with probability p_{ij} when a transition occurs. Then, for exponential failure and repair times,

$$\Delta_i = \sum_{j=0}^{N-1} \Delta_j p_{ij} + (f_N - f_i) t_i , \tag{4.4}$$

where t_i is the mean time spent in state i and $f_N - f_i$ is the loss in reward in state i compared with the "up" state N. Thus,

$$\begin{bmatrix} 1 & . & . & . & . \\ . & 1 & . & -p_{ij} & . \\ . & . & . & . & . \\ . & . & -p_{ij} & 1 & . \\ . & . & . & . & 1 \end{bmatrix} \begin{bmatrix} \Delta_0 \\ \Delta_1 \\ . \\ . \\ \Delta_{N-1} \end{bmatrix} = \begin{bmatrix} (f_N - f_0) t_0 \\ . \\ . \\ . \\ (f_N - f_{N-1}) t_{N-1} \end{bmatrix}$$

Writing this equation as $\mathbf{P} \underline{\Delta} = \mathbf{R}$,

$$\underline{\Delta} = \mathbf{P}^{-1} \mathbf{R}. \tag{4.5}$$

Now, we may rewrite equation (4.3) as,

$$\begin{aligned} m_{N,1}(t) - \overline{m} &= \sum_{i=0}^{N-1} \pi_i (m_{N,1}(t) - m_{i,1}(t)) \\ &\leq \sum_{i=0}^{N-1} \pi_i \frac{\Delta_i}{t} = \frac{\underline{\pi}^T \mathbf{P}^{-1} \mathbf{R}}{t} \end{aligned} \tag{4.6}$$

While equation (4.6) expresses a solution for the general case, for most systems, such as those examined in Section 3, $\underline{\Delta}$ reduces to a simple summation and can be written directly by inspection, as discussed below; this applies to transition diagrams in which all cycles pass through the "up" state, as in the transition diagrams of Figures 2.2.1(c) and 2.2.2(c).

Figure 4.2 illustrates an alternate method of deriving the upper bound of equation (4.6), and an approximation and lower bound. This figure shows the reward as a function of time, starting in a down state i, with reward rate f_i. At time $\hat{t}$ the system is in the "up" state N for the first time, with reward rate f_N. Thus, from time $\hat{t}$ to the end of the mission time t, the system behaves as though it had started in the "up" state N, and had a mission time of $(t - \hat{t})$. That is, the average reward from time $\hat{t}$ to t when starting in the "down" state i, is the same as the average reward from time 0 to $(t - \hat{t})$ when starting in the "up" state N. Notice from equation (4.3), we are interested in the difference $m_{N,1}(t) - m_{i,1}(t)$ representing the difference in rewards when starting in state N, and when starting in state i. From the above observation, it follows that this is the same as the difference in the reward from time t to $(t + \hat{t})$ when starting in the "up" state at time $\hat{t}$ (denoted as Reward1) and the reward from time 0 to $\hat{t}$ when starting in the down state at time 0 (denoted as Reward2). This difference in rewards is shown as area B in Figure 4.2 (where the reward from time 0 to $\hat{t}$ when starting in state i is replicated between times t and $(t + \hat{t})$, so that the difference in rewards can be indicated). The upper bound in equation (4.6) takes Reward1 as $(f_n \times \hat{t})$, with area B upper bounded by area A in Figure 4.2. A closer approximation can be obtained by taking Reward1 as $(\overline{m} \times \hat{t})$, where $\overline{m}$ is the steady state normalized capacity. With this approximation, equation (4.4) is changed to,

$$\Delta'_i = \sum_{j=0}^{N-1} \Delta'_j p_{ij} + (\overline{m} - f_i)\, t_i \ , \qquad (4.4')$$

with Δ'_i as an approximation to the mean lost reward (i.e., mean area B) when starting in state i. The rest of the development is similar to deriving equations (4.5) and (4.6). Again, Δ'_i can be written by inspection for many interesting cases, such as those in Section 3.

A lower bound for the difference between transient and steady state can be derived for cases in which all cycles pass through the "up" state. The error in the upper bound of equation (4.6) is due to two approximations. First, the up-state reward f_N is used for the mean reward during times t and $(\hat{t} + t)$, when starting in the "up" state at time $\hat{t}$. This is an upper bound, while the steady state performability is a lower bound on this mean reward, because the average normalized capacity, when starting in the up-state is a monotonically decreasing function of mission time. Second, the mean time in a down state is overestimated in the upper bound, as expressed in equation (4.2). This is because, the effect of the mission time boundary is not accounted for in the upper bound. As for the two state case, for which the mean down time is derived in equation (4.2), the exact mean down time can be derived in closed form, as discussed in Donatiello and Iyer (1984), and used in place of t_i in equation (4.4′) Combining this with using the steady state performability instead of f_N, as in the above approximation, leads to a lower bound.

First consider the partitioned system structure of Figure 2.2.1(a). The transition diagram of Figure 2.2.1(c) shows the "up" state and two failure states, each with reward 0.5, and mean down times of $1/M_R$ and $1/M_I$ respectively. Denoting these failure states as 0 and 1, and since there are no intermediate states between these failure states and the "up" state, equations (4.3) reduces to,

$$m_{N,1}(t) - \overline{m} \ le \ \frac{0.5\pi_0}{M_R\, t} + \frac{0.5\pi_1}{M_I\, t} \ .$$

The resulting bound on the transient normalized capacity is shown in Figure 4.3 for a software failure rate of 1 per week. The value computed by the exact analysis of Section 3 is shown as a solid line that is barely distinguishable from the bound, shown as the dotted line. The error in the difference between the steady state and transient normalized capacity is about 0.6%. (The percentage error in estimating the transient normalized capacity, rather than the difference between the transient and steady state, is much smaller, or about 0.0002%.)

We now consider the single-image mode. The transition diagram of Figure 2.2.2(c) shows some failure states that return to the "up" state through some intermediate states. However, the solution of equation (4.5) for $\underline{\Delta}$ can be written directly by inspection. For instance, for the failure of the interface sensor sets (I), with rate L_I the system goes to a state with zero reward, say state i, for time $1/M_I$, and then goes to a state with reward 0.5 and stays there for time $(1/M_R - 1/M_I)$. Thus, Δ_i equals $1/M_I + 0.5\,(1/M_R - 1/M_I)$. The other components of the vector $\underline{\Delta}$ can be computed similarly. Figure 4.4 shows the resulting bound on the transient normalized capacity as a function of the coverage probability, for a software failure rate of 1 per week. Again, the bound (dashed line) is seen to be very close to the exact analysis (continuous line).

The bound can also be applied to some non-exponential repair time distributions. No exact solution to the general mean normalized capacity is known for non-exponentially distributed repair times. The only difference in the bound for non-exponential repair times is in estimating T_0 in equation (4.1), or Δ_i in equation (4.4). Considering the simple two state problem for which equation (4.1) was derived, the remaining time in the "down" state given that the (mission time) interval starts in the "down" state, now depends on the repair time distribution. For a fixed down time, the mean remaining "down" time is one half of the (fixed) down time. Thus, the bound predicts that the difference between the transient and steady state normalized capacity for a fixed down time distribution is about one half that for the exponential distribution with the same mean down time. A similar argument can be used to compute Δ_i in equation (4.4) for a general system with fixed repair times. Applying this bound to the partitioned system structure and assuming a fixed repair time distribution, provides a bound on normalized capacity of 0.997865 for a mission time of 21 hours and a zero software failure rate, versus a simulation estimate of 0.997818 and a steady state normalized capacity of 0.997705. Similarly, for a software failure rate of one per week, the bound on normalized capacity is 0.99689 versus a simulation estimate of 0.99693 and a steady state value of 0.99672. The difference of 0.00004 is within simulation accuracy.

The error in the above bound (i.e., the bound versus the actual difference between the steady state and transient normalized capacity) is sensitive to the error rates. For high error rates, the probability of being in a down state increases and so does the accumulated error in equation (4.3). Figure 4.5 shows the effect on the bound of varying the error rates from the nominal values in Section 3 up to ten times the nominal error rates, for the single image mode. The figure indicates that the upper bound is good even at ten times the nominal error rates. Figure 4.6 shows the upper and lower bounds, the approximation for, and the exact difference between steady state and transient normalized capacity with increasing failure rates. The upper bound is reasonably good even at high error rates, while the approximation and lower bound are practically indistinguishable from the exact value.

5. Conclusion

In this paper we used performability analysis to compare configurable duplex system structures. Since they differ in both frequency of outages and in the impact of outages on the system performance normalized capacity is used as a measure for comparison. We found the choice of optimal mode sensitive to the failure rate and to the performance metric - average or percentile. It is precisely for this reason that choice of mode of operation should be left to the installation, that can best determine which mode is suited for a particular environment. Use of steady state analysis instead of transient analysis gave rise to some anomalies. This motivated the upper bound, lower bound and approximation formulae for transient average normalized capacity derived in this paper. Both the bounds and the approximation were found to be very close to the values of transient average normalized capacity. The approximation and bounds are also applicable for computing the difference between steady state and transient average normalized capacity. In summary, we have shown that the combined goals of performance and reliability for reconfigurable duplex systems affect the choice of operation mode. While we have applied this methodology to configurable duplex systems, it can be used in any system with combined performance and reliability requirements.

Acknowledgements

We would like to thank Don Brand for discussing the single image and partitioned mode operations of the IBM 3084 with us, Joshua Kurtzberg and Al Pastro for developing the programs and plotting the curves used in this paper, Brian Bennett for useful comments on I/O modelling, Mickey Tsao for many clarifications and Yann-Hang Lee for reviewing a previous draft of this paper.

Appendix

In this appendix we use an example to illustrate the performance levels that can be obtained by the single image and partitioned modes of operations for different failure scenarios. The performance levels depend on the workload and the configuration. Here we will assume a typical transaction processing environment similar to that reported by Yu et. al. (1985).

Figures A.1 and A.2 show the assumed system configuration for the model. The figures show a typical I/O subsystem consisting of two sets of disks connected to the EXDC's through heads of string and control units, such that each disk has a dual path to the main memory. We will assume an equal load on each disk.

We use an approximate model to estimate the transaction response time versus throughput characteristic for normal operations and different failure scenarios. The average transaction path length, *INST* that includes the overheads for locking, I/O, application and commit processing, is assumed to be 1000 K instructions. Each CPU is modelled as an M/M/1 server. An M/M/1 model is natural for each processor in the partitioned mode. For the single image mode an additional assumption is required to justify the M/M/1 queueing model, i.e., the assumption that transactions are pre-assigned to execute on specific processors. This may be done to eliminate processor cache degradation. I/O's for a disk are queued in the operating system so that only one outstanding I/O can ever exist for a disk. The I/O time is derived by estimating the utilization of each channel path (the communication path between the EXDC and a control unit controlling accesses to disks, communications occur over these paths synchronously to match the rates at which data are accessed on disks during their rotation) and disk, computing the mean time taken by the disk for each I/O and then using an M/M/1 model for the total I/O time, including queueing for I/O requests. The CPU time is added to the total I/O time to give the total transaction response time. Finally, using a binary search procedure, we compute the throughput that can be supported for a given a response time constraint.

For more detail, the I/O time is estimated as follows. The disk service time consists of components, to send the I/O request to the disk (t_d); disk seek and latency ($t_s + t_l$); missed rotation time if no path is available from the disk to the EXDC when the required data on the disk arrives under the disk head (t_m); and the time to transfer the data (t_t). Let the probability that no channel path can be obtained at the time the channel program is started (I/O request is initiated) be β. This is also the probability that channel path is not available when the disk head is over the location of the data to be transmitted for the I/O. If the I/O request is blocked due to the channel busy condition it must wait for the current I/O transmission to end. Thus $t_d = \beta t_t/2$. Similarly the disk must rotate once more if the channel path is not available when the disk is ready for transmission, giving (using a geometric probability distribution for the wait time measured in number of disk rotation times) $t_m = t_r \beta/1 - \beta$, where t_r is the the time for the disk to complete one revolution. Typical values for the other delay components are $t_s = t_l = 8$ msec, $t_t = 1.5$ msec, and $t_r = 16$ msec.

The probability of obtaining a path depends on the number of channel paths to a disk and the probability that a channel path is busy when the request is made. The probability that a path is busy due to transfer of data to or from another disk is

$$\beta' = \left(\frac{n-1}{n}\right)\frac{\lambda I t_t}{P},$$

where n is the number of disks with which the path contention occurs, λ is the transaction rate, I is the number of I/O's per transaction, and P is the number of channel paths to each disk. Then $\beta = (\beta')^P$. For these configurations of Figures A1 and A2 $P = 2$.

The disk utilization is given by

$$\rho_{disk} = \frac{\lambda I}{n}(t_s + t_l + t_m + t_t).$$

Finally, the delay for an I/O seen by the operating system, because of queueing for the device, is estimated as

$$t_{IO} = \frac{t_d + t_s + t_l + t_n + t_t}{1 - \rho_{disk}}.$$

The number of I/O's per transaction depends on the number of memories available. Typical values are used as follows: 16 I/O's per transaction for configurations with 2 (1) memories and 2 (1) processors, 15 I/O's per transaction for the configurations with 1 processor and 2 memories and 19 I/O's for the configurations with 2 processors but 1 memory. Note that these values are sensitive to the workload. If *MIPS* denotes MIPS per CPU then the CPU time is estimated as

$$t_{CPU} = \frac{INST}{MIPS(1 - \rho_{CPU})}.$$

where ρ_{CPU} is the CPU utilization estimated as

$$\rho_{CPU} = \frac{\lambda\, INST}{N_P\, MIPS},$$

where N_P is the number of available CPU's in the complex. For the fully operational system, $N_P = 2$. For either a processor, SC or I failure, $N_P = 1$.

Using the following values for the parameters: total number of disks $n = 32$, $CPUMIPS = 15$ MIPS per processor and a response time bound of 0.5 seconds, the transaction rates that can be supported are shown in Table 2.1 after being normalized with respect to the transaction rate for the fully operational case.

References

1. Boos, D. D. (1985), 'An Introduction to the IBM 3084 Processor Complex', IBM Technical Bulletin GG 22-9387-00, Washington Systems Center, Gaithesberg, Maryland.
2. Bossen, D. C., and Hsiao, M. Y. (1980) , 'A System Solution for the Memory Soft Error Problem', *IBM Journal of Research and Development*, Vol 24, 390-397.
3. Bossen, D. C., and Hsiao, M. Y. (1982) , 'Model for Transient and Permanent Error-Detection and Fault-Isolation Coverage', *IBM Journal of Research and Development*, Vol 26, No 1, 67-77.
4. Donatiello, L. and Iyer, B. R. (1984), 'Analysis of a Composite Performance Reliability Measure for Fault-Tolerant Systems', IBM Research Report RC 10325, Yorktown Heights, New York.
5. Donatiello, L. and Iyer, B. R. (1985), 'Closed-Form Solution for System Availability Distribution', IBM Research Report RC 11169, Yorktown Heights, New York.
6. Furchgott, D. G. and Meyer, J. F. (1984), 'A Performability Solution Method Degradable Non-repairable Systems', *IEEE Transactions on Computers*, No 6.
7. Goyal, A. and Tantawi, A. N. (1984), 'Evaluation of Performability in Acyclic Markov Chains', IBM Research Report RC 10529, Yorktown Heights, New York.
8. Gustafson, R. N. and Sparacio, F. J. (1982), 'IBM 3081 Processor Unit: Design Considerations and Design Process', *IBM Journal of Research and Development*, Vol 26, No 1, 12-21.
9. Iyer, B. R., Donatiello, L. and Heidelberger, P. (1984), 'Analysis of Performability for Stochastic Models of Fault-tolerant Systems', IBM Research Report RC 10719, Yorktown Heights, New York.
10. Iyer, B. R., Dias, D. M., Yu, P. S. (1985), 'Performability Comparison of Configurable Duplex structres', IBM Research Report RC 11316, Yorktown Heights, New York.

11. Kulkarni, V. G., Nicola, V. F. and Trivedi, K. S. (1984), 'On Modelling the Performance and Reliability of Multi-Mode Computer Systems', in M. Becker(ed), Proc. Int. Workshop on Modelling and Performance Evaluation of Parallel Systems, North Holland.

12. Kulkarni, V. G., Nicola, V. F., Trivedi, K. S. and Smith, R. M. (1985), 'A Unified Model for the Analysis of Job Completion Time and Performability Measures in Fault-Tolerant Systems', Computer Science Dept. Research Report CS-1985-13, Duke University, Durham, North Carolina.

13. Lancaster, P. (1969), *Theory of Matrices*, Academic Press, New York.

14. Meyer, J. F. (1980), 'On Evaluating the Performability of Degradable Computing Systems', *IEEE Transactions on Computers*, C-29, 720-731.

15. Meyer, J. F. (1982), 'Closed-form solutions of Performability', *IEEE Transactions on Computers*, C-31, 648-657.

16. Pittler, M. S., Powers, D. M. and Schnabel, D. L. (1982), 'System Development and Technology Aspects of the IBM 3081 Processor Complex', *IBM Journal of Research and Development*, Vol 26, No 1, 2-11.

17. Reilly, J., Sutton, A., Nasser, R. and Griscom, R. (1982), 'Processor Controller for the IBM 3081', *IBM Journal of Research and Development*, Vol 26, No 1, 22-29.

18. Tendolkar, N. N., and Swann, R. L. (1982), 'Automated Diagnostic Methodology for the IBM 3081 Processor Complex', *IBM Journal of Research and Development*, Vol 26, No 1, 78-88.

19. Yu, P. S., Dias, D. M., Robinson, J. T., Iyer, B. R. and Cornell, D. (1984), 'Multi-system Data Sharing Analysis', IBM Research Report RC 10979, Yorktown Heights, New York.

Fail. Type	P	S	M	X	Uncovered	Covered
Normal	2	2	2	2	-	-
Processor	1	2	2	2	<0,I>,<0.57,RI>	<0.57,R>
Sys.Cntlr	1	1	2	1	<0,I>,<0.53,RI>	<0.53,R>
Memory	2	2	1	2	<0,I>,<0.70,RI>	<0,I>
Ex.Dt.Cnt.	2	2	2	1	<0,I>,<0.75,RI>	<0.75,R>
Inf.Ckts.(I)	1	1	1	1	<0,I>,<0.49,RI>	-

Column Headings

P : Number of healthy Processors
S : Number of healthy System Controllers
M : Number of healthy Memory Systems (Memory + Memory Controller)
X : Number of healthy External Data Controllers

Column Entries

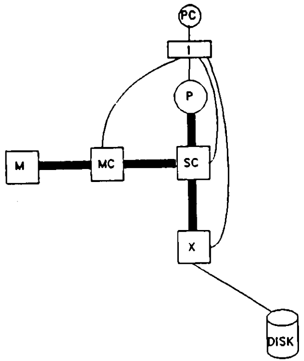
-: does not occur for this structure
(): continued operation without outage
<L1,a>,<L2,b> : sequence of events on failure - system degrades to state with reward rate L1 during the course of repair action a, at the end of action a the system moves to a state with performance level L2. The action b returns the system to its normal state of operation.

Repair Actions

I: IPL action
R: hardware repair action
RI: hardware repair action (started simultaneosly with previous IPL).

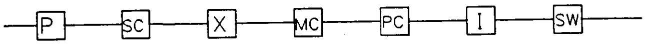
Failure Reaction Summary for the Single Image Operation Mode

Table 2.1



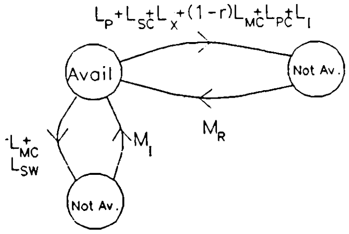
PC: Processor Controller, I: Interface Sensor, P: Processor, M: Memory
SC: System Controller, X: External Data Controller, MC: Memory Controller
Thick Lines: High Bandwidth Bus, Thin Lines: Low Bandwidth Bus

Basic Simplex System
Figure 2.1(a)

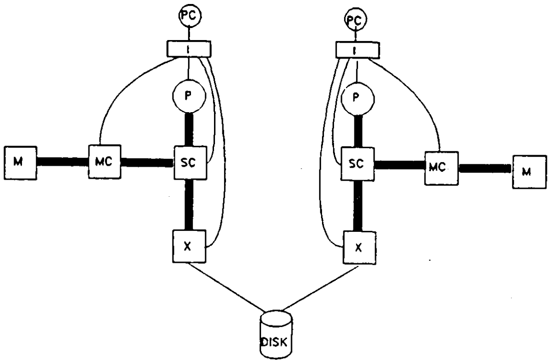


SW: Operating System Software

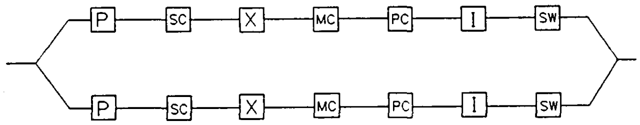
Coherence Diagram for Simplex System
Figure 2.1(b)



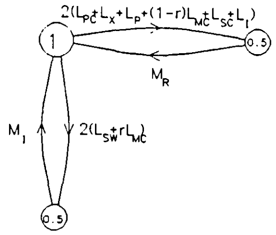
Transition Diagram for Simplex System Recovery
(Singl: Fault Assumption)
Figure 2.1(c)



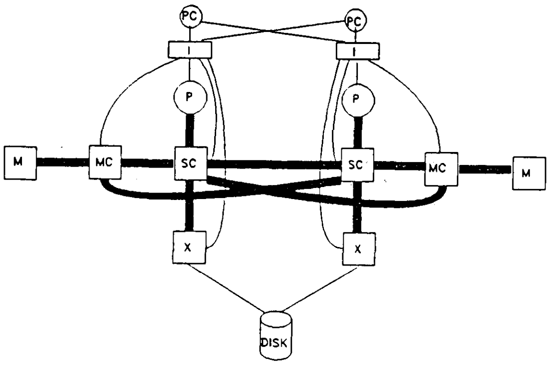
Duplex System Partitioned Mode
Figure 2.2.1(a)



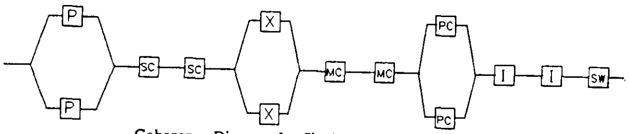
Coherence Diagram for Partitioned Mode Operation
Figure 2.2.1(b)



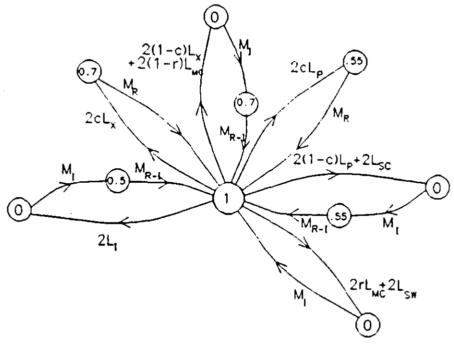
Transition Diagram for Partitioned Mode Recovery
Figure 2.2.1(c)



Duplex System Single Image Mode
Figure 2.2.2(a)



Coherence Diagram for Single Image Mode Operation
Figure 2.2.2(b)



Transition Diagram for Single Image Mode Recovery
Figure 2.2.2(c)

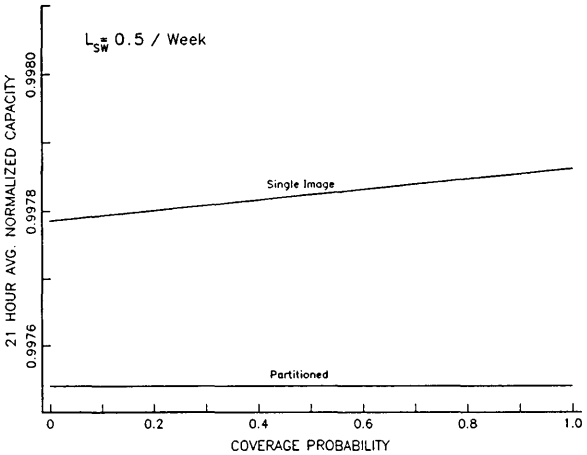


Figure 3.2.1

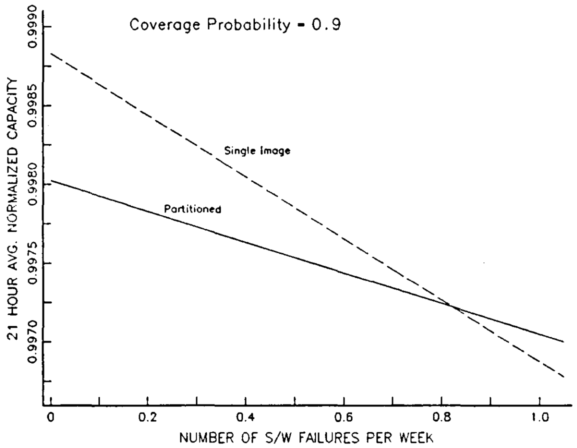


Figure 3.2.2(a)

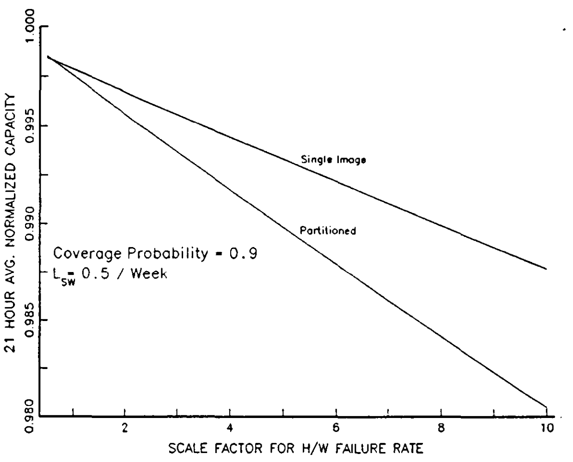


Figure 3.2.2(b)

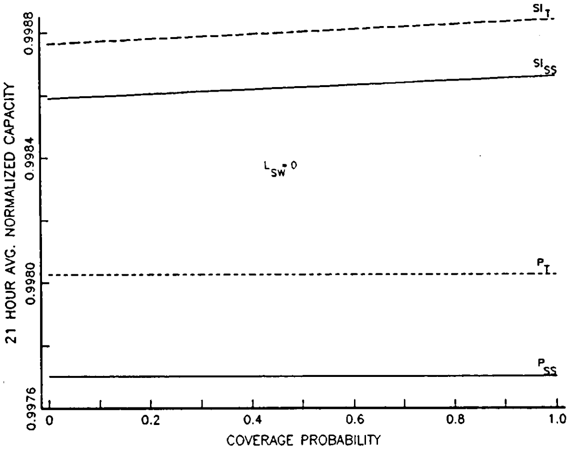


Figure 3.2.3(a)

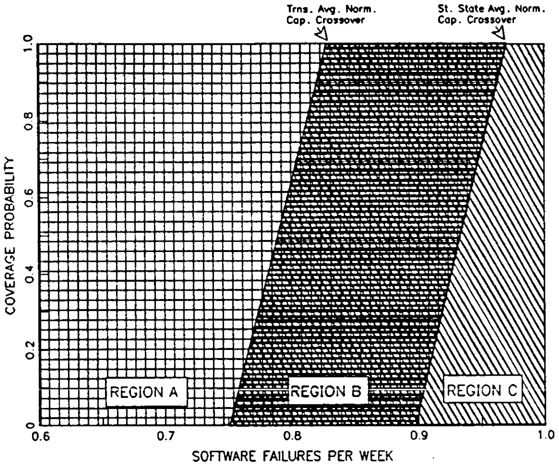


Figure 3.2.5

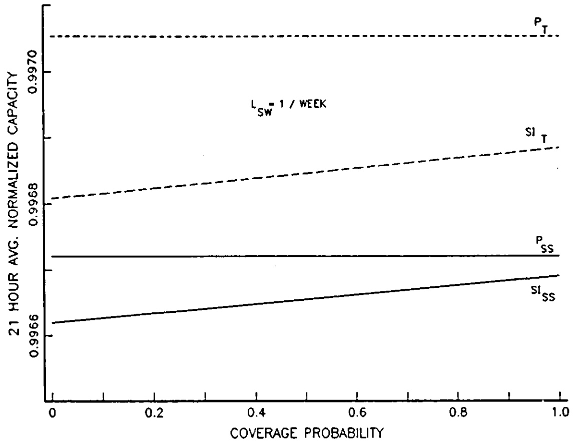


Figure 3.2.3(b)

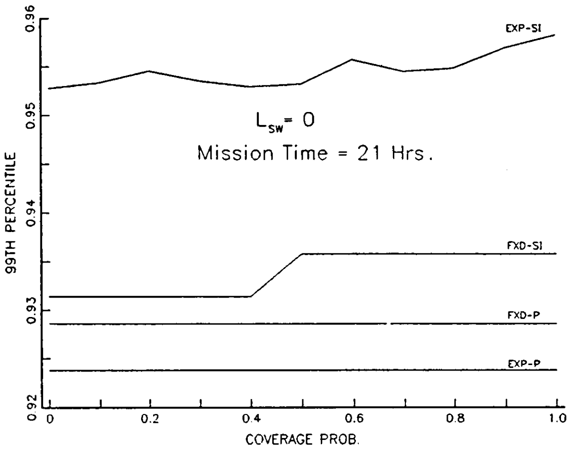


Figure 3.2.6

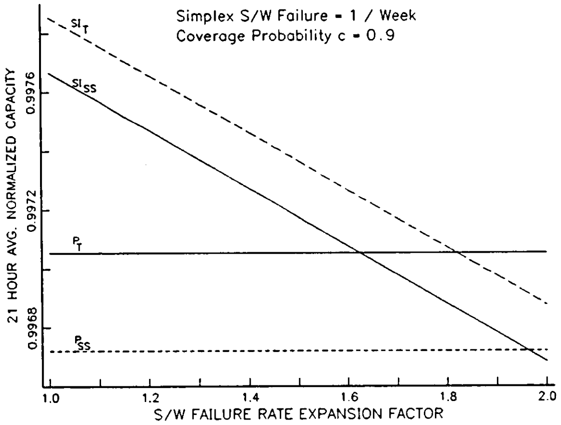


Figure 3.2.4

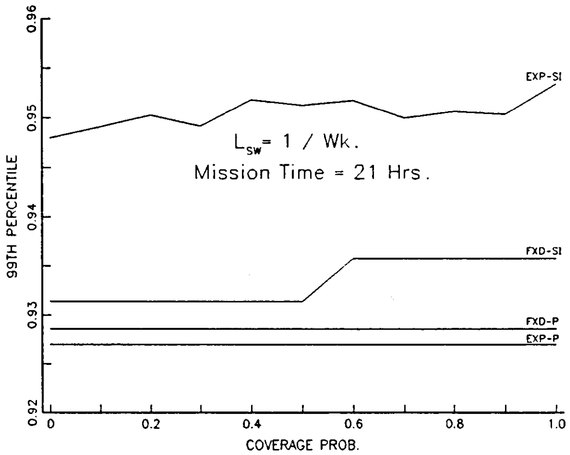


Figure 3.2.7

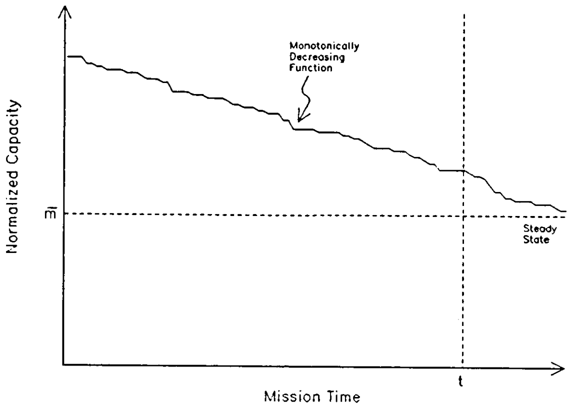


Figure 4.1(a)

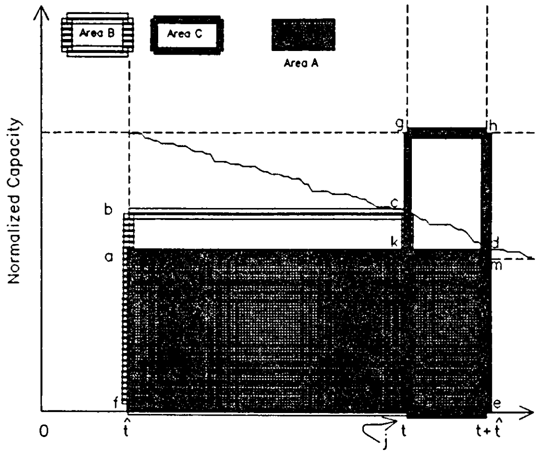


Figure 4.1(b)

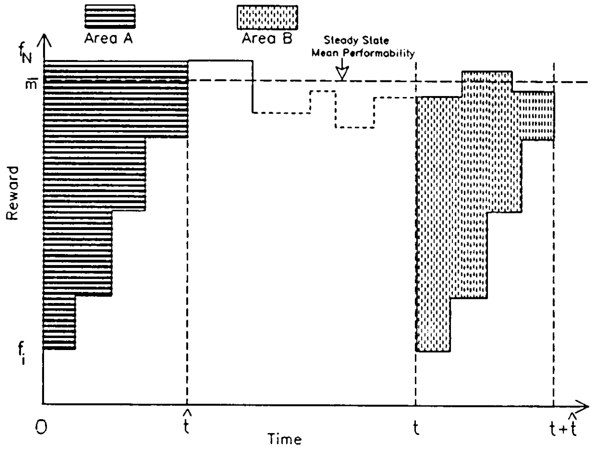


Figure 4.2

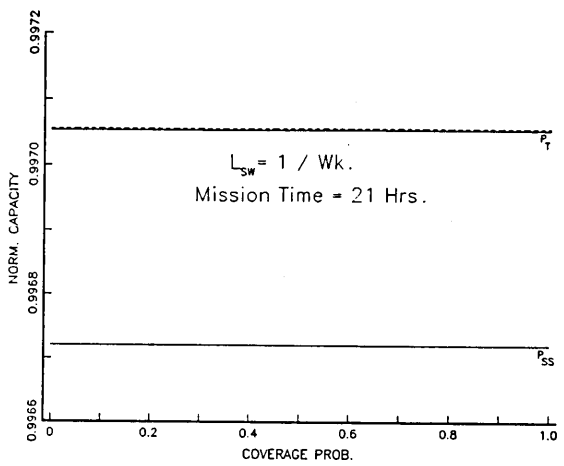


Figure 4.3

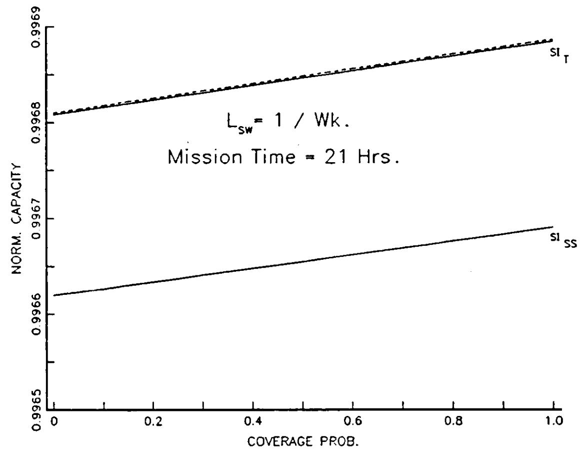


Figure 4.4

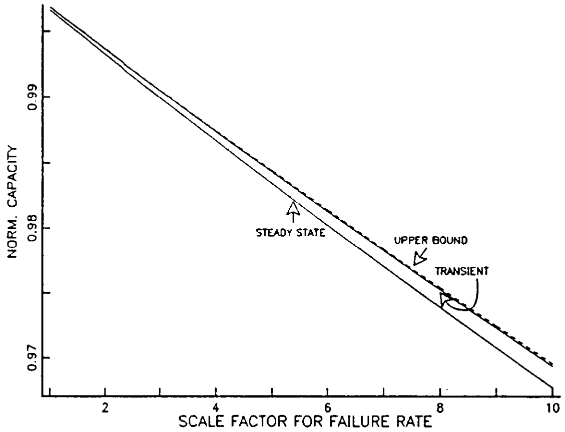


Figure 4.5

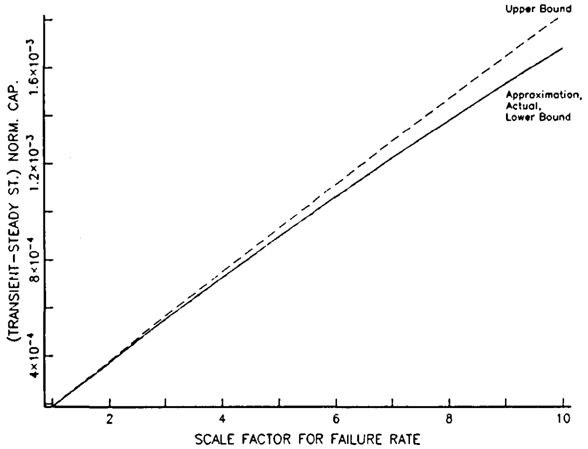


Figure 4.6

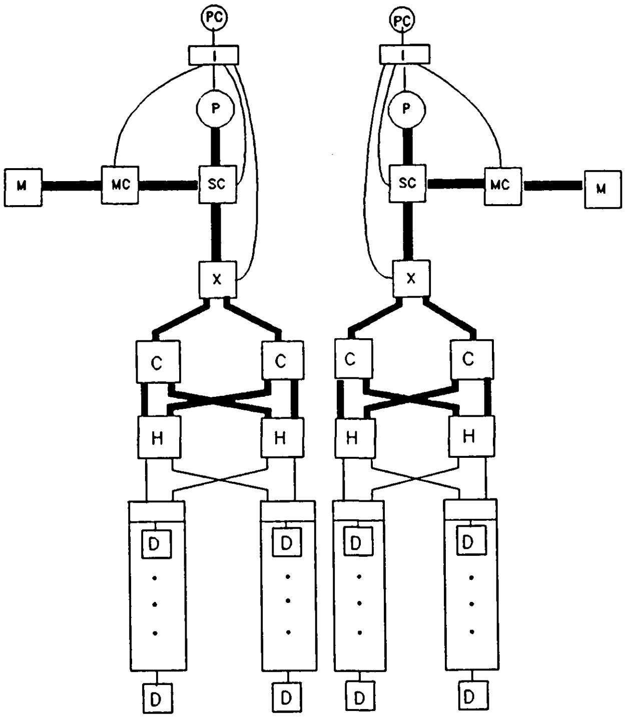


Figure A.1: Configuration for Partitioned Mode Operation

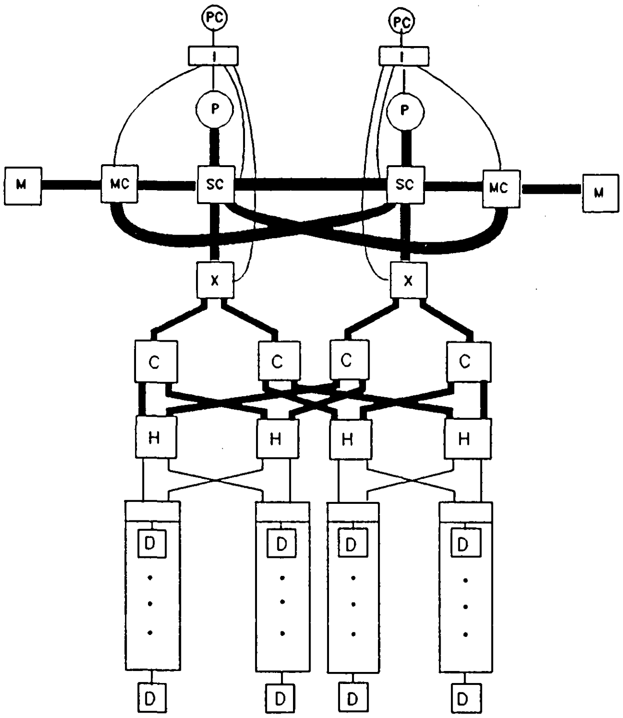


Figure A.2: Configuration for Single Image Mode Operation

Recognition of Error Symptoms in Large Systems

R.K. Iyer, L.T. Young and V. Sridhar

COMPUTER SYSTEMS GROUP
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL 61801

ABSTRACT

In this paper, a methodology for automatically detecting symptoms of frequently occurring errors in large computer systems is developed. The proposed symptom recognition methodology and its validation are based on probabilistic techniques. The technique is shown to work on real failure data from two CYBER systems at the University of Illinois. The methodology allows for the resolution between independent and dependent causes and, also quantifies a measure of the strength of relationship among the errors. Comparison made with failure/repair information obtained from field maintenance engineers shows that in 85% of the cases, the error symptoms recognized by our approach correspond to real system problems. Further, the remaining 15% although not directly supported by field data, were confirmed as valid problems. Some of these were shown to be persistent problems which otherwise would have been considered as minor transients and hence ignored.

Keywords: Error records, persistent errors, failure symptoms, probabilistic techniques, automatic recognition.

1. INTRODUCTION

The diagnosis of the causes of persisting errors in computer systems is difficult because the underlying faults are complex and may affect different parts of the system. The system usually detects the effects of the faults as many isolated errors. An incorrect diagnosis leads to improper recovery management which ultimately affects the integrity of the system. For the diagnosis to be effective, it is imperative that the system be able to relate errors occurring in different parts or different times.

When a service engineer tries to rectify a fault, he or she studies the error log for the period during which the fault occurred. Though the system may record the effect of a fault as many isolated incidents, the service engineer recognizes the different error entries as symptoms of the same error. This recognition of related error records is based on the service engineer's observance of some similarity, e.g., a recurring pattern among the error records.

This paper proposes a methodology for recognizing symptoms of severe errors in large systems. The goal of the methodology is to automate and formalize a process to relate errors occurring in different parts of a system. The method was developed after several years of study of system error logs and through close consultations with maintenance and field engineers from several manufacturers. The approach uses the system error rate to identify error records among which relationships can exist. Probabilistic techniques are then used to validate and quantify the strength of relationships among error records. The approach takes as input the raw error logs containing an entry for each error detected as an isolated event by a computer system, and produces as output symptoms which characterize persistent errors.

The methodology for automatically recognizing the error symptoms involves three steps:

(1) The recognition of error records among which relationships can exist,

(2) The determination of the existence and validity of relationships among these records and

(3) The quantification of the strength of a relationship if one is found.

The method is illustrated on error data collected from two large Cyber systems used on campus at the University of Illinois at Urbana-Champaign. The approach is simple in concept and hence easy to apply and simple to use. Since it is not based on any specialized characteristics of the Cyber systems or the error logs, it is expected to be applicable to other systems as well. Analysis of error data from two IBM systems is now in progress. Comparison made with failure/repair information obtained from field maintenance engineers shows that in 85% of the cases, the error symptoms recognized by our approach correspond to real system problems. Further, the remaining 15% although not directly supported by field data, were confirmed as valid problems by the engineers. Indeed, some of these were shown to be persistent problems which otherwise would have been considered as minor transients and hence ignored.

1.1 Related Research

Two studies which offer insight into the characterization of errors by their associated symptoms are [Velardi 84] and [Iyer 86]. Software errors in a production environment are analyzed in [Velardi 84], and permanent CPU errors and their relationship to workload are discussed in [Iyer 86].

CH2345-7/86/0000/0797$01.00 © 1986 IEEE

Research closely related to that described in this paper (symptom-based recognition) is that of [Tsao 83a] and [Sanders 85]. Tsao's work is motivated by the fact that a module exhibits a period of (potentially increasing) unreliability before final failure. The approach attempts to analyze trends in errors by first grouping errors occurring within a short period into Tuples. Heuristics, specific to DEC system error logs, are then used to determine if one tuple is similar to another [Tsao 83b]. As the author points out, two Tuples may be found to be identical by one algorithm, not identical by another. The result of the analysis is a small set of tuple types, each of which are expected to contain error records pointing to the same cause. The approach assumes that all error records in a Tuple are related.

In [Sanders 85], a technique which uses both complete and partial recurrences is proposed. The technique is similar in concept to that proposed by Tsao. Instead of using matching algorithms, a software table is used to quantify system organization. The software table is also used to capture both partial and complete recurrences as well as similarities among error groups. It was found that nearly 15% of the groups (similar to Tuples) contained a collection of unrelated error records, i.e., random groups.

An alternative approach to the above is called "structure-based" diagnosis. The development of a scheme based on the description of the structure and function of a system is described in [Davis 82]. Similar diagnostic schemes have also been proposed in [Shubin 82] and [Genesereth 82]. Another approach called "violated expectation" ([deKleer 76] and [Brown 81]) has also been proposed for troubleshooting. This approach looks for mismatches between the values expected from correct operation and those actually obtained. Whenever possible, the approach identifies a specific component that may be faulty. All these diagnostic systems have been shown to work on modules within a computer system. It is not clear how they will work when a whole computer system is taken into account.

In [Iyer 85], the analysis shows that the system is most vulnerable to errors in the hardware/software interface. It is not clear that a structure-based diagnosis can be fully successful under such circumstances. A structure-based diagnostic system may diagnose the hardware fault in a limited domain, but the chances are that complex faults (e.g. hardware-related-software errors) may be detected as more than one error. In this context, it becomes important to study the relationship among errors that occur within a short time.

In our opinion, both the symptom-based and structure-based approaches are still in their infancy. Hence, further research in both areas, particularly practical studies, needs to be done before either of them can generally be implemented.

The approach proposed in this paper is geared toward determining the symptoms of persistent errors in large systems by probabilistically relating the different manifestations of the same problem. The approach is naturally geared toward differentiating between transient and intermittent errors. To the authors' knowledge, there is no methodology that is able to do this at present. Further, the approach can also evaluate a diagnosis by analyzing the diagnoses made in the past, i.e., the goodness of system diagnostics. A methodology that quantifies the symptoms associated with related errors by determining the relationship among detected errors is essential for this purpose. The development of such an approach is discussed in the following sections.

2. SYSTEM CONFIGURATION AND ERROR MEASUREMENT

2.1 System Configuration

The system studied consisted of two Control Data CYBER 170 systems maintained by the Computer Services Office at the University of Illinois at Urbana-Champaign. The two machines, a model 174 and a model 175, are coupled by their disk system. The two machines run independently but share resources such as tape drives, etc. An interlock table is maintained for use by both machines to prevent deadlock arising from conflicts over shared resources. Details of the system configuration and the error detection are given in the Appendix.

2.2 Recognition of Error Groups

A cursory analysis of the data showed duplicate entries within a short time period describing the same error condition. It was decided to coalesce error records that report the same conditions (i.e., same type of error and same machine state) to avoid multiple records referring to the same error. This process is referred to as *error clustering*. Details of the clustering on the Cyber systems are given in [Sridhar 85]. A visual examination of the clustered data showed the existence of sets of clusters occurring within a short time interval. The close time proximity among some clusters means a substantial increase in the system error rate during that period. The high error rate introduces a suspicion that the errors occurring during the high error rate period may be related, i.e., different errors may be due to a single cause, to multiple but related causes, or to multiple and independent causes.

The high error rate periods referred to above are called error groups, and are formed by grouping all error clusters occurring within a small time interval of each other. (This interval was chosen to be fifteen minutes.) Analysis showed that the results were largely insensitive to variation in this time interval between 5 minutes and 1 hour. It will be seen in section 4 that relationships among error records not captured by this choice of time interval will be captured during subsequent analysis.[1] The primary difference between a cluster and a group is that clusters contain only occurrences of the same error (same error type and machine state), whereas groups contain occurrences of different errors (different error type or machine state).

[1] The choice of the interval is analogous to choosing a starting point in numerical optimization. A poor starting point will mean that the analysis will take a longer time.

As an illustration of the value of grouping, consider the series of observations in Figure 1. Looked upon singly, these would appear to be separate occurrences of different single bit memory errors. Yet, when the errors are combined into an error group, apparently all of them occur within bank 4. Also, the syndrome, or the problem bit, is the same for all the observations. This points to a problem within bank 4, and not to three isolated problems occurring within three different memory locations. Specifically, it refers to a problem that affects one particular bit of the word -- which means the problem is even further isolated to the module. The probable causes now are confined to those that affect only one module -- such as a faulty driver or path. In general, of course, the relationships will not be so obvious, hence the need for a probabilistic methodology proposed in the next section.

In summary, we start with the error log file and at the end of preprocessing we have reduced isolated errors to sets of clearly demarcated, periods of high error rate. These sets of error records are called error groups. Error groups identify periods of high unreliability in the system. The frequency of error groups demonstrates the need to relate errors occurring in different parts or times, i.e., global diagnosis. The tbf of the system increases considerably when groups are treated as single events. Thus, clearly, if the system were able to relate the errors in these groups it would be much more reliable. The determination of the existence and strength of relationships among error records in a group is described in the next section.

3. RECOGNITION OF ERROR PATTERNS

Errors in computer systems occur at various times and in many different locations. Often, the most severe errors are hard to diagnose because of the varied and seemingly unrelated symptoms associated with them as they occur in different locations and at different times. A successful diagnostic technique should be able to relate these errors in much the same way as a service engineer, through past experience, identifies a set of error records as the symptoms associated with a particular problem. The goal of this research was to automate and formalize a process to relate errors occurring in different parts of a system.

This section describes the key aspects of such a methodology, and the succeeding section gives examples illustrating the use of this approach. The objectives of the symptom recognition strategy are to start with the error log file, consisting of entries for all the errors detected as *isolated* errors by the system, and to produce as output symptoms associated with related errors that are being diagnosed as isolated and unrelated errors.

The symptom recognition methodology involves three steps:

(1) recognizing candidates among which relationships can exist,

(2) checking to see if a relationship actually exists among these candidates, and

(3) then estimating a measure of strength of the evaluated relationship (if a relationship does exist).

The first step, i.e., recognition of potential candidates for further analysis, is the time-based grouping described in the section 2.2. The second and third steps which are now discussed are recursively used at three levels of validation. At the first level, the relationship among the error records within an error group is evaluated; i.e., we determine whether there is a valid probabilistic relationship among the records in a group or whether they are simply a collection of unrelated records. The second level looks for inter-group relationships because the same cause(s) can give rise to multiple error groups within a short period. The third level looks for overall relationships in the entire data. Each level of analysis increases the resolution of the data and adds more certainty to the results. At each level, a measure of strength of the evaluated relationship is quantified.

The next subsection discusses the probabilistic validation procedure; the subsequent sections discuss the use of this procedure at different levels of analysis.

3.1 Probabilistic Validation Criterion

In this subsection, the technique used recursively to validate and measure the strength of relationships among the error records at various levels is described.

This technique examines the different records between which a relationship may exist and determines whether a valid relationship is possible among them on a probabilistic basis, based on the past data. In formal terms consider the probability space of n error records $A_1, A_2, \cdots, A_n$. Let $P(A_1), P(A_2), \cdots, P(A_n)$ be their respective individual probabilities calculated from the data, i.e.,

GRPNO	TIME	NPTS	REC- TYPE	ERR	SYN	QUAD	CSU	BANK	CHIP	ERR- TYPE
129	23SEP:14:38:18	3	0003	PARITY	52	1	0	4	2	MEM
129	23SEP:14:38:18	1	0003	PARITY	52	1	0	4	3	MEM
129	23SEP:14:38:18	1	0003	PARITY	52	1	0	4	0	MEM
129	23SEP:14:38:18	2	0003	PARITY	52	1	0	4	3	MEM
129	23SEP:14:38:18	2	0003	PARITY	52	1	0	4	2	MEM
129	23SEP:14:38:18	1	0003	PARITY	52	1	0	4	0	MEM

Figure 1: Sample of Grouped Data

(template)	(time)	(error description)			(associated device)	(exact location)			
Record Type	Timestamp	Error Type	Error Code	Syndrome	channel	disk	chassis	quadrant	bank

Figure 2: Error Log Entry for Memory Error

$$P(A_k) = \frac{\#\ of\ occur.\ of\ A_k}{\sum_{i=1}^{i=n} \#\ of\ occur.\ of\ A_i}$$

Then, $P(A_1) * P(A_2) * \cdots * P(A_n)$ is the probability that $A_1, A_2, \cdots, A_n$ will occur together, assuming they are independent. Further, let $P(A_1, A_2, \cdots, A_n)$ be the joint probability of occurrence computed from all the joint occurrences of $A_1, A_2, \cdots, A_n$ in the error log, i.e.,

$$P(A_1, \cdots, A_n) = \frac{\#\ of\ joint\ occur.\ of\ (A_1, \cdots, A_n)}{\sum_{i=1}^{i=n} \#\ of\ occur.\ of\ A_i}$$

If

$$P(A_1) * \cdots * P(A_n) < P(A_1, \cdots, A_n) \qquad (3.1)$$

then it is reasonable to assume that the joint occurrence of $A_1, \cdots, A_n$ is not random. A measure of the strength (S) of the evaluated relationship is given by the ratio

$$S = \frac{P(A_1, \cdots, A_n)}{P(A_1) * \cdots * P(A_n)}$$

What $A_1, A_2, \cdots, A_n$ stand for depends on the level of analysis:

(1) When we are attempting to validate the relationship among the error records in an error group, $A_1, \cdots, A_n$ stand for the individual error records within the error group

(2) When we are studying the relationship among the error groups in an event, $A_1, \cdots, A_n$ stand for each individual error group. Since the existence and strength of relationships are evaluated probabilistically, the approach is not constrained to specific architectures or system configurations.

3.2 Validation at Group level : Inter-Record Relationships

Recall that error groups represent high error rate periods during the operation of the system. The high error rate introduces a suspicion that the error records within an error group may be related. It is, of course, possible that the rise in error rate is just a random incident. To distinguish between these two possibilities, error records in an error group are analyzed in three different ways to determine if the records in an error group have a valid relationship.

Complete Analysis

The first is referred to as a *complete analysis*. The probabilistic validation procedure discussed in the previous subsection is applied to all the records in an error group, i.e., the individual and the joint probabilities of all the error records in a group are computed from the data. If all the error records in an error group are found to be statistically related, no further analysis is done on that error group.

Subset-Based Analysis

If error records in a group are found not to be related through a complete analysis described above, then it may be that subsets of the original error group have valid relationships. Different combinations of error records in a group are then analyzed to determine the existence of valid statistical relationships among them. For example, if there are three records (A, B, C) in an error group, the subsets (A, B), (A, C) and (B, C) are analyzed and Equation (3.1) is used to determine the relationship among the error records in these subsets. This approach is referred to as a *subset-based* analysis.

Truncated Analysis

If no relationship is found with complete or subset-based analysis, a third analysis based on truncated records is carried out. In this step, a few fields of the error records are masked from the analysis. This is best illustrated using the format in Figure 2, which refers to the log of a memory error.

The fields chassis, quadrant, bank and chip describe the location of error. It may be that a statistically valid relationship exists at the bank level (say among a few banks) and that the chip identity of the error records may thwart the recognition of the valid bank level relationship.[2] Thus, when the chip identity is dropped, we are attempting to determine the existence of statistically valid relationships at the bank level. Note that when the analysis is carried out with truncated records, more records can enter the probability space. Thus, the final result can be substantially different from the original result.

[2] When the chip identity is also taken into account, we are essentially trying to determine if errors occurring in different chips within a short period are statistically related or not.

[3] The same masking analysis may be carried out by starting from the highest level of resolution, i.e., the last field of the record. The chip identity is masked and the analysis is carried out; then the bank identity is masked and so on until a relationship is found or until all the fields are masked. This algorithm will require more run time. The higher resolution offered by this analysis is achieved by subsequent analysis.

The fields to be masked are determined as follows: starting from the lowest level of resolution, i.e., the beginning of error records, the records in a group are scanned until the first field wherein they are dissimilar is found. All the fields beyond this field are masked from the analysis. For example, two records with the format shown in Figure 2 may have identical fields up to the chassis. The quadrant is the first field with non-identical values. Then the fields bank and chip are masked from the analysis.[3]

This three-tier analysis eliminates error groups consisting of records with no relationships, i.e., random errors, from further analysis. The importance of this analysis is underscored by the fact that in our analysis (shown in section 4) over 25% of the groups were rejected as random groups. The validation of subsets eliminates stray records present among related error records in an error group, and the validation of truncated records captures non-obvious relationships among error records. Thus, the original error groups consisting of records among which relationships *can* exist is refined to error groups consisting of records among which relationships *do* exist.

In some cases, the stray record does not invalidate a group. It merely reduces the strength of the evaluated relationship. Consider three records in a group, say A, B and C. A and B may be related and C may be the stray record. The strength of the relationship computed by analyzing A and B may be much higher than that obtained by analyzing A, B and C. It is possible to perform the three-tier analysis in all the cases and accept only the relationship with the highest strength. Such an analysis is, however, very computer-intensive and, therefore, was not performed here. Stray records are ultimately eliminated in subsequent analysis.

Relationships can exist across error groups; i.e., a single cause can give rise to a persistent error and thus foster multiple error groups within a short time. Therefore, it becomes necessary to examine the validated groups for inter-group relationships. The methodology to recognize inter-group relationships is described in the following subsections.

3.3 Validation at Event Level : Inter-Group Relationships

The persistence of a single cause(s) and the possibility of stray error records among related error records in an error group motivate the second level of analysis, i.e., recognition of related error groups and the elimination of stray error records from error groups.[4]

To analyze the relationship among error groups, it becomes necessary to introduce the concepts of events and symptoms, and symptom sets. An event is defined as the collection of error groups occurring within twenty-four hours and having at least two error records in common. A symptom is defined as a collection of statistically related, non-redundant error records that are common to most of the groups in an event. A symptom set associated with an event is defined as a collection of symptoms that best describe the event. Symptom sets are recognized by analyzing the error groups in an event.

When an event has just one error group, the statistically related error records within that error group form the symptom set associated with that event. When there is more than one error group in an event, the symptom set is formed by intersecting the error groups within the event. The error groups are taken N groups at a time, N-1 groups at a time, and so on down to (N/2), or to the next higher integer number of groups. The intersection of such different combinations of error groups yields different symptoms that are found in these groups. It also eliminates any stray records that occur in a few of the error groups in an event.

Once the symptom set associated with an event is derived, Equation (3.1) is used to check for the relationship among the different symptoms. If only one symptoms is extracted from an event, the event is probably due to a single cause. When more than one symptom is extracted from an event, the probabilistic validation procedure is used to determine if the different symptoms are caused by related or independent causes. For example, if two symptoms, say A and B, are extracted from an event, and if the symptoms satisfy Equation (3.1), i.e.,

$P(A) * P(B) < P(A,B)$

then the event is due to multiple but related causes, and the ratio

$P(A,B) / (P(A) * P(B))$

quantifies the strength of the relationship between A and B. If

$P(A) * P(B) > P(A,B)$,

the event is caused by multiple and independent causes.

If two independent causes exist and persist simultaneously,[5] the extracted symptoms will consist of confusing sets of error records. This is because all or most of the groups in the event contain records referring to both causes. When such groups are intersected, symptoms associated with both are extracted as though they were related causes. The records due to one cause are strays with respect to the other cause, and some technique is necessary to discriminate between the two. A higher level of analysis, necessary to deal with such occurrences, is described in the following section.

3.4 Validation at Super-Event Level : Inter-Event Relationships

The possibility of the presence of stray records in all of the error groups in an event, and the need to capture relationships missed by the choice of twenty-four hours as the time interval while forming events, motivate the next and the last level of analysis, i.e., determination of inter-event relationships. Further, when the events resulting from the same cause(s) across the entire data are analyzed, the stray records occurring in one particular event become visible. This allows for the resolution between independent and dependent (related) failures.

Three simple rules are used to recognize related events and to form *super events*. Two events are grouped into a super event if they satisfy any one of the following

[4] Noise is a representative term for stray error records.

[5] Such conditions occur rarely, but they do occur.

criteria:

(1) They have at least one symptom in common

(2) A symptom of one event is a sub or super set of at least one symptom of another event

(3) If they are single-group events, then they have at least two records in common.

Importantly, there is no time restriction when forming super events.

Next, a set of super symptoms, which are collections of symptoms found in common among two or more events in a super event, is generated. The procedure is exactly the same as for events except that instead of taking groups, we take events described by the symptom sets. The intersection operation among the events yields symptoms found in common among these events. The non-null result of each intersection operation forms a super symptom for the associated super event.

Some super events have just one super symptom associated with them, while others have more than one associated with them. To determine whether the different super symptoms associated with a super event are related, (i.e., whether the super event has a single cause) we use the probabilistic validation procedure described in Section 3.1. If the different super symptoms are not related, then the super event is due to multiple and independent causes.

When a super event is due to multiple and independent causes, the number of such distinct independent causes may be determined as follows: the super symptoms are considered in different combinations, taken N-1 super symptoms at a time, and so on down to 2 super symptoms. The probabilistic validation procedure is applied to each such combination. This analysis determines the relationship within a particular subset of super symptoms associated with a super event. If the subset is found related, it is counted as one cause. The analysis is continued until all independent causes associated with a super event are found.

Recall that the goal of this paper is to automate the process by which a service engineer relates errors occurring in different parts of a computer system. The super symptoms derived from the error log file not only consist of sets of related error records but also quantify the strength of their relationships. Whereas the service engineer usually scans the error log pertaining to the short period during which the error occurred, recognizes related error records and makes a local judgement, the methodology scans all the errors that occur in a system and is capable of making a global judgement. The application of this technique to real data is discussed in the next section.

3.5 Summary

The methodology described in this section establishes that it is possible to build a system that will relate errors occurring in different parts/times in a computer system or when two unrelated causes occur simultaneously in a system. It has been shown that the proposed methodology allows for resolution between them. Some causes persist for days or even months. While such causes persist, other independent errors can also occur in a system.

Three levels of validation are used in a recursive fashion. In the first, all groups (high error rate periods) are examined to determine if they contain related records. Next, the validated groups are examined for inter-group relations (events). Finally, the events across the entire date are examined for inter-event relationship.[6]

The symptom sets obtained using this methodology imply that if one error within a symptom set occurs, other related errors within the symptom set can be expected to occur. The strength of the relationship says how likely it is that related errors will follow.

The number of levels of analysis can be varied for higher resolution. That is, if a system is highly error-prone and a grouping of errors occurring within fifteen minutes leads to coalescing of unrelated errors, then this grouping time interval may be reduced to, say, five minutes or less. The next section illustrates various facets of this approach by applying it to determine the error symptoms in the CYBER system.

4. EXAMPLES OF ERROR SYMPTOMS

This section illustrates the various stages of the analysis involved in the automatic recognition of error symptoms from error log data. A summary of the analysis is given first. Then three examples are provided, illustrating the three levels of analysis, namely, the group, event and super event levels. Some features of the proposed methodology not illustrated by the first example are illustrated in the last two examples.

4.1 Summary of the Analysis

Table 1 sums up the number of groups, events and super events derived from the error logs of Cyber 174 and Cyber 175 machines. Note that 67.1% of the Cyber-174 groups and 27.5% of the Cyber-175 groups were rejected as random groups. Most of the super events in both the machines were due to single or related causes. In all, there are fifteen distinct causes that result in severe errors on the Cyber-174 and thirty-three causes on the Cyber-175.

TABLE 1:
FREQUENCY OF GROUPS, EVENTS AND SUPER EVENTS.

	Cyber 174	Cyber 175
# of groups	85	142
# of groups rejected	57	39
# of events		
due to single cause	19	53
due to multiple causes	0	2
# of super events		
due to single cause	15	33
due to multiple causes	0	2

[6] Note that each level of analysis adds to the information gathered at previous levels, i.e. a group validated at the group level cannot be rejected in subsequent analysis but can find other related groups.

There are 85 error groups in the Cyber-174 data and 142 groups in the Cyber-175 data. These numbers quantify the periods of highly unreliable operation during the operation of the computer system. During these high error rate periods, many of the system resources may have been utilized in recovery measures.

4.2 Cyber-174 Super Event with Multiple Super Symptoms

This section illustrates the various facets of the proposed methodology. This particular example is chosen because it illustrates all the important steps and the major problems involved in the automatic recognition of symptoms.

Figure 3 shows a super event consisting of four events. Here all events are single group events. As an example of analysis at the group level, we consider the group of error records in the second event, marked EVT 2. In this group, we find four types of records, i.e., those records with ERRTYPE/locations corresponding to E/..., S/0-0-7-1, S/1-0-1-2, and S/1-0-5-0. Using the method of Section 3.1, the strength of this group was 76.50. Since this strength is greater than 1.0, the records exhibit a valid relationship within this group, so subset-based and truncated record analyses are unnecessary. The strengths of the groups respective of the order that they appear in Figure 3 are 1.56, 76.50, 1.56, and 4.00. Analysis of symptoms at the event level is trivial in Figure 3, since each event has only one type of error symptom. An example which better illustrates event level analysis is discussed in the next section.

```
                                     Q
        R                  E  C   S  U C
        E                  R  H   Y  A H
        C                  R  A   N  D A
        T          T       T  N   R  R S B C
        Y          I       Y  N   O  A S A H
        P          M       P  E   M  N I N I
        E          E       E  L   E  T S K P

        3 84May17 08:05:01 S 14 D0 1 0 1 2
EVT 1   3 84May17 08:07:23 S 14 25 1 0 5 0
        3 84May17 08:15:47 S 14 23 0 0 7 1
       ------------------------------------
        3 84May20 11:24:59 E 14 23   .....
        3 84May20 11:33:21 S 14 23 0 0 7 1
EVT 2   3 84May20 11:40:48 E 14 23   .....
        3 84May20 11:58:14 S 14 D0 1 0 1 2
        3 84May20 11:58:14 S 14 25 1 0 5 0
       ------------------------------------
        3 84May28 12:30:45 S 14 23 0 0 7 1
EVT 3   3 84May28 12:44:22 S 14 25 1 0 5 0
        3 84May28 12:49:12 S 14 23 0 0 7 1
        3 84May28 12:53:47 S 14 D0 1 0 1 2
       ------------------------------------
        3 84Jun02 03:54:51 S 14 23 0 0 7 1
EVT 4   3 84Jun02 04:00:00 S 14 38 0 1 5 3
        3 84Jun02 04:15:54 E 14 23   .....
```

Figure 3:
Example of Multiple Symptom Super Event on CYBER-174

Analysis at the super event level is performed by applying the probabilistic validation procedure to the set of super symptoms extracted from a super event. In Figure 3, the union of event symptoms yields the following super symptom set:

$$(1)\ (\ E/\cdots\ .\ S/0\text{–}0\text{–}7\text{–}1\ .\ S/1\text{–}0\text{–}1\text{–}2\ .\ S/1\text{–}0\text{–}5\text{–}0\ )$$

$$(2)\ (\ E/\cdots\ .\ S/0\text{–}0\text{–}7\text{–}1\ .\ S/0\text{–}1\text{–}5\text{–}3\ ) \qquad (4.1)$$

By definition of a super event, these super symptoms can occur within only this super event. The strength of the entire super event by Equation (3.1) is 2.0, which indicates that the events are strongly related. Thus, this super event may be considered the result of a single cause. Had these super symptoms failed to satisfy Equation (3.1), we would have inferred that the super event was due to multiple causes.

It should be noted that the error records in this super event span a period of seventeen days. The cause appears to be an intermittent error which recurs, apparently affecting different components. That they are due to a single cause would not be immediately obvious from looking through the entire error log. This example clearly demonstrates the power of the proposed methodology to extract good quality symptoms associated with severe errors in computer systems.

4.3 Cyber-175 Super Event with Multiple Super Symptoms

Figure 4 shows part of a super event. This super event consists of ten events persisting over a period of the ten weeks from July 8 to September 18 of 1985. Three of these events are shown in Figure 4. The second event, marked EVT 2, consists of three groups, spanning a period of less than seven minutes each. In addition to occurring less than an hour apart and having at least two records in common with another group, these three groups had to be validated at the group level to be included in the same event. Using the measure of strength of relationship between records as described in Section 3.1, the value for the first group's locations (1-1-1-0 , 1-1-2-2 , 1-1-0-1 , 1-0-0-1) is 8.444. For the second group, the value for locations (1-1-1-0 , 1-1-2-2 , 0-1-7-2 , 1-1-3-3) is 12.901. And for the third group, the value for locations (1-1-1-0 , 1-0-0-1 , 0-1-7-2 , 1-1-3-3) is 10.411. These strengths clearly indicate the validity of the record relationships within the three groups.

Event Level Analysis

As an example of analysis at the event level, we consider the set of symptoms generated by the groups in EVT 2. Four symptoms are generated:

$$(1)\ (\ 1\text{–}1\text{–}1\text{–}0\ )$$

$$(2)\ (\ 1\text{–}1\text{–}1\text{–}0\ .\ 1\text{–}0\text{–}0\text{–}1\ )$$

$$(3)\ (\ 1\text{–}1\text{–}1\text{–}0\ .\ 1\text{–}1\text{–}2\text{–}2\ )$$

$$(4)\ (\ 1\text{–}1\text{–}1\text{–}0\ .\ 1\text{–}1\text{–}3\text{–}3\ .\ 0\text{–}1\text{–}7\text{–}2\ ) \qquad (4.2)$$

By Equation (3.1) the strength of event EVT 2 is 32.16. This indicates a valid relationship between the three error groups in event EVT 2, i.e., they were all caused by the same problem. But this inference is only a local view. To get a global view, we look for related events across the entire data.

Super Event Level Analysis

For the super-event level analysis, other events with at least one symptom set in common are extracted from the data. Here we illustrate the procedure for the events shown in Figure 4. Events EVT 1 and EVT 3 are single-group events. Their symptom sets are simply the groups themselves. The super symptoms obtained are:

$$(1)\ (\ 1\text{–}0\text{–}2\text{–}0\ ,\ 1\text{–}0\text{–}2\text{–}2\ ,\ 1\text{–}1\text{–}3\text{–}3\ )$$

$$(2)\ (\ 1\text{–}1\text{–}1\text{–}0\ ,\ 1\text{–}1\text{–}3\text{–}3\ ,\ 0\text{–}1\text{–}7\text{–}2\ )$$

$$(3)\ (\ 1\text{–}1\text{–}1\text{–}0\ ,\ 1\text{–}1\text{–}2\text{–}2\ ,\ 1\text{–}0\text{–}0\text{–}1\ ) \qquad (4.3)$$

Note that the first three symptoms of Equation (4.2) have been absorbed into the third super symptom of Equation (4.3). This provides global support of the inference made at a local level that errors occurring in locations 1-1-1-0, 1-1-2-2, and 1-0-0-1 are due to the same cause.

The physical interpretation of these three super symptoms is that the occurrence of one error implies that another error is likely to follow. The strength of the relationship among the errors in a symptom set tells us how likely it is that the related error will follow. The probabilistic validation procedure quantifies a measure of the strength of the relationship among error records based on their joint and individual occurrences.

For example, the values of the measure of relationship strength between error records pertaining to the three super symptoms of Equation (4.3), respectively, are 5.37, 5.26, and 2.17. Thus, when an error occurs in location 1-1-1-0, errors are likely to occur in locations 0-1-7-2, 1-0-0-1, 1-1-2-2 and 1-1-3-3. Given that an error has occurred in location 1-1-1-0, the chances are that an error is more likely to occur in location 1-1-3-3 or 0-1-7-2 than in location 1-1-2-2 or 1-0-0-1. Thus, the symptom sets tell us about the other parts or other errors that are likely to occur, and the measure of strength of the relationship among the records in a symptom set tells us which parts are more likely to be affected from among all the parts that can be affected.

A practical benefit of the proposed methodology is that it makes service/maintenance engineers aware of all the severe or persisting errors in a system. Although most causes of error took the form of single, transient groups of errors, there were quite a few that resulted in numerous recurring groups of errors, spanning months of time. For example, 26.7% of the CYBER-174's super events were composed of multiple groups of errors, as were 36.4% of the CYBER-175's super events. In this context, it is interesting to note that the maintenance engineers were unaware of the persisting errors discussed in this section until we brought it to their attention. This was not withstanding the fact that the same maintenance engineers went through the error log on alternate days of the week during the six-month period when these errors persisted. They, however, failed to notice them.

```
                                      Q
   R                  E C      S      U C
   E                  R H      Y      A H
   C                  R A      N  C P D A
   T            T     T N      R  H P R S B C
   Y            I     Y N  E E O  P P A S A H
   P            M     P E  S C M  A A N I N I
   E            E     E L  T S E  R R T S K P

E  3 84Sep05 04:01:33 S 14 0 5 58 0 0 1 0 2 2
V  3 84Sep05 04:01:47 S 14 0 5 89 0 0 1 0 2 0
T  3 84Sep05 04:02:15 S 14 0 5 3D 0 0 1 1 3 3
   3 84Sep05 04:02:28 S 14 0 5 58 0 0 1 0 2 2
1  3 84Sep05 04:03:32 S 14 0 5 3D 0 0 1 1 3 3
   -------------------------------------------
   3 84Sep13 06:00:09 S 14 0 5 54 0 0 1 1 2 2
   3 84Sep13 06:00:22 S 14 0 5 E9 0 0 1 1 0 1
   3 84Sep13 06:00:42 S 14 0 5 54 0 0 1 1 2 2
   3 84Sep13 06:01:31 S 14 0 5 23 0 0 1 1 1 0
   3 84Sep13 06:02:40 S 14 0 5 54 0 0 1 1 2 2
   3 84Sep13 06:03:05 S 14 0 5 23 0 0 1 1 1 0
   3 84Sep13 06:03:28 S 14 0 5 54 0 0 1 1 2 2
   3 84Sep13 06:05:11 S 14 0 5 8A 0 0 1 0 0 1
   3 84Sep13 06:05:40 S 14 0 5 54 0 0 1 1 2 2
   3 84Sep13 06:05:52 S 14 0 5 23 0 0 1 1 1 0
   3 84Sep13 06:05:58 S 14 0 5 8A 0 0 1 0 0 1
E
V  3 84Sep13 07:00:02 S 14 0 5 3D 0 0 1 1 3 3
T  3 84Sep13 07:01:40 S 14 0 5 54 0 0 1 1 2 2
   3 84Sep13 07:02:03 S 14 0 5 23 0 0 1 1 1 0
2  3 84Sep13 07:02:40 S 14 0 5 3D 0 0 1 1 3 3
   3 84Sep13 07:02:44 S 14 0 5 07 0 0 0 1 7 2
   3 84Sep13 07:02:47 S 14 0 5 3D 0 0 1 1 3 3

   3 84Sep13 08:00:33 S 14 0 5 23 0 0 1 1 1 0
   3 84Sep13 08:01:09 S 14 0 5 3D 0 0 1 1 3 3
   3 84Sep13 08:01:15 S 14 0 5 8A 0 0 1 0 0 1
   3 84Sep13 08:04:12 S 14 0 5 23 0 0 1 1 1 0
   3 84Sep13 08:05:32 S 14 0 5 07 0 0 0 1 7 2
   3 84Sep13 08:06:12 S 14 0 5 3D 0 0 1 1 3 3
   3 84Sep13 08:06:12 S 14 0 5 23 0 0 1 1 1 0
   3 84Sep13 08:07:24 S 14 0 5 3D 0 0 1 1 3 3
   -------------------------------------------
E  3 84Sep18 05:01:53 S 14 0 5 8A 0 0 1 0 0 1
V  3 84Sep18 05:03:26 S 14 0 5 23 0 0 1 1 1 0
T  3 84Sep18 05:03:30 S 14 0 5 8A 0 0 1 0 0 1
   3 84Sep18 05:03:59 S 14 0 5 54 0 0 1 1 2 2
3  3 84Sep18 05:05:06 S 14 0 5 23 0 0 1 1 1 0
   3 84Sep18 05:05:20 S 14 0 5 54 0 0 1 1 2 2
```

Figure 4:
Example of Multiple Symptom Super Event on CYBER-175

4.4 Super Event with Unrelated Super Symptoms

This section illustrates an example of a super event whose records appear to be strongly related at both the group and event levels but fail to show significant relationships at the super event level.

Figure 5 shows a super event constructed from CYBER-175 error records and consisting of two events. In the first event, marked EVT 1, there are two groups. Analysis of the records in each group yields strengths of

1.42 and 19.27 for the smaller and larger, respectively. In the second event, marked EVT 2, there are three groups. Their strengths, in chronological order, are 19.27, 5.23, and 1.42. Clearly, the records in all five groups are strongly related.

This strong relationship is preserved at the event level as well. The groups of EVT 1 are represented by a single symptom: (1-1-1-0, 1-0-2-0). EVT 2 had three symptoms and an event strength of 4.54:

$$(1)\ (\ 1\text{–}0\text{–}2\text{–}0\ ,\ 0\text{–}0\text{–}2\text{–}1\ ,\ 1\text{–}1\text{–}7\text{–}2\ )$$

$$(2)\ (\ 1\text{–}0\text{–}2\text{–}0\ ,\ 1\text{–}1\text{–}1\text{–}0\ )$$

$$(3)\ (\ 1\text{–}0\text{–}2\text{–}0\ ) \qquad (4.4)$$

This strength again indicates that the error records in EVT 2 are related to a single cause.

When analyzed at the super event level, however, we find that there is no longer enough statistical support to indicate that the super event of Figure 5 was due to a single cause. The super symptom set is found by taking the union of the symptoms generated by EVT 1 and EVT 2 and is simply the first two symptoms of Equation (4.4). The first super symptom appears independently four different times among all the events generated from the CYBER-175 data. The second super symptom appears eight times independently. By Equation (3.1), the strength of this super event is only 0.38. This low strength indicates that the super event consists of errors due to unrelated causes. The super symptoms occur independently so frequently that their joint appearance in the super event of Figure 5 cannot be considered significant.

Two specific procedure of the proposed methodology, namely, the subset based and truncated analyses, have not been illustrated in the examples discussed so far. Examples of these procedures can be found in a related technical report [Sridhar 85].

```
                                 Q
  R                  E C  S      U C
  E                  R H  Y      A H
  C                  R A  N  C P D A
  T            T     T N  R  H P R S B C
  Y            I     Y N  O  P P A S A H
  P            M     P E  M  A A N I N I
  E            E     E L  E  R R T S K P

  3 84May18 03:12:59 S 14 23 0 0 1 1 1 0
  3 84May18 03:14:01 S 14 89 0 0 1 0 2 0
  3 84May18 03:15:12 S 14 23 0 0 1 1 1 0

E 3 84May18 04:00:06 S 14 92 0 0 0 0 2 1
V 3 84May18 04:00:17 S 14 23 0 0 1 1 1 0
T 3 84May18 04:00:31 S 14 92 0 0 0 0 2 1
  3 84May18 04:01:09 S 14 A1 0 0 1 1 7 2
1 3 84May18 04:01:13 S 14 89 0 0 1 0 2 0
  3 84May18 04:01:24 S 14 92 0 0 0 0 2 1
  3 84May18 04:01:55 S 14 23 0 0 1 1 1 0
  3 84May18 04:02:30 S 14 92 0 0 0 0 2 1
  3 84May18 04:03:19 S 14 89 0 0 1 0 2 0
 ----------------------------------------
  3 84May19 05:00:21 S 14 92 0 0 0 0 2 1
  3 84May19 05:01:13 S 14 89 0 0 1 0 2 0
  3 84May19 05:01:53 S 14 A1 0 0 1 1 7 2
  3 84May19 05:02:31 S 14 92 0 0 0 0 2 1
  3 84May19 05:02:54 S 14 89 0 0 1 0 2 0
  3 84May19 05:02:58 S 14 92 0 0 0 0 2 1
  3 84May19 05:03:40 S 14 A1 0 0 1 1 7 2
  3 84May19 05:03:59 S 14 23 0 0 1 1 1 0
E
V 3 84May19 06:00:09 S 14 92 0 0 0 0 2 1
T 3 84May19 06:00:10 S 14 A1 0 0 1 1 7 2
  3 84May19 06:00:11 S 14 89 0 0 1 0 2 0
2
  3 84May19 07:01:04 S 14 23 0 0 1 1 1 0
  3 84May19 07:04:15 S 14 89 0 0 1 0 2 0
  3 84May19 07:04:30 S 14 23 0 0 1 1 1 0
  3 84May19 07:04:31 S 14 89 0 0 1 0 2 0
  3 84May19 07:04:52 S 14 23 0 0 1 1 1 0
  3 84May19 07:04:55 S 14 89 0 0 1 0 2 0
  3 84May19 07:07:11 S 14 23 0 0 1 1 1 0
  3 84May19 07:07:27 S 14 89 0 0 1 0 2 0
  3 84May19 07:07:48 S 14 23 0 0 1 1 1 0
```

Figure 5:
Example of Super Event Due to Unrelated Causes

5. PERFORMANCE OF THE METHODOLOGY

As a final check on the performance of the methodology, we obtained independent corroboration of our results from the log of failures and repair maintained by the field engineers. Although this recording was not always complete, it did allow us to perform an independent check on our evaluation. The results of this check are summarized below.

In nearly 85% of the cases the engineers were directly able to confirm that our validated superevents corresponded to real system problems. The evaluation was made both on the basis of their experience and from their field maintenance logs. The confirmed cases corresponded to single event superevents.

For the remaining 15% of the cases, although the engineers could not confirm "noticing" a valid problem, they agreed that (with the benefit of hindsight) that a problem had existed. However, the manifestation (an event) was not severe enough to be noticed by their analysis. One important case was the superevent consisting of ten events, discussed in section 4.2. This superevent spans ten weeks. Although, each event by itself was not severe enough to show up, when taken as a whole it was clear that a persistent problem had existed. It appears that as a result of our information corrective action was taken because this particular superevent does not exist in the data which has been gathered since then. In summary, we find that not only do our results agree with field findings but also we are able to relate problems over a long period of time which would otherwise go undetected.

The proposed methodology, thus, can be used to evaluate the goodness of any diagnostic system, whether primitive or sophisticated. This can be done by analyzing the relationship among the errors the diagnostic system detects as isolated. A very simple measure for the goodness of diagnosis of a system could be given by

$$1 - \frac{\textit{number of error entries coalescing into related errors}}{\textit{total number of error entries in the error}\ \text{log}}$$

6. CONCLUSION

This paper has developed a methodology for automatically detecting symptoms associated with persistent errors in computer systems. The methodology was shown to work on two large computer systems in a multisystem configuration. The recognition process is based on probabilistic techniques and is implemented in three levels. Each level adds confidence to the results. The power of the methodology to allow for the resolution between single and multiple but independent causes was also demonstrated. With multiple but overlapping symptoms, the methodology determines the number of distinct independent symptoms. The methodology quantifies the strength of the relationship among related errors. Research is now in progress to develop a suitable diagnostic system that will use the extracted error symptoms and automatically carry out diagnosis and recovery. This would not only facilitate better diagnosis and recovery management, but will also improve the overall reliability and performance of a system.

ACKNOWLEDGEMENTS

The authors thank George Badger, Director CSO, Warner Brigham, Sandy Moy and Milt Cloud at CSO for providing insight into the measured systems and access to the data. This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under contract number N00014-84-C-0149 and in part by IBM corporation.

APPENDIX
SYSTEM DESCRIPTION

This analysis uses detailed error data automatically collected over a six-month period by the operating system. As with other large systems, the CYBER machines maintain a log of a variety of normal and abnormal events -- called the "system dayfile." The abnormal events are errors automatically detected by the system. Errors can originate in the peripheral processors, the central processor, main memory, or the disk subsystem. The errors are captured by hardware that checks the integrity of information being transmitted between physical elements in the system. When an error is detected, a system routine collects pertinent information from the hardware concerning the state of the machine at the time of the error and stores it in the dayfile. A decoded sample of the information logged is shown Figure A.1.

The fields of information include:

(1) record type: specifies the format of the rest of the record (RECTYPE)

(2) error code: specifies the nature of the error (ERR)

(3) time and date of error

(4) channel in use at the time of error (CHAN)

(5) equipment number of the device involved (EST)

The three major record types are:

(1) deadstarts (RECTYPE = 0001): these are logged for all scheduled and unscheduled system restarts. The two common deadstart levels are (a) deadstart level zero which denotes a complete halt of the machine with no job recovery possible, and (b) deadstart level three which implies that the integrity of the machine state is maintained and recovery is possible for most jobs.

(2) disk subsystem errors (RECTYPE = 0024): these included disk checkword errors, channel parity errors, and device contention errors.

(3) parity-related errors (RECTYPE = 0003): these include memory errors (single error correction, double error detection), peripheral processor errors, channel errors, extended core parity and central memory controller errors.

REFERENCES

[Brown 81] J. S. Brown, et. al., "Pedagogical and knowledge engr. techs. in the SOPHIE systems," Xerox CIS-14,1981.

[Davis 82] R. Davis, *et al*, "Diagnosis based on structure and function," AAAI-82, Pittsburgh, 1982, pp. 137-142.

[deKleer 76] "Local methods for localizing faults in electronic circuits," Vol. PAMI-4, No. 3, May 1982.

[Genesereth 82] M. Genesereth, "Diagnosis using hierarchical design models," AAAI-82, Pittsburg, 1982.

[Iyer 85] R. K. Iyer, P. Velardi, "Hardware-related software errors," *IEEE Tr. SWE.*, vol. SE-11, Feb 1985.

[Iyer 86] R. K. Iyer, et. al., "Measurement and modeling of comp. rel. as affected by syst. activity," *ACM TOCS*, Aug 1986.

[Sanders 85] D. Sanders, M.S. thesis, ECE-UIUC, 1985.

[Shubin 82] H. Shubin and J. W. Ulrich, "IDT: An intelligent diagnostic tool," AAAI-82, Pittsburg, 1982.

[Sridhar 85] V. Sridhar, et. al., "Recognition of error symptoms in large systems," CSG-46, CSL, UIUC, 1985.

[Tsao 83a] M. M. Tsao, "Trend analysis and fault prediction," Tech. Rep. CMU-CS-83-130, Carnegie-Mellon University, 1983.

[Tsao 83b] M. M. Tsao and D. P. Siewiorek "Trend analysis on system error files" FTCS-13, 1983.

[Velardi 84] P. Velardi, R. K. Iyer, "A study of software failures and recovery in MVS," *IEEE Tr. Comp.*, vol. C-33, 1984.

OBS	TIME	REC-TYPE	ERR	CHAN	EST	SYN	POSIT	QUAD	CSU	BANK	CHIP	ERR-TYPE
1	19MAY84:2:06:12	0024	CHKWD	5	10	.	266051	.	.	.	.	.
2	19MAY84:5:00:21	0003	PARITY	.	.	92	.	0	0	2	1	MEM
3	19MAY84:5:00:57	0003	PARITY	.	.	92	.	0	0	2	1	MEM
4	19MAY84:5:01:13	0003	PARITY	.	.	89	.	1	0	2	0	MEM
5	19MAY84:5:01:53	0003	PARITY	.	.	A1	.	1	1	7	2	MEM
6	19MAY84:5:02:31	0003	PARITY	.	.	92	.	0	0	2	1	MEM
7	19MAY84:5:02:54	0003	PARITY	.	.	89	.	1	0	2	0	MEM
8	19MAY84:5:02:58	0003	PARITY	.	.	92	.	0	0	2	1	MEM
9	19MAY84:5:03:40	0003	PARITY	.	.	A1	.	1	1	7	2	MEM

Figure A.1: Sample From Error Log

METASAN: A Performability Evaluation Tool Based on Stochastic Activity Networks

W. H. Sanders and J. F. Meyer

Computing Research Laboratory
Dept. of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan, USA 48109
and
Communications and Network Laboratory
Industrial Technology Institute
Ann Arbor, Michigan, USA 48106

Abstract

This paper describes a software package that assists the construction and solution of performability models based on stochastic activity networks. Stochastic activity networks, a generalization of stochastic Petri nets, permit the representation of concurrency, timeliness, fault tolerance and degradable performance in a single model. Their structure allows for direct construction of analytically tractable stochastic base models under certain well defined conditions, as well as simulation under very general conditions. For a given system, the input to METASAN consists of two parts: a description of the structure of the net, and a description of the desired performance variables and solution method. Each part is specified in a flexible input language. This language allows the analyst to define very general performance variables in terms of the behavior of the net. The user interface and solution methods employed are reviewed, and an example is given illustrating use of the package.

1. Introduction

Extensions to Petri nets have proved to be valuable tools for both the verification and evaluation of concurrent systems. Many of these extensions have included the addition of an explicit representation of time. In particular, one approach has been to add timing of a probabilistic nature to transitions in the net. This permits the representation of timeliness as well as concurrency in a stochastic setting. As models for performability evaluation [1], they also permit the representation of fault tolerance and degradable performance. Stochastic activity networks (SANs) [2, 3, 4] were developed to facilitate the performability evaluation of systems which exhibit any or all of these four characteristics. Through the introduction of several new primitives, they allow the model to be specified in a convenient way, while providing the formal structure necessary for analytic solutions. When model characteristics preclude analytical evaluation, performability can be evaluated via simulation.

Others have likewise examined the use of stochastic extensions of Petri nets for performance and/or reliability evaluation. Solutions via analysis, and to a lesser extent, simulation have been considered. Notable in the area of analysis is the work of Natkin ([5]; Timed Petri Nets), Molloy ([6]; Timed Petri Nets), Marsan et al. ([7]; Generalized Stochastic Petri Nets), Chiola ([8]; Generalized Stochastic Petri Nets), Cumani ([9]; Stochastic Petri nets with phase type distributions), and Dugan et al. ([10]; Extended Stochastic Petri Nets). Examples of solution via simulation include work by Behr et al. ([11]; Evaluation Nets), Dugan ([12]; Extended Stochastic Petri Nets), Chiola ([8]; Generalized Stochastic Petri Nets), Godbersen and Meyer ([13]; Function Nets), and Törn ([14]; Simulation Nets).

In what follows, we assume familiarity with the basic structure and execution of SANs [2, 3]. For a brief review of basic concepts and terminology, consider the stochastic activity network of Figure 1. Activities ("transitions" in Petri net terminology) are of two types, timed and instantaneous. Elongated ovals represent *timed activities* (e.g., compute1) and solid bars represent *instantaneous activities* (e.g., disk1). Timed activities are used to represent activities of the modeled system whose durations impact the system's ability to perform. Instantaneous activities represent system activities which, relative to the performance variable in question, complete in a negligible amount of time. Places are depicted as circles (e.g., A-J) and, as with Petri nets, each place can hold a nonnegative number of *tokens*. The distribution of tokens in the places of the network at a given time constitutes the *marking* of the network at that time.

Cases can be associated with both timed and instantaneous activities and are represented by small circles (e.g., as in compute1). Cases permit the realization of two types of spatial uncertainty. Uncertainty about which activities are enabled in a given marking is realized by cases associated with intervening instantaneous activities. Uncertainty about the next marking assumed upon completion of a timed activity is realized by cases associated with that activity. The two remaining net primitives are input gates and output gates. Input gates (e.g., computein1) contain both an enabling predicate and input function (on the marking of the places). The enabling predicate must be true for the activity associated with that gate to be enabled. Upon completion of the associated activity, the input function is executed, possibly changing the marking of the net. Output gates (e.g., diskout1) have a single output function (on the marking of the places) associated with them, which is executed upon completion of the associated activity.

The stochastic nature of the nets is realized by associating an activity time distribution function with each of the timed activities and a probability distribution with each set of cases. A *reactivation function* [3] is also associated with each timed activity. This function specifies, for each marking, a set of *reactivation markings*. Informally, given that an activity is activated in a specific marking, the activity is *reactivated* whenever any marking in the set of reactivation markings is reached. This provides a mechanism for restarting activities that have been activated, either with the same or different distribution. Note that this decision is made on a per activity basis (based on the reactivation function), and is not a net-wide execution policy.

As stochastic activity network models grow in size and complexity, analysis by hand quickly becomes intractable. Both the growth in possible markings and complexity of activity time distributions contributes to this difficulty. Such considerations have resulted

CH2345-7/86/0000/0807$01.00 © 1986 IEEE

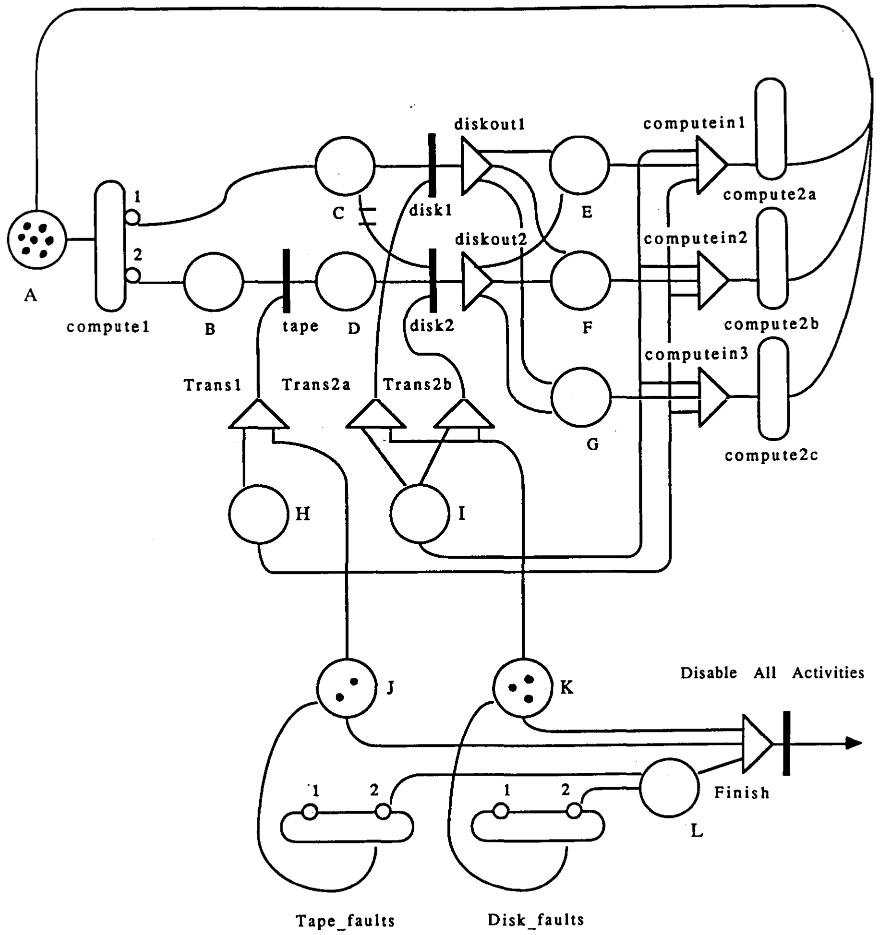


Figure 1. Example Stochastic Activity Network

in the development of METASAN (Michigan Evaluation Tool for the Analysis of Stochastic Activity Networks). METASAN contains routines for both model construction and model solution, and contains both simulation and analytical solvers.

In the discussion that follows, Section 2 describes the high level organization of METASAN. Section 3 discusses model construction and Section 4 describes model solution. Use of the package is illustrated in Section 5 via its application to a specific evaluation problem. Finally, suggestions for further work are discussed in the concluding section.

2. METASAN Organization

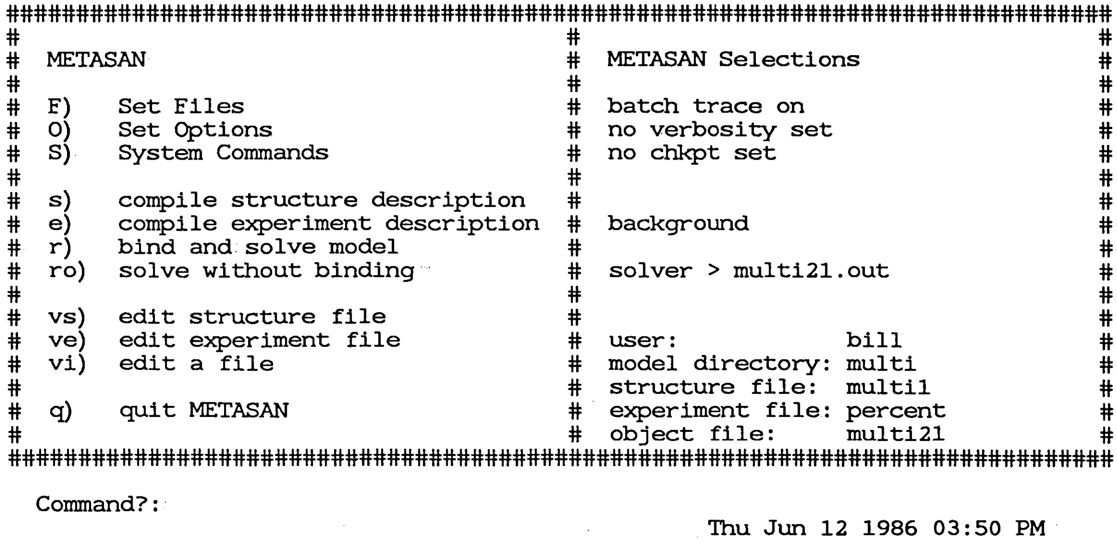
METASAN was designed in a modular manner to allow for the addition of new solution methods as these become available. It is written using UNIX tools (C, Yacc, Lex, and Csh) and currently contains some 30,000 lines of source code. Steady-state and transient evaluation, via both analysis and simulation, are supported by the tool.

At the highest level, the analyst interacts with METASAN through a menu structure. This menu (Figure 2) permits access to the two basic files that make up a METASAN model: a structure file and an experiment file. The menu also permits access to the compilers for each of the description files and the solution modules. The structure file is a direct translation of the SAN into a textual form that can be accepted by the package. Specification of desired performance variables and solution algorithm is done via the experiment file. Each of these files will be described in more detail in Section 3. Model construction consists of describing the structure of the system to be modeled using the editor (option vs), compiling the description (option s), describing the experiment file (option ve) and compiling the experiment file (option e). The result of these actions is a machine understandable description (a collection of C data structures and procedures) of the system to be modeled and the desired performance variables. By then selecting model solution (option r) this machine readable description is bound to the correct solution module and is executed. A flowchart of this process is given in Figure 3. Output from the solver depends on the solver chosen, but contains either estimates or exact values of the chosen performance variables.

Items in the pictured menu that are identified by a capital letter designate lower level menus. "Set Files" allows the user to specify each of the files associated with a particular model. "Set Runtime Options" offers a variety of options which, depending on the solution module selected, allow the user to set various trace and debug options, run the model in the background or on remote machines, and direct the output to several locations. "System Commands" consists of a collection of commands to aid the user in creating new model directories, removing or copying description files, etc.

3. Model Construction

In the context of METASAN, model construction consists of describing both the stochastic activity network and chosen performance variables in a manner understandable to the package and translating these descriptions into a form that is understandable to the solution modules. This is accomplished in METASAN via two input languages and their corresponding compilers.



```
###############################################################################
#                                        #                                    #
#   METASAN                              #   METASAN Selections               #
#                                        #                                    #
#   F)    Set Files                      #   batch trace on                   #
#   O)    Set Options                    #   no verbosity set                 #
#   S)    System Commands                #   no chkpt set                     #
#                                        #                                    #
#   s)    compile structure description  #                                    #
#   e)    compile experiment description #   background                       #
#   r)    bind and solve model           #                                    #
#   ro)   solve without binding          #   solver > multi21.out             #
#                                        #                                    #
#   vs)   edit structure file            #                                    #
#   ve)   edit experiment file           #   user:            bill            #
#   vi)   edit a file                    #   model directory: multi           #
#                                        #   structure file:  multi1          #
#   q)    quit METASAN                   #   experiment file: percent         #
#                                        #   object file:     multi21         #
###############################################################################

  Command?:
                                                     Thu Jun 12 1986 03:50 PM
```

Figure 2. METASAN Main Menu

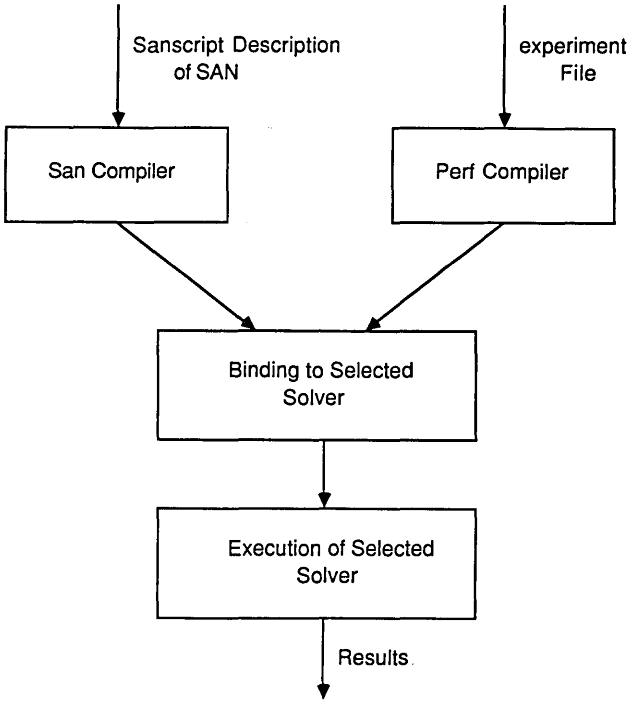


Figure 3. METASAN Flow Diagram

3.1. Sanscript and the SAN Compiler

The SAN description language, *Sanscript*, allows the analyst to specify the SAN in a textual form understandable to the (SAN) compiler. Sanscript permits easy specification of complex enabling predicates, activity time functions, reactivations functions and gate functions. To give the reader a feel for Sanscript, we describe the language via the example presented earlier. Figure 4 is the Sanscript representation of the portion of the stochastic activity network that represents variations in the internal state and environment of the system; activities representing fault occurrences are not represented. At a high-level, a Sanscript description consists of four parts: a header, local variable declarations, definition of all the primitives used, and a specification of all functions, values, and interconnections associated with each primitive.

In our example, the header specifies that the name associated with this SAN is multi1. The local variable declaration section is not used. Next comes the declaration of primitives. Note that the initial marking for each place is specified directly after the place name. The specification section follows. In any function, the current marking of the place can be referenced by using the notation *MARK*(*place*), where "place" is the name of the desired place. For an example specification, refer to the definition of compute1. As can be seen in the graphical representation, timed activity "compute1" has two cases, one connected to place C, and one to place B, and a single input place A. The activity time distribution is specified to be exponential with parameter 1.0 * *MARK*(*A*) and the case distribution is specified such that there is a 30% chance of choosing case 1 and 70% chance of choosing case 2 upon completion of the activity. The reactivation function is specified so that for every marking (denoted by the 1 in the first set of brackets after "react") the set of reactivation markings is all markings (denoted by the second 1 in brackets). In other words, the activity is reactivated at every state change. The reason for this will be discussed in the example section. Note that any expression that evaluates to the correct type (real for prob and exp, boolean for react) may be given between the curly brackets; complex interactions, activity time descriptions, reactivation functions, and case distributions can be represented easily via a few C statements.

A host of activity time distribution types are available, representing all service distributions normally used in evaluation. An activity time distribution function of "inst" is used to denote an instantaneous activity. Of course, the choice of distribution affects the nature of the underlying stochastic process and the solution methods that may be used. Complex reactivation functions can be represented by specifying several pairs of predicates, the interpretation being that the activity is reactivated if, a) the activity was activated in the subset of the reachable markings specified by the first predicate, and b) a marking in the subset specified by the corresponding second predicate is reached. Again, if the activity is enabled in this second marking it is immediately restarted.

Gates are specified in a similar manner. For example, see the description of the input gate "computein1" in both the graphical and textual representation. The input places associated with the gate are specified as an indexed list. This abbreviated notation allows place names to be abbreviated in the predicate and function descriptions according to the following rule: "Xi" denotes the i-th place specified in the associated input or output (in output gates) place specification.

```
DESCRIPTION multi1;

OBJECTS

place:          A,7;            /* definition of places */
                B,0;
                C,0;
                D,0;
                E,0;
                F,0;
                G,0;
                H,0;
                I,0;
                J,2;
                K,3;
activity:       compute1;       /* definition of activities */
                tape;
                disk1;
                disk2;
                compute2a;
                compute2b;
                compute2c;
input_gate:     computein1;     /* definition of input gates */
                computein2;
                computein3;
                Trans1;
                Trans2a;
                Trans2b;
output_gate:    diskout1;       /* definition of output gates */
                diskout2;

SPECIFICATION

compute1        [ cases 1: prob { .3 } C;
                        2: prob { .7 } B;
                  input A;
                  exp { 1.0 * MARK(A) };
                  react { 1 } { 1 }; ]
tape            [ case D;
                  inputs B; Trans1;
                  inst; ]
disk1           [ case diskout1;
                  inputs C; Trans2a;
                  inst; ]
disk2           [ case diskout2;
                  inputs ~C; D; Trans2b;
                  inst; ]
compute2a       [ case A;
                  input computein1;
                  determ { 1.0 * MARK(E) };
                  react { 0 } { 0 }; ]
compute2b       [ case A;
                  input computein2;
                  determ { 1.0 * MARK(F) };
                  react { 0 } { 0 }; ]
compute2c       [ case A;
                  input computein3;
                  determ { 1.0 * MARK(G) };
                  react { 0 } { 0 }; ]
diskout1        [ outputs 1: E; 2: F; 3: G;
                  func { if (X1 == 0) X1 = 1;
                         else if (X2 == 0) X2 = 1;
                         else if (X3 == 0) X3 = 1; } ]
diskout2        [ outputs 1: E; 2: F; 3: G;
                  func { if (X1 == 0) X1 = 2;
                         else if (X2 == 0) X2 = 2;
                         else if (X3 == 0) X3 = 2; } ]
computein1      [ inputs 1: I; 2: E; 3: H;
                  pred { X2 >= 1 }
                  func { if (X2 == 2) {
                           X1 = X1 - 1; X3 = X3 - 1; }
                         else
                           X1 = X1 - 1;
                         X2 = 0; } ]
computein2      [ inputs 1: I; 2: F; 3: H;
                  pred { X2 >= 1 }
                  func { if (X2 == 2) {
                           X1 = X1 - 1; X3 = X3 - 1; }
                         else
                           X1 = X1 - 1;
                         X2 = 0; } ]
computein3      [ inputs 1: I; 2: G; 3: H;
                  pred { X2 >= 1 }
                  func { if (X2 == 2) {
                           X1 = X1 - 1; X3 = X3 - 1; }
                         else
                           X1 = X1 - 1;
                         X2 = 0; } ]
Trans1          [ inputs 1: H; 2: J;
                  pred { X2 > X1 }
                  func { X1 = X1 + 1; } ]
Trans2a         [ inputs 1: I; 2: K;
                  pred { X2 > X1 }
                  func { X1 = X1 + 1; } ]
Trans2b         [ inputs 1: I; 2: K;
                  pred { X2 > X1 }
                  func { X1 = X1 + 1; } ]

end.
```

Figure 4. Example Description File

An example of this can be seen in both the predicate and input function description of computein1. Note again that any legal C statements may be placed within the brackets, as well as the abbreviated notation described above and the *MARK* function notation. Output gates are specified in an identical manner except that the keyword "outputs" is used in place of "inputs" and there is no associated enabling predicate.

Many stochastic activity networks contain numerous similar subnetworks (e.g., nodes in a computer network) that are replicated many times. Construction of these subnetworks directly in Sanscript would be tedious. To make this easy, we have recently completed development of a macro-preprocessor for METASAN. This preprocessor allows one to define subnetworks once in a parameterized manner, and then construct a specific subnetwork via a single macro call.

After the specification of the SAN in Sanscript is complete, it is passed to the SAN compiler to be translated into an internal form understandable by the solution modules. The SAN compiler is written in Yacc (yet another compiler compiler), Lex and C.

3.2. Performance Variables

In order to understand the available performance variables, it is necessary to characterize the execution of a stochastic activity network in a more formal manner than is found in [2]. This is done by introducing the notion of a configuration and a step. A ***configuration*** is a triple $<\mu, a, c>$ where μ is a marking, a is an activity, and c is a case of that activity. A SAN is said to be in configuration $<\mu, a, c>$ if μ is the current marking of the network, activity a will complete next, and case c will be chosen. A ***step*** is a triple $(<\mu, a, c>, t, s)$ where $<\mu, a, c>$ is a configuration, t is the time of activation of activity a, and s is the activity time of activity a. Furthermore, an ***execution of a stochastic activity network*** is a sequence of sets of steps $S_1, S_2, \cdots, S_n, \cdots$.

Except for markings in which more than one instantaneous activity is enabled, each set in the sequence of steps is a singleton and corresponds to the state of the network at the specified time. Multiply enabled instantaneous activities lead to sets of cardinality greater than one. A detailed understanding of the execution of the network in this situation is not necessary to understand what follows. A more precise definition of execution is found in [15].

In order to specify general performance variables in terms of the execution of the net, it is helpful to introduce the notion of a path. A ***path*** is a finite sequence of configurations $C_1, C_2, C_3, \cdots, C_n$ such that for each pair of configurations $<\mu_i, a_i, c_i><\mu_{i+1}, a_{i+1}, c_{i+1}>$ the completion of a_i and choice of c_i in μ_i results in μ_{i+1}. During an execution a path may be traversed. The following discussion makes this more precise. A ***completion of a configuration*** $<\mu, a, c>$ occurs at time $t+s$ when during an execution of a SAN the configuration is found in a step $(<\mu, a, c>, t, s)$ in some set of steps. An ***initiation of a path*** occurs when the SAN reaches the marking associated with the first configuration in the path. A ***completion of a path*** occurs when the SAN completes the final configuration in the path after completing each configuration in the path in the order specified without reaching any intermediate configurations other than those specified in the path. A ***traversal of a path*** is the act of first initiating the path and then completing the path.

These definitions allow us to introduce a set of performance variables to study the behavior of the SAN. These variables are listed below. Note that they are indexed either by time or number of occurrence so that, if they converge in distribution, their limit can be studied.

$N_T(f,t)$

a measure of the value of the function f at time t+. f can be any real-valued function on the marking of the net. Examples include the marking of a single place and the sum of the markings of several places.

$T_b(S,n)$

the time between n-1*th* and *nth* completion of any path in a set of paths S.

$T_w(S,n)$

the time to traverse the *nth* path to complete in a set of paths S.

$I(T,t)$

an indicator random variable denoting the event of being in any configuration in a set of configurations T at time t+.

$N_{TR}(S,t_1,t_2)$

the sum of the number of traversals of any path in a set of paths S during $[t_1,t_1+t_2]$.

$T_I(S,t_1,t_2)$

the sum of the time spent traversing all paths in a set of paths S during $[t_1,t_1+t_2]$.

Note that there is a chance, depending on the definition of the sets of paths, that two paths may complete at the same time. Since some of the random variables defined above are indexed by the order of completion of the paths, a rule must be given to resolve any ambiguity as to which path is which. To do this we use the convention that if two paths complete at the same time, the path which took the longest time to traverse will be counted as completing first.

By varying the index of any of the first four variables above, one can construct a sequence of random variables. Under certain conditions, the distributions of the random variables in this sequence may converge to a single (steady-state) distribution. More precisely, following the terminology in [16] , for a sequence of random variables V_n, $n=1,2,3, \cdots$, we define

$$F_V(x) = \lim_{n\to\infty} P\{V_n \le x\}.$$

In terms of the random variables defined above, we have

$F_{N_T(f)}(x) = \lim_{t\to\infty} P\{N_T(f,t) \le x\}$

The distribution of the value of function f in steady-state.

$F_{T_b(S)}(x) = \lim_{n\to\infty} P\{T_b(S,n) \le x\}$

The distribution of the time between completions of any path in a set of paths S in steady-state.

$F_{T_w(S)}(x) = \lim_{n\to\infty} P\{T_w(S,n) \le x\}$

The distribution of the time between initiations and completions of any path in a set of paths S in steady-state.

$F_{I(T)}(x) = \lim_{t\to\infty} P\{I(T,t) \le x\}$

The distribution of the event of being in any configuration in a set of configurations T in steady-state.

Not all solution methods will yield results for all of the performance variables listed above. Currently, all can be obtained using an appropriate simulation (i.e. terminating or steady-state), but analytical methods are limited to those that can be determined from the stochastic process representing the changes in marking of the SAN as a function of time. Work is currently underway to extend analytical methods to capture activity related behavior.

3.3. Experiment File and Perf Complier

Instances of the above performance variables, along with their corresponding path sets, are specified in an experiment file. The exact form of this file varies from solver to solver. In general, the experiment file specifies the paths sets, variables defined on those paths, definitions of any variables derived from previous variables, and characteristics of the defined variables desired (e.g., mean, PDF, variance). For the simulation solvers, these characteristics take the form of mean, variance, interval, and percentile estimators with confidence intervals; for analytic solution they are exact.

For example, consider an experiment file (Figure 5) for the example of the SAN of Figure 1. Here the goal is to obtain the mean, variance, an interval, and several percentile estimates of the the time between firings of activity "compute1". Because the activity time distributions of activities compute2a, compute2b, and compute2c are uniform, the steady-state simulation solver is used. The experiment file requires definitions of configurations, path sets, performance variables, and estimators for the defined variables. Configurations are specified in a compact notation and path sets are built up from sets of configurations. The meta-character "*" can be used to denote any marking, activity, or case, depending on its position. In the example, (MARK(A) == 0, (*,*)) specifies all configurations such that the marking of A is 0, any activity completes (denoted by the first *), and any case is chosen (denoted by the second *). "==TIMED" and "==INST" can also be used in the activity specification to denote any timed or instantaneous activity, respectively. Path sets are then constructed from these configuration sets. The syntax "||" can be used to "or" configuration sets together at either the CONFIGURATION or PATHSET level in the experiment file. Although all of the path sets in our example consist of a single configuration set, longer paths can be specified by naming a sequence of configuration sets separated by commas. The interpretation of this notation (in the context of our earlier theory) is that the set of paths included is all paths constructed by selecting a configuration from each of the configuration sets in order. Two additional operators aid in the definition of path sets. "*( )" denotes that the sequence of configurations within the parenthesis must be repeated 0 or more times. "+( )" denotes that the sequence of configurations within the parenthesis must be repeated 1 or more times.

The user then specifies the desired performance variables in terms of the previously defined path sets. Each of these variable definitions is itself implemented in a low-level language that makes it easy to create new variables. Experiment file section ESTIMATIONS allows for specification of the estimator to be defined on the variables. Currently, mean, variance, interval, and percentile estimators are supported, all with confidence interval estimation. Examples of the syntax for each of these estimators can be found in Figure 5. In each case the user specifies first the relative confidence interval width (.01 in the example) and confidence level (95%). For the interval estimator the user also specifies the bounds for the interval to be considered. In the example, the bounds indicate that the probability that the time is between .5 and 1.5 is to be estimated. For the percentile estimator the user specifies the percentile for which the estimate is desired (.05, .35, .70, and 1.0 in the example).

Similar experiment files are used in conjunction with the analytic and terminating simulation solvers. In each case, the file specifies to the package the performance variables and the characteristics of those variables that are to be determined. The experiment file is then passed to the perf compiler, where the specification is translated into a second collection of C data structures and procedures that are understandable to the appropriate solver.

4. Model Solution

After the Sanscript and experiment files are converted to their internal machine form, they are bound to the chosen solver. Depending on the characteristics of the model to be solved, the solver can be one of several analytic or simulation solvers. Each solver will be discussed in some detail below.

4.1. Analytic Solvers

Model solution, in the context of analytical solution methods, refers to the solution of a base stochastic process for the desired performance variables [17]. In light of this, generation of the stochastic process realized by a specific stochastic activity network is properly part of model construction. However, because generation is specific to analytic solutions, it is incorporated as part of each analytic solver and hence discussed under model solution. In our framework, this consists of generating a ***stochastic activity system (SAS)*** [3] realized by the stochastic activity network. Stochastic activity systems can be regarded as probabilistic extensions of Keller's "named transition systems" [18, 19]. The most detailed description of an activity system's behavior are its ***state-activity*** sequences, i.e., for a given state, the possible sequences of alternating states and activities that can result from a finite number of applications of the transition relation. In the solution techniques implemented to date, we have not exploited the activity related behavior, just state related behavior. The state behavior of a stochastic activity system is a stochastic process that can serve as the base model of a performability model [1].

As can be seen in Figure 6, all analytic solution methods require generation of the stochastic activity system realized by the SAN model. Informally, this is done as follows. First, the set of activities which are enabled in the initial marking is found, by examining the enabling predicate of each input gate. If the enabling predicates for all input gates connected to an activity are true, then the activity is enabled. If any instantaneous activities are enabled in this initial marking, then this is not a valid SAN. Given that the initial marking is stable, the next step is to generate the next marking(s) of the SAN supposing that each activity enabled in the initial marking completes. A possible next marking is generated for each case of each activity. Then, for each of these new markings which is not already in the reachability set, the possible next marking(s) are computed (i.e. the set of enabled activities in each of these new markings is computed, as well as the resulting new marking(s) upon completion of each of these activities) except that now any enabled instantaneous activities must also be dealt with. When all next possible markings which are generated are already in the reachability set, all reachable stable markings have been found. These markings correspond to the states in the realized SAS. The activity time distributions and transition distribution for each labeled transition is then constructed as in definition 3.7 of [3], and the generation of the SAS realized by the structure submodel is complete.

Instantaneous activities complete whenever they are enabled, while timed activities complete only if they are enabled in stable markings. Therefore, whenever an unstable marking (a marking in which at least one instantaneous activity is enabled) is reached, the marking does not correspond to a state in the realized SAS. To find the next stable state, one must examine all possible next markings reached upon completion of this instantaneous activity as well as any other instantaneous activities which become enabled, generating new markings until the set of possible next markings consists of only stable markings. Because we do not know whether the SAN is well-behaved [2] (at the time it is constructed), we must keep track of all reachable unstable markings, until the set of next stable markings is found. The resulting realization procedure is, therefore, somewhat complicated and is omitted in the interest of brevity. It should be noted that the procedure requires human intervention to decide when a particular marking space is judged infinite, and hence the procedure stopped. The result of this procedure is a machine representation of the stochastic activity system; after the nature of this process is determined, one of several techniques may be used to obtain the desired performance variables.

After the stochastic activity system corresponding to the SAN model is generated, a check must be made to determine the nature of the underlying stochastic process. The nature can be determined directly from the structure of the SAS. This check and the desired performance variables determine which analytic solver is to be used.

Solutions for many traditional performance variables can be formulated in terms of the steady-state state occupancy probabilities of the resulting stochastic process. When this process is Markovian, solution for these occupancy probabilities is done either by Gaussian

```
PERF percent;

CONFIGURATIONS

        CYCLE = ( *, (compute1,*) )
        CYCLEV = (*, (compute1,*) )
        CYCLE1 = ( *, (compute1,*) )
        CYCLE2 = ( *, (compute1,*) )
        CYCLE3 = ( *, (compute1,*) )
        CYCLE4 = ( *, (compute1,*) )
        CYCLEI = ( *, (compute1,*) )


PATHSETS

        Mean  = [ CYCLE ]
        Variance = [ CYCLEV ]
        Percentile1 = [ CYCLE1 ]
        Percentile2 = [ CYCLE2 ]
        Percentile3 = [ CYCLE3 ]
        Percentile4 = [ CYCLE4 ]
        Interval = [ CYCLEI ]


MEASURED VARIABLES

        SS_TB ( Est_Mean, Mean, 100, 3200);
        SS_TB ( Est_Var, Variance, 100, 3200);
        SS_TB ( Per_05, Percentile1, 100, 3200);
        SS_TB ( Per_35, Percentile2, 100, 3200);
        SS_TB ( Per_70, Percentile3, 100, 3200);
        SS_TB ( Per_100, Percentile4, 100, 3200);
        SS_TB ( Interval_1, Interval, 100, 3200);


ESTIMATIONS

        MEAN( Est_Mean, .01, .95)
        VARIANCE( Est_Var, .01, .95)
        PERCENTILE( Per_05, .03, .95, .05)
        PERCENTILE( Per_35, .01, .95, .35)
        PERCENTILE( Per_70, .01, .95, .70)
        PERCENTILE( Per_100, .01, .95, 1.0)
        PINTERVAL( Interval_1, .01, .95, .5, 1.5 )


HALTING CONDITIONS

RESOLUTION RULES

        { compute_2a, compute_2b, compute_2a }

TYPE
        steady_state

end.
```

Figure 5. Example Experiment File

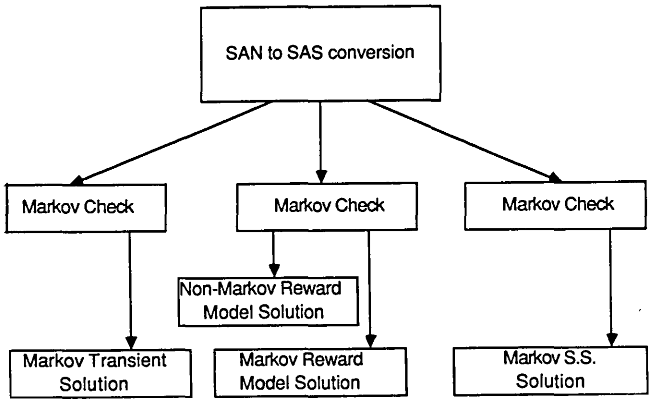

SAN to SAS conversion
Markov Check
Markov Check
Markov Check
Non-Markov Reward Model Solution
Markov Transient Solution
Markov Reward Model Solution
Markov S.S. Solution

Figure 6. Analytic Solvers

elimination using a full matrix technique, or by iterative methods using sparse matrix techniques, depending on the size of the state space. The Gaussian elimination technique employed makes use of an algorithm [20] that operates on blocks of columns of the transition matrix individually, in order to reduce paging by the virtual memory management system. This makes solution possible for quite large systems without resorting to iterative solution methods. When this technique becomes impractical, an iterative method based on the Gauss-Seidel technique [21] is used.

Reward model [22] techniques have been used successfully to solve many performance-reliability evaluation problems (for example, see [23, 24, 25, 26, 27]). In the context of performability evaluation, one typically constructs a reward model by associating a set of reward rates with the stochastic activity system realized by the structure submodel [3] (a SAN representing the structure related activities). Each reward rate represents the performance rate of the system in the respective structure state. This reward model then serves as the base model of a performability model. Reward model solution techniques for both Markov and non-Markov systems are implemented as solvers in METASAN.

Once again, applicable solution techniques depend on the nature of the base model and the performance variables. In the case of Markov reward models, a variation on a technique proposed by Goyal and Tantawi [26] is used. Our implementation computes a conditional performability distribution for each trajectory type using the technique described in [26]. Performability is then obtained via knowledge of the probabilities of trajectory types [24]. When the reward model is not Markov, a technique developed by Furchtgott and Meyer [24] is used. This solution technique, while it requires the underlying stochastic process to be acyclic and non-recoverable (i.e. reward rates non-increasing with time), it allows for solutions when the base model is not even semi-Markov. Selection of the appropriate algorithm is based on the Markov check described earlier.

Solution for the state occupancy probabilities at a specific time for Markov stochastic activity systems is accomplished using a randomization technique proposed by Gross and Miller [28, 29]. This technique provides for computationally efficient computation of approximate transient state occupancy probabilities by conditioning on the number of state transitions that may occur during the bounded interval under consideration. The approach is based on the known technique of subordinating a Markov chain to a Poisson process (see[30], for example).

4.2. Simulation Solver

Conditions exist when solution of the base model via analytic means becomes intractable. This can occur, for example, when complex reactivation functions are specified, activity time distributions are general, the desired performance variables are sufficiently complex, or the state space of the underlying stochastic process is extremely large or infinite. To fill this need, METASAN provides facilities for both terminating (transient) and steady-state simulation.

To do this we provide a discrete-event next-event time advance simulator core. Currently, two methods for confidence interval estimation are supported. The first is an iterative method based on the replication approach, and is used for terminating simulations. Using this method, one specifies the relative precision and level of confidence desired as part of the experiment file input. Confidence intervals for steady-state simulation are currently determined using an iterative batching procedure, where the user must specify the length of initial transient, batch size, relative precision desired, and level of confidence desired. Work is currently underway to permit estimation of confidence intervals using the regenerative and spectral techniques.

5. Example

5.1. System Description

Consider, for example, a distributed system where all resources needed are available locally except tape and disk drives. Whenever a disk or tape drive is needed, the processor requests one from a common pool. Time to process such requests is negligible, and the requests are either immediately granted, or the process is blocked. As resources become available, they are allocated to blocked processers in a FIFO manner. Each processor is running an identical application program, whose goal is to process blocks of data. Each block requires processing consisting of the following steps (see Figure 7): 1) computation for an exponentially (with parameter β) distributed amount of CPU time, 2) allocation of either a disk and tape, or just a disk, with fixed probabilities p and 1-p respectively, 3) computation for a deterministic amount of CPU time equal to (α * the number of resources requested), and 4) release of all resources allocated. The disk and tape drive are used only for temporary storage, and hence, are not specific to any processor. Upon completion of the processing of a block of data, the processor immediately begins processing another block.

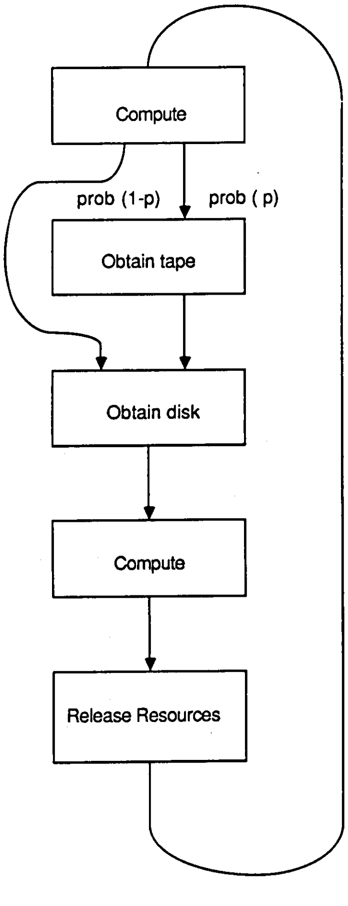


Figure 7. Processing Algorithm

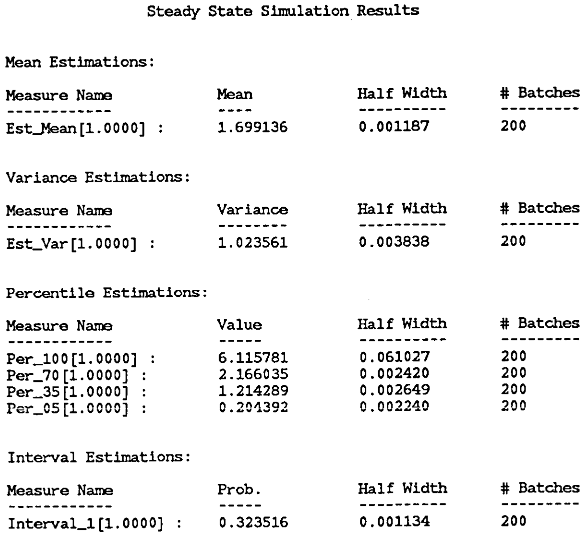
Faults can occur due to the failure of a disk or failure of a tape drive. In each case, the fault may be covered (i.e. the system degrades successfully to a less productive structure state) or it may result in a total loss of processing capability (i.e. total system failure). We assume further that faults in a tape drive and a disk occur as a Poisson process with rates λ and γ respectively. The performance variable considered is the number of blocks that are processed during a finite utilization period of t hours.

5.2. Construction of SAN model of system

Performability evaluation requires the construction of a SAN that corresponds to the system being modeled and meets the characteristics required by the particular solution algorithm. A seven processor case is examined. A SAN that meets these requirements is found in Figure 1. Here tokens represent the jobs executing on the processors and resources (tape and disk drives). Tokens in places A,B,C,D,E,F, and G represent the state of jobs executing on the processors. The marking on the diagram is the initial marking and corresponds to the state where all processors are executing the first compute in the sequence of events. Since the length of this compute time is exponentially distributed with parameter β for each processor, all first compute times can be represented via a single activity. The activity time distribution function for compute1 is, thus, exponential with parameter $\beta * MARK(A)$. Since the rate at which compute1 completes is determined by the number of tokens in place A, it should be reactivated at each state change to insure that the correct rate is always used. Each completion of compute1 corresponds to a processor completing the first compute. When this occurs, the process either requests both a tape and disk or a disk only. This choice is represented by the cases associated with compute1, where case 1 is chosen with probability (1-p) and represents an request for a disk only; case 2 is chosen with probability p and represents a request for a tape and disk.

These requests are processed by instantaneous activities disk1, disk2, tape, and places H and I. Place H represents the number of allocated tape drives. Place J represents the number of functioning tape drives. Input gate Trans1 determines whether there is an available tape drive. Completion of activity tape results in the allocation of a tape drive to the requesting process and the addition of a single token to place H. The number of tokens in place I represents the number of allocated disk drives and the tokens in place K represent functioning disk drives. Input gates Trans2a and Trans2b determine if there is an available disk. Similarly, completion of disk1 or disk2 represents the allocation of a disk drive to the requesting process. The function of gates diskout1 and diskout2 is to keep track of the resources that each process possesses during the second compute phase. Two tokens are placed in an output place if the process has both a tape and disk drive; one token signifies the process possesses only a disk. Activities compute2a, compute2b, and compute2c represent the second compute in the algorithm and have deterministically distributed activity times. The compute times cannot be represented by a single activity since the deterministic distribution is not memoryless. An activity is needed for each process in this phase. Since the maximum number of functioning disk drives in this example is three, only three processes can be in the second compute phase concurrently. Hence, only three activities are needed to represent this phase. Completion of each of these three activities represents the completion of the second compute for a process. The action of the input gate for the activity is to subtract the appropriate tokens from H and I to signify the release of the allocated resources. In addition, a token is added to the output place of the activity (place A) to indicate that the process is again beginning the first compute in the algorithm.

The arrival of faults and (possible) recovery of the distributed system is represented by the remaining places, input gates, and activities. Here places J and K represent the number of fault-free tape and disk drives respectively. Faults arrive to the system upon completion of activities Tapes_faults and Disk_faults. The activity time distributions of these activities are exponential with rates $\lambda * MARK(J)$ and $\gamma * MARK(K)$ respectively. Selection of case 1 of either activity represents successful recovery. In this case, one token is subtracted from the appropriate place to indicate the failure of the corresponding resource. Probabilities c_t and c_b are associated with case one of activities tape_faults and disk_faults, respectively. Selection of case 2 represents unsuccessful recovery. If this occurs, a token is placed in L. Completion of the instantaneous activity Disable signifies total system failure. This occurs when either recovery from a fault is unsuccessful (signified by a token in L) or when the pool of functioning resources is exhausted (zero tokens in J or K). When either of these events occur, instantaneous activity Disable completes and removes all tokens in the network. No blocks are processed in this state.



```
                 Steady State Simulation Results

Mean Estimations:

Measure Name           Mean           Half Width      # Batches
------------           ----           ----------      ---------
Est_Mean[1.0000] :     1.699136       0.001187        200


Variance Estimations:

Measure Name           Variance       Half Width      # Batches
------------           --------       ----------      ---------
Est_Var[1.0000] :      1.023561       0.003838        200


Percentile Estimations:

Measure Name           Value          Half Width      # Batches
------------           -----          ----------      ---------
Per_100[1.0000] :      6.115781       0.061027        200
Per_70[1.0000] :       2.166035       0.002420        200
Per_35[1.0000] :       1.214289       0.002649        200
Per_05[1.0000] :       0.204392       0.002240        200


Interval Estimations:

Measure Name           Prob.          Half Width      # Batches
------------           -----          ----------      ---------
Interval_1[1.0000] :   0.323516       0.001134        200
```

Figure 8. Example METASAN Output

5.3. Model Construction and Solution

In order to solve for the specified performance variable, we use the performability solution method described in [3]. Using this method, the analyst decomposes the SAN into two submodels, a ***performance submodel*** and a ***structure submodel***. Informally, the performance submodel contains all activities that represent variations in the internal state and environment of the system; the structure submodel contains all activities which represent variations in the system due to a change in system structure. In Figure 1, the places, activities, and gates above and including J and K comprise the performance submodel. The places, activities, and gates below and including J and K comprise the structure submodel. Places J and K are ***common*** places, and represent the structural configuration of the system.

As described in [3] the evaluation consists of determining a reward rate corresponding to each structure state, and solving the resulting reward model. In our example, the reward rate is determined by noting that each completion of compute1 corresponds to a completion of processing on a block of data. Hence, the rate of completion of block processing is just the inverse of the expected time between completions of compute1. In terms of METASAN variables, this corresponds to $1 \,/\, E[T_b(S)]$, where (the path set) S is defined to be $\{< *, compute\ 1, * >\}$. Note that $E[T_b(S)]$ is an estimator defined in the experiment file in Figure 5. Estimates of this measure were then obtained using the Sanscript and experiment files presented together with the steady-state simulation solver. The initial marking of J and K was varied to correspond to each possible structure state. Figure 8 is the resulting METASAN output for the case $MARK(J) = 2$, $MARK(K) = 1$. The results for the mean time between completions of "compute1" for each run are as follows:

MARK(J)	MARK(K)	$E[T_b(S)]$	Half Width	$1/E[T_b(S)]$
2	3	0.710	.003	1.409
1	3	1.403	.004	0.713
2	2	0.850	.001	1.177
1	2	1.408	.004	0.710
1	1	1.701	.002	0.588
2	1	1.699	.001	0.589

"Half Width" refers to the half width of a 95% confidence interval constructed about the estimate. The interpretation of each

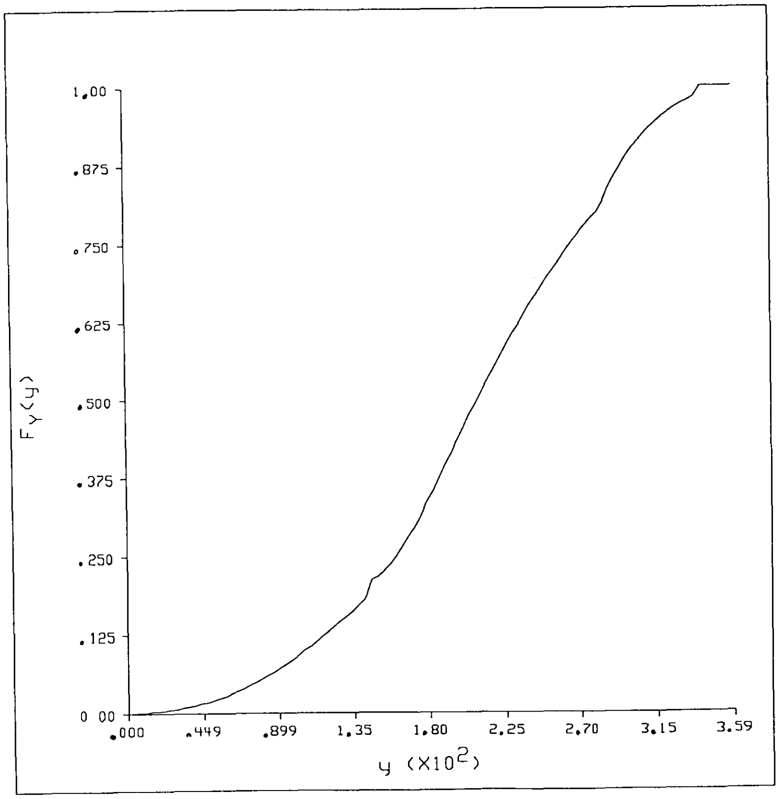


Figure 9. PDF of Number of Jobs Processed during [0,240]

entry in the last column is the rate at which blocks are processed in that structure state. These rates serve as the reward rates in the subsequent reward model solution. This model was then solved using the Markov reward model solver for the following model parameters: $\lambda = .001$, $\gamma = .005$, $c_t = .98$, $c_d = .99$, and performance submodel parameters as in Figure 4. This results in the probability distribution function (PDF) of the number of blocks processed during a specified utilization period [0,t]. Choosing t=240 produces the PDF in Figure 9.

6. Future Extensions

We have described METASAN, a comprehensive package that facilitates performability evaluation via both analysis and simulation. It permits detailed evaluations of systems such as complex computer networks, distributed computing systems, and flexible manufacturing systems. For example, in a recent evaluation study of a CSMA/CD network, a model containing over 500 places, gates, and activities was solved. Although the package has proved to be extremely useful in its current version, extensions would enhance this usefulness. Regarding model construction, a graphical input interface may help entry and debugging of large models. Regarding model solution, several extensions are possible. First, the available methods of confidence interval estimation for steady-state simulation should be extended to include the regenerative and spectral methods. During analytic model construction, an explosion in the number of states with increase in model size should also be dealt with. Methods of decomposition and state aggregation need to be studied and incorporated into future versions. Finally, work should be done to extend analytical solution methods to capture the activity related behavior of the networks.

ACKNOWLEDGEMENT

The authors would like to thank Steve Sparks and Paul Daugherty for their assistance during the design and implementation of this package. Their support has proved invaluable in making the tool a reality.

References

[1] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Trans. Comput.*, vol. C-22, pp. 720-731, Aug. 1980.

[2] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proc. 1984 Real-Time Systems Symp.*, Austin, TX, Dec. 1984.

[3] J.F. Meyer, A. Movaghar, and W.H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *International Workshop on Timed Petri Nets*, Torino, Italy, July 1-3, 1985, pp. 106-115.

[4] A. Movaghar, "Performability modeling with stochastic activity networks," CRL-TR-8-85, Computing Research Laboratory, University of Michigan, Ann Arbor, Sept. 1985.

[5] S. Natkin, "Reseaux de Petri Stochastiques," Thése de Docteur-Inge'nieur, CNAM-PARIS, June 1980.

[6] M. K. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Trans. Comput.*, vol. C-31, pp. 913-917, Sept. 1982.

[7] M. A. Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic Petri nets for performance evaluation of multiprocessor systems," *ACM Trans. on Computer Systems*, vol. 2, no. 2, pp. 93-122, May 1984.

[8] G. Chiola, "A software package of the analysis of generalized stochastic Petri net models," in *International Workshop on Timed Petri Nets*, Torino, Italy, July 1-3, 1985, pp. 136-143.

[9] Cumani, "ESP - A package of the evaluation of stochastic Petri nets with phase-type distributed transition times," in *International Workshop on Timed Petri Nets*, Torino, Italy, July 1-3, 1985, pp. 144-151.

[10] J. B. Dugan, K. S. Trivedi, R. M. Geist, and V. F. Nicola, "Extended stochastic Petri nets: Applications and analysis," in *Performance 84*, North-Holland, 1984, pp. 507-519.

[11] J. P. Behr, N. Dahmen, J. Muller, and H. Rodenbeck, "Graphical modeling with FORCASD," in *Computer Applications in Production and Engineering*, North-Holland Publishing Company, 1983, pp. 61-630.

[12] J. B. Dugan, "Extended stochastic Petri nets: Applications and analysis", Ph.D. Thesis, Department of Electrical Engineering, Duke University, 1984.

[13] H. P. Godbersen and B. E. Meyer, "A net simulation language," in *Proceedings of the Summer Computer Simulation Conference*, Seattle, WA, August 25-27, 1980.

[14] A. A. Torn, "Simulation nets, a simulation modeling and validation tool," *Simulation*, vol. 45, no. 2, pp. 71-75, Aug. 1985.

[15] W.H. Sanders, "Stochastic activity network execution and performance variable specification", Internal Report, Modeling Group, Communications and Network Laboratory, Industrial Technology Institute, Sept. 1985.

[16] S. S. Lavenberg, *Computer Performance Modeling Handbook.* New York, NY: Academic Press, 1983.

[17] J. F. Meyer, "Unified performance-reliability evaluation," in *Proc. of the American Control Conference*, San Diego, California, June 6-8, 1984.

[18] R. M. Keller, "Vector replacement systems: A formalism for modeling asynchronous systems," Computer Science Lab, no. 117, Princeton Univ. , Dec. 1972 .

[19] R. M. Keller, "Formal verification of parallel programs," *CACM*, vol. 19, pp. 371-384, July 1976.

[20] *IMSL library reference manual.* Houston, TX: IMSL, 1982.

[21] G.H. Golub, *Matrix Computations.* Baltimore, MD: Johns Hopkins University Press, 1983.

[22] R. A. Howard, *Dynamic Probabilistic Systems, Vol II: Semi-Markov and Decision Processes.* New York, NY: Wiley, 1971.

[22] R. A. Howard, *Dynamic Probabilistic Systems, Vol II: Semi-Markov and Decision Processes.* New York, NY: Wiley, 1971.

[23] J. F. Meyer, "Closed-form solutions of performability," *IEEE Trans. Comput.*, vol. C-31 , pp. 648-657, July 1982.

[24] D. G. Furchtgott and J. F. Meyer, "A performability solution method for degradable, nonrepairable systems," *IEEE Trans. Comput.*, vol. C-33, June 1984.

[25] L. Donatiello and B. R. Iyer, "Analysis of a composite performance reliability measure for fault tolerant systems," IBM Res. Report RC10325, January 1984.

[26] A. Goyal and A. N. Tantawi, "Evaluation of performability for degradable computer systems ," IBM Res. Report RC10529 (Revised), Dec. 1984.

[27] V. G. Kulkarni, V. F. Nicola, and K. S. Trivedi, "A unified model for performance and reliability of fault-tolerant/multi-mode systems," CS-1984-12, Dept. of Computer Science, Duke University.

[28] D. Gross and D. R. Miller, "The randomization technique as a modeling tool and solution procedure for transient Markov processes," *Operations Research*, vol. 32, no. 2, pp. 343-361, March-April 1984.

[29] D. R. Miller, "Reliability calculation using randomization for Markovian fault-tolerant computing systems," in *Proc. 1983 Int. Symp. Fault-Tolerant Computing*, Milano, Italy, June 1983, pp. 284-289.

[30] E. Cinlar, *Introduction to Stochastic Processes.* Englewood Cliffs, N.J.: Prentice-Hall, 1975.

A Hierarchical, Combinatorial-Markov Method of Solving Complex Reliability Models

Robin A. Sahner Kishor S. Trivedi

Gould CSD, Urbana
1101 E. University
Urbana, IL. 61801

Department of Computer Science
Duke University
Durham, NC. 27706

ABSTRACT

Combinatorial models such as fault-trees and reliability block diagrams are efficient in both specification and evaluation of system models. But it is difficult if not impossible to allow for various types of dependency (such as repair dependency and near-coincident-fault type dependency), transient and intermittent faults, standby systems with warm spares, and so forth. Markov models can capture such interesting system behavior. However, the size of a Markov model for the evaluation of such a system may grow exponentially with the number of components in the system.

This paper presents a modeling tool called SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator), which is a general hierarchical modeling tool for analyzing complex reliability models. SHARPE allows its user complete freedom to choose the number of levels of models and the type of model (combinatorial or Markov) at each level. Thus it allows the flexibility of Markov models where necessary and retains the efficiency of combinatorial solution where possible. The analysis of each model produces a probability function that is symbolic in the time variable *t*.

INTRODUCTION

Combinatorial models (fault trees and reliability block diagrams) and Markov or semi-Markov models are two of the approaches available for analytically predicting the reliability or availability of fault-tolerant systems. The advantages of using combinatorial models are the parsimony in specification and efficiency of evaluation they provide. If components are independent, reliability block diagrams with series-parallel ("well-nested") structure and fault trees without repeated nodes can be analyzed using linear time algorithms.

One method for analyzing structures with dependent components and non-series-parallel structures is the use of conditioning (the theorem of total probability) [16]. With this method, it is necessary to enumerate all of the sequences of events that lead to system failure. The number of such sequences can grow exponentially with the number of components, so this method is quite expensive.

Another alternative for analyzing these structures is to use Markov models. Again, the method is expensive; if there are n components in a block diagram (or n basic events in a fault tree), the corresponding Markov chain may have up to 2^n states. This severely limits the size of problems that can be handled. Since most practical problems do not satisfy the assumptions of independence and series-parallel structure, it is important to investigate ways of dealing with potentially very large state spaces.

There are two separate, but related, aspects to the large state space problem: constructing the Markov chain and analyzing it. The problem of construction can be alleviated by an automatic generation of the Markov chain from a more parsimonious description of system behavior. Examples of programs that do this are SAVE (System AVailability Estimator) [8] and DEEP (Duke Evaluator for Extended stochastic Petri nets) [5,6]. A system can be specified for DEEP by means of a stochastic Petri net, which is automatically transformed into a Markov chain that is solved numerically [6].

The problem of analyzing a Markov chain with a large state space can be handled by one of two methods: either tolerating the large state space or avoiding it. By "tolerating" the large state space, we mean that special solution methods are used that work for very large matrices. Examples are matrix-level decomposition [2] and various numerical methods for solving large, sparse, possibly stiff systems of algebraic equations [8] or ordinary differential equations [12].

One way of avoiding the large state space is by using a stochastic Petri net model and simulation to analyze the system [5,6]. Another way is to use model-level decomposition. This is the method used in HARP [7,17,18], where the reliability model is decomposed behaviorally along temporal lines. The fault-occurrence behavior (a slow submodel) and the fault/error-handling behavior (a fast submodel) are analyzed separately.

We have developed a hierarchical modeling technique that makes it possible to use mixtures of different kinds of models at different levels in order to avoid a state space explosion. Our technique differs from models such as HARP and CARE III [15] in several ways. HARP and CARE III assume a specific fixed hierarchy of models geared toward modeling a chosen class of systems. Our technique allows complete freedom in the number of levels in the hierarchy, which kinds of models to use at each level, and how to combine the models. The previous efforts have used numerical methods to obtain the reliability for a specified mission time. The results are therefore approximate (due to numerical integration), and have to be computed repeatedly for each value of the mission time. Our technique provides a result that is symbolic in the mission time variable.

This work was supported in part by the Air Force Office of Scientific Research under grant AFOSR-84-0132, by the Army Research Office under contract DAAG29-84-0045 and by the National Science Foundation under grant MCS-830200.

CH2345-7/86/0000/0817$01.00 © 1986 IEEE

In the next section, we provide some details of our approach to hierarchical modeling. In section 3, we show how to use our method to solve non-series-parallel block diagrams. In section 4, we discuss ways to model various kinds of dependencies and give several examples of system models that contain dependency.

THE SHARPE FRAMEWORK

We have developed a hybrid, hierarchical modeling framework that we call SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator). The framework was developed with automation in mind, and it has been implemented as a software tool, which we also call SHARPE. The SHARPE framework provides five model types:

- series-parallel reliability block diagrams
- fault trees without repeated nodes
- cyclic or acyclic Markov chains
- acyclic semi-Markov chains
- series-parallel directed (acyclic) graphs

Block diagram and fault tree models are specialized for modeling reliability and availability. Each component is assigned a cumulative distribution function (CDF) for the time-to-failure of the component. The system is analyzed to obtain the CDF of the time-to-failure of the system as a whole.

The other model types can be used to model performance as well as reliability. This paper focuses on reliability modeling; for examples of the use of SHARPE for performance modeling, see [14].

Markov and semi-Markov chains that have absorbing states are analyzed for the CDF of the time to absorption. If such a chain is acyclic, the analysis also produces the probability of ever visiting each state. Irreducible cyclic Markov chains are analyzed for the steady-state probabilities of being in each state.

The series-parallel graph submodel is the most general model. In this model, the nodes represent activities and the arcs represent precedence constraints placed on the activities. Each node in the graph is assigned a CDF, and the graph submodel is analyzed for the CDF of the time-to-completion of all required events in the graph. Once the CDF of the time-to-completion is obtained, it is easy to compute the mean and variance.

This model type is similar to series-parallel stochastic PERT (Program Evaluation Review Technique) networks [4] but is more general in several respects. (Note that PERT networks have activities on the edges whereas SHARPE associates activities with nodes.) We allow parallel graphs to be executed either concurrently or probabilistically. When a node has more than one predecessor and the predecessors are executing concurrently, most graph models (including PERT) require that the node wait for all of its n predecessors to finish before it begins execution. We include this case as a special case in our graph model; in general, the number of predecessors that must finish before the node starts is part of the graph model specification, and may be any number between (and including) 1 and n.

The different types of submodels can be combined hierarchically by using all or part of the solution to one submodel as part of the specification of another submodel. The solution for each model includes various scalar quantities. SHARPE makes available the mean and variance of each CDF produced by the analysis of a system, and the value of each CDF at specified values of t (including $t=0$ and $t=\infty$). SHARPE also makes available the probability of visiting a state in a Markov or semi-Markov chain. These scalars can be used in another model as elements in the expressions that specify probability values, transition rates, and the parameters of distribution functions. This mechanism allows for the expression of aggregation/approximation within the SHARPE framework. Examples of the use of this mechanism are given in section 4.

A second way to combine models hierarchically is to assign the solution CDF from a submodel as the CDF for a basic event in some other model. This method of combining submodels allows us to efficiently analyze large systems whose "badness" (non-series-parallel structure) is contained in a subsystem or a set of subsystems, with the remaining portions of the system being "well-behaved". We can extract the non-series-parallel portions of the overall structure and pay the price of 2^n states to analyze them exactly. Then we use the results from those portions as the CDFs of basic components in the remaining graph, and use a combinatorial solution method to analyze the system exactly. Section 3 presents an example of this method.

Because we allow the solution CDF from one submodel to be assigned as the CDF of a basic event in another submodel, the CDFs for individual events and the solution CDF for each model type must have the same form. If we are to work with symbolic expressions for the CDFs, we need a class of functions that is closed under the operations we need to perform during the analysis of the different kinds of models. The computational operations involved in the analysis of the models are convolution, order statistics [3] of independent (not necessarily identically distributed) random variables, and mixing of distributions (weighted averages). Therefore, we need a class of functions that is closed under addition, multiplication, differentiation and integration, and includes the exponential distribution.

The class of functions we have chosen is the class of exponential polynomials that are valid distribution functions. An exponential polynomial is a function with the following form:

$$\sum_{j=1}^{n} a_j t^{k_j} e^{b_j t} \qquad (1)$$

where k_j is a nonnegative integer and a_j and b_j are real or complex numbers.

Of course, a distribution function must be real-valued, even if it contains complex coefficients or complex powers of e. Allowing a distribution to contain complex numbers is equivalent to allowing it to contain sine and cosine functions. Such distributions arise naturally in the analysis of processes that have cyclic behavior. In particular, a cyclic Markov chain with absorbing states may have complex numbers in the CDF for its time-to-absorption.

We allow the distributions to have a mass at zero and/or a mass at infinity. A distribution has a mass at infinity if it does not reach 1 in the limit, and in that case, we call it a "defective" distribution. Defective distributions are useful for modeling the performance of programs that might fail to complete because of either software or hardware faults. It is also possible to use distributions that have mass only at zero and infinity. Assigning such a distribution to a component is equivalent to assigning to the component a probability value rather than a distribution function.

The SHARPE program consists of about 4400 lines of code in C [10] and about 260 lines of Fortran and runs under UNIX^{TM} [9] or VMS^{TM}. It is a full implementation of the modeling technique and has been tested extensively. The SHARPE program may be used either interactively or in batch mode. The specification format is the same in either mode. In interactive mode, SHARPE will prompt the user for all the information it needs and indicate the proper format for the information. Experience has shown that this mode is easier for beginning users. There is also an option that allows for interactive input to be stored in a file, thus allowing for a relatively painless way to obtain data files for large systems. The specification language is simple enough that many users will find it easy and convenient to prepare data files using a text editor.

ANALYZING A NON-SERIES-PARALLEL STRUCTURE

When a system has series-parallel structure, its failure-time CDF can be obtained (in linear time) by multiplying together the failure-time CDFs of parallel subsystems, and by subtracting from one the product of the reliability functions of series subsystems. This method works because the series-parallel nature of the system means that it can be broken down into substructures whose failure times are statistically independent of one another. When a system is not series-parallel, such a breakdown is not possible and so the standard block diagram method described above is not applicable.

Consider the system in Figure 1. The system is operative if there is a path from source to sink. We want to compute the CDF of the time to failure of the system. The system has two non-series-parallel subsystems having the same structure, often called a "bridge" structure. Such a structure can be analyzed using conditioning on the cross-over component (*8* in the upper bridge subsystem). Alternatively, if all of the components failure-time distributions are exponential, it is possible to expand each bridge system into a Markov chain.

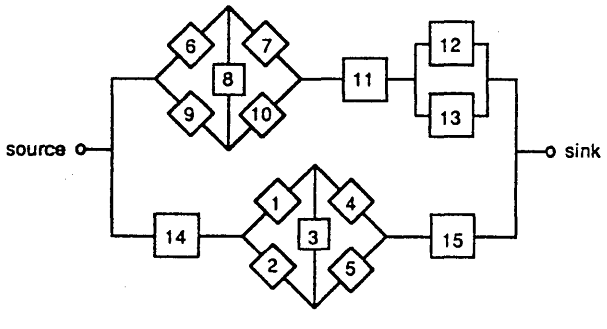


Figure 1. A Non-Series-Parallel Reliability Diagram

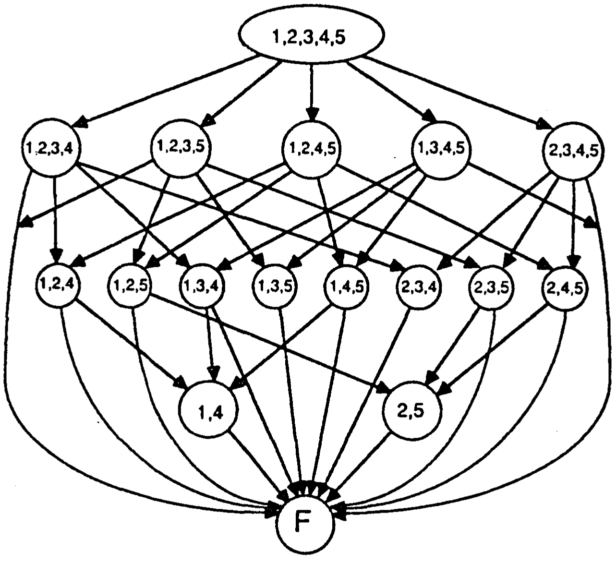


Figure 2. Markov Chain for the Bridge System

We have chosen the second approach, to illustrate the hierarchical nature of SHARPE and its ability to combine results of different model types. The 5-node bridge system expands to a Markov chain with 17 states, shown in Figure 2. Rates are not shown, for the sake of readability. The SHARPE framework allows us to use the time-to-absorption of each Markov chain as the failure-time CDF of a basic component in an upper-level reliability block diagram model. This decomposition and aggregation will give us exact results.

Figure 3 shows a batch input file for the two-level, hybrid system. For the sake of this exposition, the lines in Figure 3 are numbered; an actual SHARPE input file would not have its lines numbered. The SHARPE input language is described in detail in [13]. First, on lines 1 through 66, we define the lower level: a Markov chain called *bridge*, with five parameters. The chain is specified by giving each state transition and its associated rate, one transition per line. The rates are allowed to be any valid arithmetic expression. No initial state probabilities are given; the SHARPE program will assume that the initial state probability is one for the only state (*12345*) with no predecessors.

Next, as an illustration of the distribution specification mechanism of SHARPE, we show (lines 68 through 71) how to define the 2-stage Erlang distribution Erlang2(μ) $= 1-e^{-\mu t}-\mu te^{-\mu t}$. Then, on lines 73 through 85, we define a reliability block diagram for the upper level. We first define the basic components, assigning each one a CDF. Nodes 11, 12 and 14 are assigned the exponential distribution function, which is built in. Nodes 13 and 15 are assigned 2-stage Erlang distributions. Node B represents the upper bridge system of Figure 1 and node A represents the

^{TM} UNIX is a trademark of Bell Laboratories.

^{TM} VMS is a trademark of DEC.

```
1    markov bridge(u1,u2,u3,u4,u5)
2
3    12345  2345  u1
4    12345  1345  u2
5    12345  1245  u3
6    12345  1235  u4
7    12345  1234  u5
8
9    1234  234  u1
10   1234  134  u2
11   1234  124  u3
12   1234  F    u4
13
14   1235  235  u1
15   1235  135  u2
16   1235  125  u3
17   1235  F    u5
18
19   1245  245  u1
20   1245  145  u2
21   1245  125  u4
22   1245  124  u5
23
24   1345  F    u1
25   1345  145  u3
26   1345  135  u4
27   1345  134  u5
28
29   2345  F    u2
30   2345  245  u3
31   2345  235  u4
32   2345  234  u5
33
34   124  F   u1
35   124  14  u2
36   124  F   u4
37
38   125  25  u1
39   125  F   u2
40   125  F   u5
41
42   134  F   u1
43   134  14  u3
44   134  F   u4
45
46   135  F   u1+u3+u5
47
48   145  F   u1
49   145  F   u4
50   145  14  u5
51
52   234  F   u2+u3+u4
53
54   235  F   u2
55   235  25  u3
56   235  F   u5
57
58   245  F   u2
59   245  25  u4
60   245  F   u5
61
62   14  F  u1+u4
63   25  F  u2+u5
64
65   end
66   end
67
68   poly Erlang2 (u) gen\
69      1,0,0\
70      -1, 0, -u\
71      -u, 1, -u
72
73   block b
74   comp 11 exp (.0002)
75   comp 12 exp (.00002)
76   comp 14 exp (.00002)
77   comp 13 Erlang2 (.0002)
78   comp 15 Erlang2 (.00002)
79   comp A cdf(bridge;u1,u2,u3,u4,u5)
80   comp B cdf(bridge;u6,u7,u8,u9,u10)
81   parallel C 12 13
82   series D 11 B C
83   series E 14 A 15
84   parallel top D E
85   end
86
87   bind
88   u1 .0001
89   u2 .00001
90   u3 .0001
91   u4 .00001
92   u5 .0001
93   u6 .00001
94   u7 .0001
95   u8 .00001
96   u9 .0001
97   u10 .00001
98   end
99
100  expr value (10000;bridge;u1,u2,u3,u4,u5)
101  expr value (10000;bridge;u6,u7,u8,u9,u10)
102  expr value (10000;b)
103  eval (b) 0 100000 10000
104  end
```

Figure 3. Input for Non-Series-Parallel System

lower bridge system. Nodes A and B are each assigned the CDF of the time to absorption in the chain *bridge* with appropriate arguments. After the basic components have been defined, we build the block diagram using series and parallel combinations.

Before we can analyze the systems, we must bind the variables used to particular values. This is done on lines 87 through 98. Then we ask SHARPE for the probability that each non-series-parallel subsystem has failed by time t=10,000 (lines 100 and 101) and the probability that the overall system has failed by that time (line 102). Finally (line 103), we ask to have the CDF of the overall system evaluated between t=0 and t=100,000, at intervals of 10,000. Because SHARPE computes the system CDF symbolically, the only work it needs to do after the initial analysis is to evaluate the system CDF for the specified values of t. The results produced by SHARPE are shown in Figure 4. It took SHARPE 4.5 seconds on a lightly loaded DEC VAX 11/750 to produce these results.

```
value (10000;bridge;u1,u2,u3,u4,u5):    3.2426e-01
-------------------------------------------------
value (10000;bridge;u6,u7,u8,u9,u10):    1.4794e-01
-------------------------------------------------
value (10000;b):    4.0948e-01
-------------------------------------------------

            system b
      t          F(t)

    0.0000 e+00  0.0000 e+00
    1.0000 e+04  4.0948 e-01
    2.0000 e+04  8.1903 e-01
    3.0000 e+04  9.5264 e-01
    4.0000 e+04  9.8820 e-01
    5.0000 e+04  9.9713 e-01
    6.0000 e+04  9.9931 e-01
    7.0000 e+04  9.9984 e-01
    8.0000 e+04  9.9996 e-01
    9.0000 e+04  9.9999 e-01
    1.0000 e+05  1.0000 e+00
```

Figure 4. Results for Non-Series-Parallel System

MODELING DEPENDENCE

Fault trees and reliability block diagram models generally assume that the component lifetimes are stochastically independent. In real systems, there are several kinds of dependencies. Some of these are:

- **repair dependence**. Two or more components or subsystems may share a repair person.
- **state-dependent failure rates**. It is possible that the failure rate of a component may depend on the past history of the system. For example, it is possible that the repair of a component does not restore it to its original state. In that case, the component may have a larger failure rate after it has been repaired.
- **near-coincident fault dependence**. The design of a system may be such that near-coincident faults cause a system failure, while faults that are separated in time can be handled individually without overall system failure [7,17,18].

The common approach to modeling systems that possess any one of these kinds of dependence is to use a global Markov (or semi-Markov) model. Using the SHARPE framework, we can avoid a state space explosion by isolating the dependencies to portions (subsystems) of the overall system, using decomposition/aggregation, or by a combination of these two methods. The examples in the next three sections show how the SHARPE framework can be used in this way to model the kinds of dependence discussed above.

Repair Dependence

Consider a system consisting of 2 processors and 3 memories connected to a bus [8]. Let us assume that the mean time to failure of a processor is a month (720 hours), that of a memory unit is 2 months, and that of the bus is 3 months. The system as whole is considered to be up if the bus is up, at least one of the two processors is up, and at least 2 out of the three memory units are up. The condition under which the system is down can be expressed by the fault tree in Figure 5. If the system is non-repairable, we can easily evaluate the fault-tree directly by one of many tools available, including SHARPE. Using SHARPE, we can attach either a probability of failure or a distribution of time-to-failure to each basic event.

If the units are repairable and each individual unit has its own repair person, we can model the steady-state unavailability of the system using the same fault tree, by assigning to each component its steady-state unavailability. Using SHARPE, we can also use this fault tree to find the instantaneous unavailability of the system, by assigning the instantaneous unavailability expression as the "distribution" for each basic event. The instantaneous unavailability for a component, labelled i, that is initially operational and has failure rate λ_i and repair rate μ_i is [16, p. 301]

$$U_i(t) = \frac{\lambda_i}{\lambda_i + \mu_i} - \frac{\lambda_i}{\lambda_i + \mu_i} e^{-(\lambda_i + \mu_i)t} \qquad (2)$$

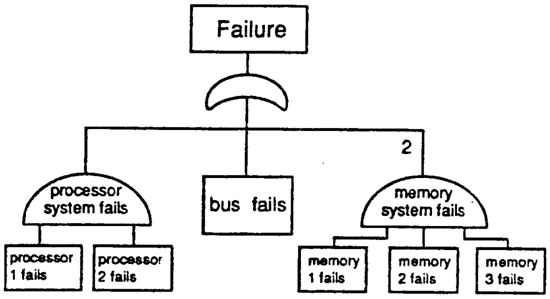


Figure 5. Fault Tree

For a system consisting of parallel components, the unavailability of the system is the product of the component unavailabilities (the system is unavailable only when all components are unavailable). This is the "and" gate computation in a fault tree. For a series of components, the availability is the product of the component availabilities (the system is available only when all subsystems are available). Thus the unavailability of the system is exactly the "or" combination of the components. When the components are combined in this way, the solution "distribution" for the system will be the system unavailability.

The limit of $U_i(t)$ as t approaches infinity is the steady-state unavailability. $U_i(t)$ does not reach one in the limit; it is possible to use this expression as if it were a distribution because the SHARPE framework permits defective distributions.

If we set $\mu_i = 0$ in Equation 2, we have the CDF for the time-to-failure of the component. Thus the same model can be used for both reliability and availability. Furthermore, we may allow some components to have a repair facility and other components to be non-repairable by letting μ_i be zero for some components and nonzero for others.

Now suppose that an independent repair facility is available for each subsystem (not for each unit). Because of the hierarchical capability of SHARPE, we can model each subsystem as an independent Markov chain and still model the overall system by a fault tree. This is shown in Figure 6. In the Markov chain for each system, the state gives the number of failed components in the system. In the fault tree, each component is assigned the steady-state unavailability of a subsystem. For the memory subsystem, the steady-state unavailability is the sum of the steady-state probabilities for states 2 and 3 in the corresponding Markov chain.

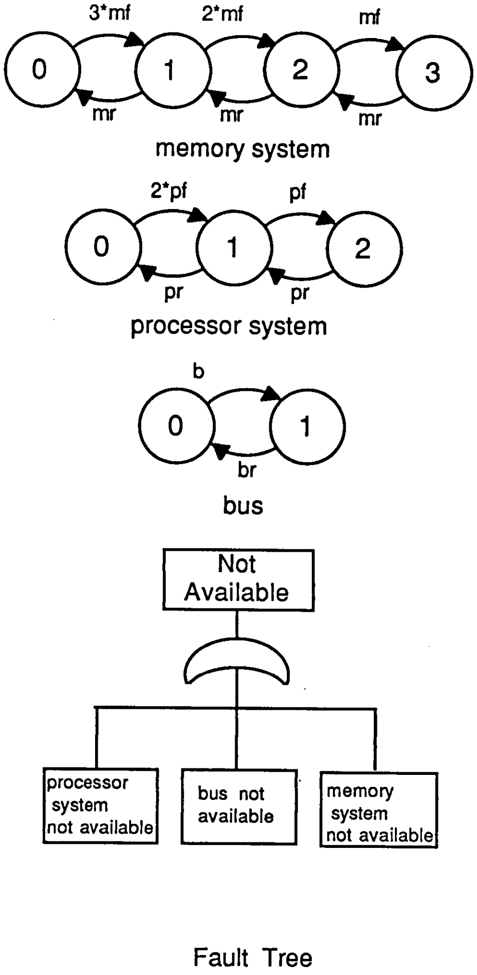


Figure 6. Three Markov Chains Within a Fault Tree

If repair is shared across all subsystems, we must model the entire system by a Markov chain. We assume that the repair priorities are bus first, then processor, then memory. Figure 7 shows the steady-state availability of the system when there is a single repair facility for the system, a repair facility for each subsystem, and a repair facility for each component. The figure shows that the effect of sharing repair persons is relatively minor.

State-Dependent Failure Rates

In this section, we consider a model for a system where the fault-occurrence behavior depends on the history of failures and repairs in the system. Suppose we have a repair strategy that involves two kinds of repair procedures. When a component fails, we first apply a quick, simple repair procedure. This procedure fixes the component so that it is as good as new with probability r, and with probability 1-r the procedure introduces another flaw that alters the failure behavior of the component, causing it to fail more quickly and with a different kind of failure. If the component exhibits the second kind of failure, we use a more complex repair procedure that either restores the system to its original state (with probability c) or fails to repair the system at all (with probability 1-c). A Markov model for the failure of a single component is shown in Figure 8. It is assumed that $\lambda_2 > \lambda_1$ and $\mu_2 < \mu_1$. SHARPE can analyze this cyclic chain for the CDF of the time to reach state F.

type of repair	steady state availability
per component	.998822
per subsystem	.998802
one for system	.998801

Figure 7. Effect of Sharing Repair Facilities

Now assume that we have a system that uses parallel redundancy with three components, using the repair strategy discussed above for each component, and that the system is operational as long as any one of the components is functioning. We can model this system as a reliability block diagram where each component is assigned the CDF of the time to absorption in the Markov chain of Figure 8.

It is possible to give another interpretation to the Markov chain of Figure 8. If μ_1 is very large, then state *fail1* can represent a transient failure, μ_1 the rate of a transient recovery procedure, and r the probability that the transient recovery procedure was successful. If the procedure was not successful, the fault has become permanent, and λ_2 is the repair rate of a procedure for repairing permanent faults, with c being the permanent-fault coverage.

Near-Coincident Fault Dependency

A system design may be such that faults that are separated in time can be handled individually without overall system failure, but near-coincident faults cause a system failure. If this is so, the system components are not independent. In this example, we show how near-coincident faults can be modeled, and also illustrate the flexibility of our hierarchical combination mechanism. We will start with a single-level model and add levels as we show how the model can be refined and how aggregation can be expressed in the SHARPE framework.

We consider problem 7 in appendix G of [1]. This problem models an aircraft flight control system. The system contains three inertial reference sensors (IRS) and three pitch rate sensors (PRS), that monitor the status of the aircraft. All of the sensors are connected to each of four computer systems (CS). The computer systems independently collect information from the sensors and process the information. The computers are connected to each other and to three secondary actuators (SA). At least two of each type of component must be operational in order for the overall system to function correctly. In this system, the computers are most susceptible to failure (the failure rate for a computer is an order of magnitude greater than the failure rates for any of the other components). That is why there are four computers and only three of each of the other component types.

A Single-Level Model: Fault Tree

If we assume that all components operate (and fail) independently, and that system reconfiguration after a failure is perfect, we can model the system using a fault tree, as shown in Figure 9. The SHARPE program calculates that the probability of failure during an interval of 10 hours is 1.02381 × 10^{-6}. This agrees with the result produced by the CARE III package.

A Two-Level Model: Markov Chain within Fault Tree

The fault tree model assumes that system reconfiguration upon failure is a perfect operation. Now assume that the probability of failure in the computer subsystem is high enough that imperfect system reconfiguration may cause the unreliability of the overall system to be unacceptable. Therefore, we would like to model that part of the system more closely to incorporate some of the details of the fault/error-handling procedure. We assume for now that a second (near-coincident) fault does not occur while a first fault is being resolved.

The handling of a single fault can be modeled by the Markov chain in Figure 10. This is a model of the sequence

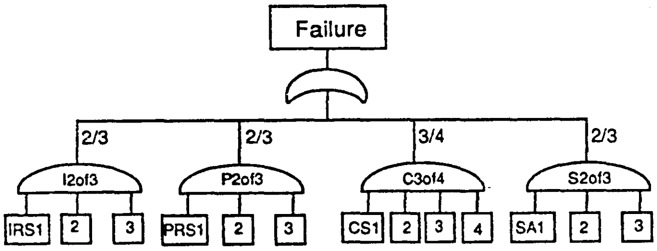


Figure 9. Aircraft Control System: Fault Tree

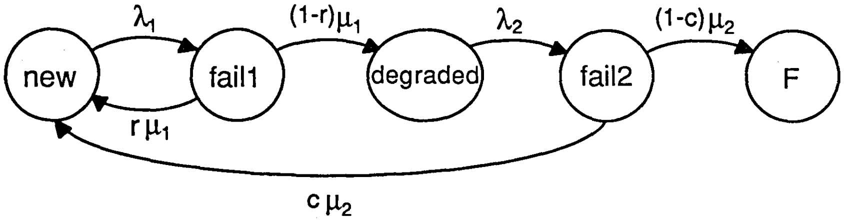


Figure 8. A Component with State-Dependent Failure Rates

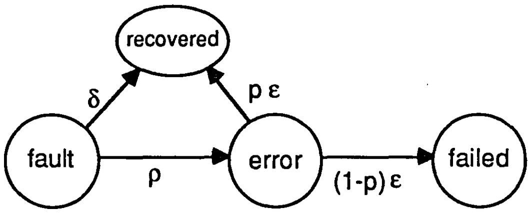


Figure 10. Fault/Error Handling Model

of events that follows the occurrence of a fault in a system that monitors itself periodically. A distinction is made between a "fault" and an "error" [11]. A fault may occur at any time, but it need not cause an error right away. For example, the failure of a pitch rate sensor is a fault, but an error does not occur until one of the computers tries to use results gathered by the sensor.

In state *fault*, a fault has occurred. At this point, one of two things might happen. The first is that the system may detect the fault itself through the self-monitoring process and recover from the fault. This happens with rate δ. The second way of leaving state *fault* is for an error (owing to the fault) to occur, causing the system to go to state *error*. The error rate is ρ. Once an error occurs, some time may elapse before it is detected. The error detection rate is ϵ. It is assumed that the system may not be able to recover from an error. If it can, then the error is "covered", and the system goes to state *recovered*; if not, the system fails and goes to state *failed*. A detected error is covered with probability p.

We now introduce a lower-level Markov model that incorporates the possibility of lack of coverage. The model, shown in Figure 11, is based on the single-fault/error-handling model of Figure 10. The states *4*, *3*, and *2* represent the system when *4*, *3*, and *2* of the computers are operational. Each time a fault occurs, we enter the fault/error-handling model and eventually leave it to enter either the failure state F or a state having one fewer operating computer.

To incorporate this lower-level model into the fault tree, we replace the gate *C3of4* and its four inputs by a basic event having the CDF of the time to reach node F in the Markov chain. (Figure 14 shows how all of the models discussed in the example are specified for the SHARPE program). When this two-level model is analyzed with $p = .999$, the probability of failure during an interval of 10 hours is computed to be 7.40614×10^{-6}. Again, this agrees with the result computed by the CARE III package. There is a noticeable increase in the likelihood of a failure, even though the error detection coverage value is .999.

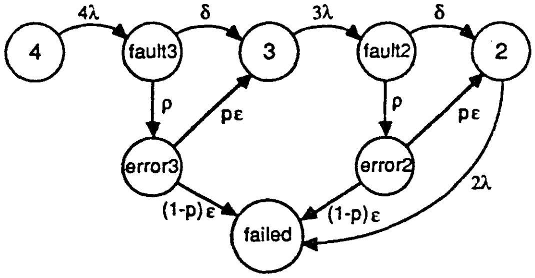


Figure 11. Markov Chain for Computer Subsysteנ

We note that in analyzing this model, CARE III obtains its solution numerically for the specified mission time. The SHARPE program computes the CDF for the system lifetime symbolically in t, and then evaluates that CDF for the specified value of t. Using SHARPE, we could find the probability of failure for other mission times without having to reanalyze the system.

A Three-Level Model: Markov Chain within Markov Chain within Fault Tree

Now suppose that the reliability requirements of the system are so stringent that the possibility of near-coincident faults cannot be ignored. Furthermore, suppose we assume that the occurrence of a second fault during the period between the occurrence of a fault and the completion of system reconfiguration causes the system to fail.

We could model this with a two-level system by refining the model of Figure 11 to include the possibility of the occurrence of a fault while in states *f3*, *f2*, *e3*, and *e2*. Instead, we would like to show how the SHARPE hierarchy of models can be used to express behavioral decomposition [7,17,18].

Behavioral decomposition is based on the premise that a long time passes between faults, but that once a fault occurs, the resolution of the fault happens quickly. Based on this assumption, we model separately the fault/error-handling behavior of the system. In Figure 12 (a) we show the fault/error-handling model, which is the lowest-level model. This is a Markov chain that models the sequence of events that follows the occurrence of a fault. The difference between this model and the model shown in Figure 10 is that here we incorporate an additional way for failure to occur; failure occurs if a second fault occurs while the chain is in state *fault* or state *error*. The rate at which the second

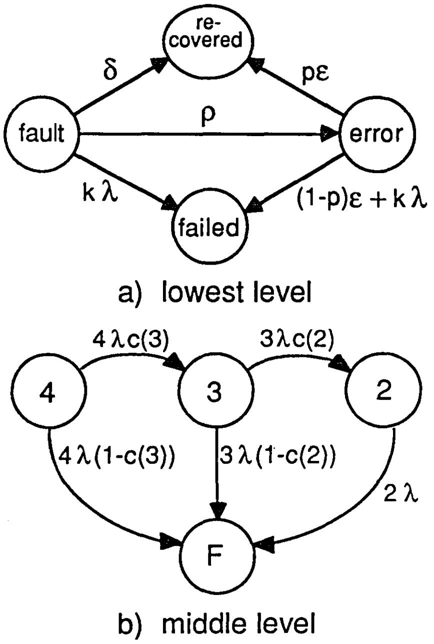


Figure 12. Markov Chains for 3-level Model

failure occurs is $k^* \lambda$, where k is the number of components that remain operational after the occurrence of the first fault.

Figure 12 (b) shows the middle-level model for this system. At this level, we have a Markov chain where state k represents the system having k operational computers. A state change occurs whenever there is a fault in one of the computers. >From state k, we go to state k-1 if there is a covered failure and the number of operational computers has not fallen below 2. We go to state F if there is an uncovered failure or the number of operational computers falls below 2. The coverage values (which are state-dependent) will be computed by solving the lower-level model of Figure 12 (a). The middle-level model is incorporated into the fault tree of Figure 9 by replacing the gate *C3of4* and its four inputs by a basic event having the CDF of the time to reach node F in the Markov chain of Figure 12 (b).

When SHARPE analyzes this three-level system, the results show that the probability of failure during an interval of 10 hours is 7.46150×10^{-6}. As expected, this is a little higher than in the previous two-level model. It is also slightly larger than the exact failure probability, which is computed by SHARPE to be 7.45961×10^{-6}.

We have noted that because the rate of occurrence of the second fault depends on k, the coverage values are state-dependent. This means that SHARPE will have to reanalyze the lowest-level (coverage) model of Figure 12 (a) for each value of k. With a model of this size, the expense of doing that is negligible, but it is interesting to see in the next section how we can extend the model one further level to avoid some of that extra computation.

A Four-Level Model: Markov Chain within Semi-Markov Chain within Markov Chain within Fault Tree

Let the bottom-most level of our model consist of the single-fault/error-handling model of Figure 10. At the next level up, we use the semi-Markov model shown in Figure 13. The initial state is *fault*. When a fault has occurred, two processes compete. The first process is the fault/error-handling process of Figure 10. If this process finishes first, we go to state *resolved*. The second process is the occurrence of a second fault, happening at a rate of $k^* \lambda$ (a state-dependent rate).

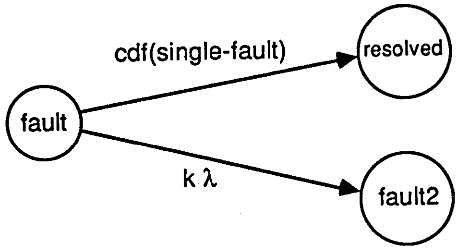


Figure 13. Semi-Markov Chain in 4-level Model

The fault is covered if the fault/error-handling process finishes first and if, within that process, the state *recovered* is reached rather than *failed*. Thus the coverage value is the probability of reaching state *resolved* in the semi-Markov chain of Figure 13, multiplied by the probability of reaching state *recovered* in the Markov chain of Figure 10. We still have to solve the model of Figure 13 for each value of k, but we only have to analyze the single-fault system one time. When we have SHARPE analyze this four-level system, the results are, as we expect, the same as for the three-level system.

In Figure 14, we show the complete SHARPE input file for all of the models we have discussed. They are defined in the input file in the same order as presented here.

CONCLUSION

We have developed a hierarchical modeling technique that provides a very flexible mechanism for using decomposition and aggregation to model large systems. The technique allows for both combinatorial and Markov or semi-Markov submodels, and can analyze each model to produce a distribution function. The choice of the number of levels of models and the model types at each level is left up to the modeler. Component distribution functions can be any exponential polynomial whose range is between zero and one.

Work is currently underway to improve and extend the SHARPE technique. Possibilities for future work include adding one or more queueing network models to the list of allowed model types, adding some kind of looping capability to the SHARPE input language, and allowing fault trees to have repeated nodes.

REFERENCES

[1] Bavuso, S., Peterson, P., and Rose, D., *Care III Model Overview and User's Guide,* NASA Technical Memorandum Number 85810, June 1984.

[2] Bobbio, A. and Trivedi, K.S., *An Aggregation Technique for the Transient Analysis of Stiff Markov Systems,* IEEE Transactions on Computers, 1986.

[3] David, H.E., *Order Statistics,* John Wiley & Sons, NY., 1981.

[4] Dodin, B., *Bounding the Project Completion Time Distribution in PERT Networks,* Operations Research, Vol. 33, No. 4 (July-August 1985), 862-881.

[5] Dugan, J.B., Bobbio, A., Ciardo, G., and Trivedi, K.S., *The Design of a Unified Package for the Solution of Stochastic Petri Net Models,* Proceedings International Workshop on Timed Petri Nets, Torino Italy, July, 1985.

[6] Dugan, J.B., Trivedi, K.S., Geist, R.M. and Nicola, V.F., *Extended Stochastic Petri Nets: Analysis and Applications,* PERFORMANCE '84, Paris, North-Holland (Dec 1984).

[7] Dugan, J.B., Geist, R., Trivedi, K.S., Smotherman, M., *The Hybrid Automated Reliability Predictor,* AIAA Journal on Guidance Control and Dynamics (1986).

[8] Goyal, A., Carter, W.C., de Souaz e Silva, E., Lavenberg, S.S., Trivedi, K.S., *The System AVailability Estimator,* IBM Computer Science Research Report (November 1985).

[9] Kernighan, B. and Pike, R., *The Unix Programming Environment,* Prentice-Hall (1984).

[10] Kernighan, B. and Ritchie, D., *The C Programming Language,* Prentice-Hall (1978).

```
* aircraft flight control system

bind
lambda    .00048
mIRS      .000015
mPRS      .000019
mSA       .000037
p         .999
delta     360
rho       180
epsilon   3600
end

precision 5

ftree  Non_Computers
basic  IRS    exp   (mIRS)
kofn   I2of3 2,3, IRS
basic  PRS    exp   (mPRS)
kofn   P2of3 2,3, PRS
basic  SA     exp   (mSA)
kofn   S2of3 2,3, SA
or     TOP    I2of3  P2of3  S2of3
end

ftree  CRASH
basic  CS      exp (lambda)
kofn   C3of4  3,4, CS
basic  others cdf (Non_Computers)
or     TOP     C3of4    others
end

expr value (10;CRASH)

* add  a  lower  level:
* exact  Markov  model  for
* single-fault/error  handling

markov  2
4   f3  4*lambda
f3  3   delta
3   f2  3*lambda
f2  2   delta
2   F   2*lambda
f3  e3  rho
f2  e2  rho
e3  3   p*epsilon
e2  2   p*epsilon
e3  F   (1-p)*epsilon
e2  F   (1-p)*epsilon
end
end

ftree 2-level
basic   CS      cdf  (2)
basic   others  cdf  (Non_Computers)
or      TOP     CS     others
end

expr  value (10;2-level)
```

```
* add  a  third  level,  with
* aggregation.
* third-level  model:
* fault/error  handling  model
* with  near-coincident  faults

markov   second-fault(i)
active  recovered  delta
active  error      rho
error   recovered  p*epsilon
error   F  (1-p)*epsilon + i*lambda
active  F  i*lambda
end
end

* approximation - second-level model
* with coverage  value  taken  from
* third-level  model

func c(i) \
    prob(second-fault,recovered;i)

markov 3
4 3 4*lambda*c(3)
4 F 4*lambda*(1-c(3))
3 2 3*lambda*c(2)
3 F 3*lambda*(1-c(2))
2 F 2*lambda
end
end

ftree 3-level
basic   CS      cdf (3)
basic   others  cdf (Non_Computers)
or      TOP     CS    others
end

expr  value (10;3-level)

* now  for  a  four-level  model
* lowest  level  is  the  single
* fault/error  handling  model

markov  single-fault
fault   recovered  delta
fault   error      rho
error   recovered  p*epsilon
error   failed     (1-p)*epsilon
end
end

* next level up: semimarkov
* chain showing competition of
* single-fault model with
* occurrence of a second fault

semimark  2nd-fault(i)
fault   second  exp (i*lambda)
fault   first   cdf(single-fault)
end
end
```

```
* second level: markov chain
* with coverage values

func  C(i) \
     prob(2nd-fault,first;i) *\
     prob(single-fault,recovered)

markov 4
4 3 4*lambda*C(3)
3 2 3*lambda*C(2)
4 F 4*lambda*(1-C(3))
3 F 3*lambda*(1-C(2))
2 F 2*lambda
end
end

* top level

ftree  4-level
basic   CS      cdf  (4)
basic   others  cdf  (Non_Computers)
or      TOP     CS     others
end

expr  value (10;4-level)

* exact two-level model
* with second fault

markov   2fault-exact
4        active3  4*lambda
active3  3        delta
3        active2  3*lambda
active2  2        delta
2        F        2*lambda
active3  error3   rho
active2  error2   rho
error3   3        p*epsilon
error2   2        p*epsilon
error3   F  (1-p)*epsilon + 3*lambda
error2   F  (1-p)*epsilon + 2*lambda
active3  F  3*lambda
active2  F  2*lambda
end
end

* top level

ftree  3-exact
basic   CS      cdf  (2fault-exact)
basic   others  cdf  (Non_Computers)
or      TOP     CS     others
end

expr  value (10;3-exact)
end
```

Figure 14. Input for Multi-level Aircraft Example

[11] Laprie, J., *Dependable Computing and Fault-tolerance: Concepts and Terminology,* Proposal to the IFIP WG 10.4 Summer 1984 meeting, Kissimmee, Florida, June 16-19, 1984.

[12] Reibman, A., and Trivedi, K.S., *Transient Analysis of Markov Dependability Models,* in preparation.

[13] Sahner, R. *SHARPE User's Guide,* Duke University, Durham, NC, 27706 (Dec. 1985).

[14] Sahner, R. and Trivedi, K.S., *Performance and Reliability Analysis Using Directed Acyclic Graphs,* accepted subject to revision, IEEE TSE.

[15] Stiffler, J.J., and Bryant, L.A., *CARE III Phase III Report - Mathematical Description,* NASA Contractor Report 3566 (Nov 1982).

[16] Trivedi, K.S., *Probability and Statistics with Reliability, Queuing and Computer Science Applications,* Prentice-Hall, Englewood Cliffs, N.J., 1982.

[17] Trivedi, K.S., Dugan, J. B., Geist R., and Smotherman, M., *Modeling Imperfect Coverage in Fault-Tolerant Systems,* Proc. of the Fourteenth Int. Conf. on Fault-Tolerant Computing (FTCS - 14), Orlando, FL., June 1984, 77-82.

[18] Trivedi, K.S., Geist R., Smotherman, M., and Dugan, J. B., *Hybrid Modeling of Fault-Tolerant Systems,* Computers and Electrical Engineering, An International Journal, Vol. 11, No. 2 and 3 (1985), 87-108.

DESIGN OF SYSTEMS WITH CONCURRENT ERROR DETECTION USING SOFTWARE REDUNDANCY

Kien A. Hua and *Jacob A. Abraham*

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

ABSTRACT

This paper describes a methodology for the design of systems with concurrent error detection through the use of software redundancy. We present a technique for the development of self-checking programs in which assertions are generated systematically from the design of the program. Software that is developed using this technique is guaranteed to be self-checking with respect to all software faults and can be shown in practice to detect most errors due to hardware faults. A less rigorous technique using data encoding is also described. The self-checking capabilities are evaluated using a fault simulation method that allows us to study the coverage of errors due to hardware and software faults. We also propose a dual processor technique that can improve the performance, reliability and availability of the self-checking system.

1. INTRODUCTION

Off-line circuit testing is one of the widely used techniques to detect physical failures and to ensure that a system is defect-free. Unfortunately, the decrease in geometries has increased the possibility of transient errors. Since these errors are usually nonrecurring and not reproducible, off-line testing (useful for permanent faults) will not reliably detect transients. Consequently, it is important that we incorporate concurrent error detection (CED) capability into the design of digital systems in order to detect errors concurrently with normal operations.

Depending on the level in a system at which we apply the checking mechanisms, concurrent error detection can be subdivided into three categories (Figure 1):

- gate-level checking
- functional level checking
- system level checking.

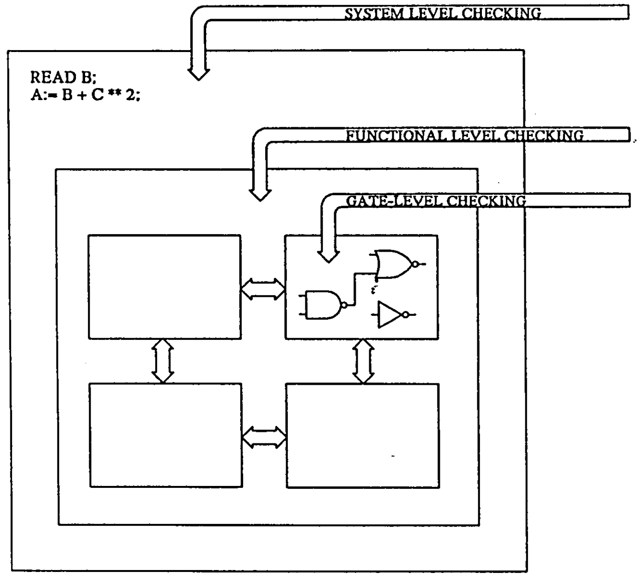


Figure 1. Three categories of concurrent error detection.

Gate-level techniques such as those using error detecting coding [1] usually assume the single or double stuck-at fault model. As the geometric features of integrated circuits become smaller, physical defects can affect a local area of a circuit, and stuck-at fault models are therefore not satisfactory. Functional-level techniques such as those used in algorithm-based fault tolerance [2, 3, 4] assume a more general model which allows any single module in the system to be faulty. These techniques can hence detect errors due to a block of faulty logic that is local to any single module. The third category of concurrent error detection assumes an even more general fault model that includes both design faults in software and physical failures in hardware. These techniques introduce software redundancy in the form of executable assertions into the program to check for the correct operation of the system during its execution. Since the checking mechanism is placed at programming level, both software and hardware faults that affect the dynamic behavior of the program can be detected. Such computer programs that check their own dynamic behavior automatically during their execution are called *self-checking programs* (SCPs).

In recent years, there have been several reports on self-checking programming. Depending on the aspects of the process which the assertions are checking, SCPs can be classified into three major categories:

(a) **Control flow checking** [5, 6, 7]: These techniques assume that malfunctions, such as processor failures, bus faults, or faults in the code being executed will cause errors in the control flow of the program. These techniques insert redundant code into the program so that the actual path traversed at run time can be recorded and checked against the precomputed path information.

(b) **Data structure checking** [8, 9]: These techniques use redundancy in data structures, such as additional pointers, identifier fields, or counters to detect any failures that change the data structure into a structurally incorrect one. The central idea in these

Acknowledgment: This research was supported in part by a contract from IBM Corporation and in part by the Semiconductor Research Corporation under Contract SRC RSCH 84-06-049-4.

techniques is that if a data structure is designed such that at least (n+1) changes are needed to update the data structure, then any faults that cause n or fewer changes in the data structure can be detected.

(c) **Data value checking** [10, 11, 12, 13]: These techniques assume that hardware or software faults will eventually corrupt the data and produce wrong results. They detect such faults by inserting into the programs assertions about the validity of the data values.

Control flow checking is a powerful technique for detecting hardware faults that changes the control sequence of the programs. However, it does not perform well in case of data manipulation errors. For instance, control flow checking will not be able to detect the use of an incorrect algorithm. Data structure checking is suitable for the implementation of reliable data base systems [14]. For general programs, data value checking is the more powerful technique. Since the assertions used in this technique are based on the semantics of the program, control sequence faults or data manipulation errors will change the intended semantics of the program and therefore will be detected by the assertions at run time. Unfortunately, there has not been a systematic technique for generating the assertions for data value checking. A structured technique for the design of SCPs was presented in [13]. This technique derives the assertions from the functional specifications of the program by performing stepwise refinement. Nonetheless, the process of stepwise refinement relies on intuition and is therefore still very much ad-hoc. In this paper, we will present in Section 2 a methodology for the design of SCPs, in which the assertions for data value checking are systematically derived from the design of the program. Software developed using this methodology is guaranteed to be self checking with respect to all software faults.

For some programming problems, checking the correctness of the outputs requires an additional copy of the input. This could be very inefficient if the number of inputs is large. A less rigorous self-checking programming technique for solving this class of problems will be discussed in Section 3.

Although the distinguishing feature of SCPs is the ability to detect both software and hardware faults, hardware fault coverage of SCPs has not been thoroughly studied hitherto. We propose in Section 4 a fault simulation technique that can be used to evaluate the affectiveness of SCPs against both hardware and software faults. This technique was used to study the coverage of errors, due to faults, of some popular programs. The empirical results of the experiments will be discussed

One weakness of SCPs is that the addition of software redundancy will degrade system performance. We propose a dual redundancy architecture to be used for SCPs. The proposed technique can improve both performance and reliability. A comparison of this technique with the conventional dual redundancy approach for CED will be discussed in Section 5.

2. DESIGN OF SELF-CHECKING PROGRAMS

Depending on their scope, assertions can be global or local. A single assertion for a program is an example of a global assertion. Global assertions have the advantage of being checked only once for each execution of the program. Unfortunately for some programs, their global assertions are as complex as the code of the programs. Using global assertions for this class of programs is therefore very inefficient. For such programs, one should replace the global assertion by a set of appropriate assertions inserted at appropriate locations throughout the program.

In this section, we present a methodology for the design of SCPs. First, we discuss the generation of global and local assertions. We then describe a procedure for the design of large scale SCPs. Since assertions are simple, they can be easily verified for correctness. In this paper, we assume that the executable assertions are free of software bugs.

2.1 Global Assertion Techniques

Global assertions are derived from the input/output specifications of the program. Since global assertions check the correctness of the final outputs, SCPs that use global assertions are always self-checking. The following are some examples of global assertions:

Problem:	Find the roots of a polynomial
Inputs:	A polynomial
Outputs:	Roots
Assertion:	The roots must satisfy the polynomial

Problem:	Find the solution of N simultaneous linear equations
Inputs:	A system of linear equations
Outputs:	Solution of the equations
Assertion:	The solution must satisfy all the equations

Problem:	Searching
Inputs:	A key and a list of items
Outputs:	Matched item
Assertion:	The search field of the matched item must match the key

Although global assertions can be easily derived from the functional specifications for many programs, some of the global assertions are very inefficient in terms of memory requirements, time requirements or complexity. This is discussed in [13]. As seen in the preceding examples, programs that are candidates for the global assertion technique are those which do not modify their inputs. Other good candidates are programs which have only a small set of inputs.

2.2 Local Assertion Technique

If the detailed design of a program is available, local assertions can be systematically generated from the semantics of the program design by symbolically "executing" [15] the design in the virtual machine environment (machine that supports the program design language). Since these assertions are derived directly from the design text, they reflect what is actually happening during the execution. These assertions therefore can be used to prove the correctness of the design with respect to the program specifications.

Once the correctness has been verified, the design can be transformed into executable form by translating the statements in the design into statements written in the target language. Even though this translation process may introduce coding errors, they will be detected during run time because the design has already been proved to be correct and the assertions are derived from the design, not the program code. Programs developed using this methodology are therefore guaranteed to be self-checking with respect to software faults: free of design bugs, while coding errors are guaranteed to be detected during the execution.

Conventional software verification techniques are not commonly used in practice due to the large size of software systems. The design of a program, however, is much smaller. The process of generating assertions and proving correctness is therefore much simpler in the proposed technique than in conventional program verification methods. Nevertheless many mechanical techniques for assertion generation intended for conventional program verification [16, 17, 18, 19] can be used in the proposed method as well.

We have given an overview of the assertion generation technique. The proposed technique, however, is best explained by examples. For the remainder of this subsection, we will illustrate the application of the technique to two popular programs:

- sorting (non-numeric)
- finding the Eigenvalues and Eigenvectors of a real symmetric matrix (numeric)

Example 1: This example illustrates our technique that systematically generates assertions from the design of a sorting program. For clarity, we choose a simple single-loop sorting program for the demonstration. The resulting assertions turn out to also be the global assertions for the program. In general, programs are more complex and we usually have multiple sets of assertions located throughout a program.

```
Procedure SORT(A,n)
    k:= 1
    Dowhile k<n
        Find $A_t = min(A_{k+1}, . . . , A_n)$,
              where k+1 $\leqslant$ t $\leqslant$ n
        If $A_k > A_t$ Then exchange($A_k, A_t$)
        increment k
    enddo
end;  (* SORT *)
```

The flow chart for the design is shown in Figure 2. The loop has two paths. Let us call the path with path condition $A_k > A_t$ the right path, and the path with path condition $A_k \leqslant A_t$ the left path. Also we denote by y/m/ the value of y the (m+1)-st time the cutpoint N (Figure 2) is reached since the most recent entrance to the block. We then have the following:

left path:

$$[k/m\text{-}1/ < n \wedge A_{k/m-1/}/m\text{-}1/ \leqslant A_t/m/] \supset [A_t/m/ = \min(A_{k/m-1/+1}/m-1/, \dots, A_n/m-1/) \wedge k/m/ = k/m\text{-}1/ + 1] \quad (1)$$

right path:

$$[k/m\text{-}1/ < n \wedge A_{k/m-1/}/m\text{-}1/ > A_t/m/] \supset [A_t/m/ = \min(A_{k/m-1/+1}/m-1/, \dots, A_n/m-1/) \wedge (A_{k/m-1/}/m/, A_t/m/) \leftarrow (A_t/m/, A_{k/m-1/}/m-1/) \wedge k/m/ = k/m\text{-}1/ + 1] \quad (2)$$

Since the assignments to k and A_t are not affected by which path is used, the following equations are always true at cutpoint N:

$$k/m/ = k/m\text{-}1/ + 1 \quad (3)$$

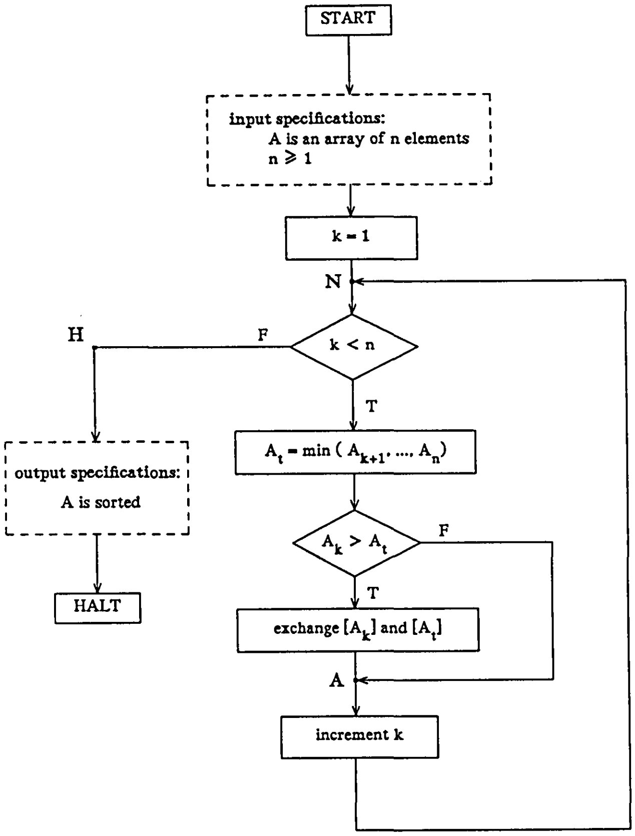


Figure 2. Flowchart for the sort program.

$$A_t/m/ = min\,(A_{k/m-1/+1}/m-1/, \dots, A_n/m-1/) \quad (4)$$

$$(3) \rightarrow k/m/ = k/0/ + m = m + 1 \quad (5)$$

$$(4)\ \&\ (5) \rightarrow A_t/m/ = min\,(A_{m+1}/m-1/, \dots, A_n/m-1/) \quad (6)$$

Ignoring the exit test k<n, and the assignments to k and A_t, combining (1) and (2) gives the following:

If $A_m/m-1/ > A_t/m/$
  then $(A_m/m/, A_t/m/) \leftarrow (A_t/m/, A_m/m-1/)$ (7)

From (6) and (7) we have the following:

m=1:
$A_t/1/ = min\,(A_2/0/, \dots, A_n/0/)$
If $A_1/0/ > A_t/1/$
  then $(A_1/1/, A_t/1/) \leftarrow (A_t/1/, A_1/0/)$
m=2:
$A_t/2/ = min\,(A_3/1/, \dots, A_n/1/)$
If $A_2/1/ > A_t/2/$
  then $(A_2/2/, A_t/2/) \leftarrow (A_t/2/, A_2/1/)$

.
.
.

m=n-1:
$A_t/n-1/ = min\,(A_n/n-2/, \dots, A_n/n-2/)$
If $A_{n-1}/n-2/ > A_t/n-1/$
  then $(A_{n-1}/n-1/, A_t/n-1/) \leftarrow (A_t/n-1/, A_{n-1}/n-2/)$

By induction, we therefore have the following loop invariants:

A1: $(A_1, ..., A_{k-1})$ is monotone
A2: $A_{k-1} \leqslant \min(A_k, ..., A_n)$
A3: A/k-1/ = PERMUTATION(A/0/)

The assertions at H therefore are:

A4: A/n-1/ is monotone
A5: A/n-1/ = PERMUTATION(A/0/)

Since A4 and A5 imply the output specifications, the design of the program is correct with respect to the specifications.

We can also prove the termination based on the classic Floyd's well-found set technique [20, 21]. We choose point A as the cutpoint which cuts both paths around the loop (Figure 2). Let us take the set N of all natural numbers, with regular < ordering, as the well-found set. Since $u_A = n-k$ is strictly monotonic, and are bounded (i.e., $u_A \geqslant 0$), the resultant sequence is therefore well founded. Thus clearly no loop or combination of loops could be executed indefinitely because the no-infinitely-descending-chain condition would be violated.

We see that the sets $\{A_1, A_2, A_3\}$ and $\{A_4, A_5\}$ are equivalent. Thus all five assertions are not required for self-checking. Since the set $\{A_4, A_5\}$ is more time-efficient, we choose it for the program. We might also want to add another assertion, A0: n>0, to check whether the input meets the input specifications.

Although we have proved the termination of the program design, the self-checking program will fail if the program code causes the loop to execute indefinitely. To overcome this problem, we can have a convention that the program designers reserve the right to design the control structures and their termination conditions, and programmers may not alter these constructs. If the designers always prove the termination of the designs and if assertions are added to dynamically check the well founded properties, then this convention guarantees the termination of the final product. There are two kinds of loops: DO loop and WHILE loop. For a DO loop, if the designer proved the termination properties of the loop based on the loop index, then clearly the DO loop must eventually terminate if the programmers do not modify the loop construct and the loop index designed by the designer in any way. In the case of a WHILE loop, the function for proving the termination of the loop (i.e., the function that maps elements of the program's data domain into a well-found set, e.g., u_A in Example 1) can be used as an executable assertion to dynamically check the termination condition. Since this function is monotone and is bounded, if the programmers do not modify this termination-checking function, then the WHILE loop must eventually terminate if the assertion is true for each iteration of the loop.

The design of the sorting program with assertions A0, A1 and A4 added is given in the following. Programmers now can take over and transform it into the target language.

```
Procedure SORT_SCP(A,n)
   If not A0 then ERROR
   k:= 1
   Dowhile  k<n
      Find $A_t = min(A_{i+1}, . . . , A_n)$,
            where i+1 $\leqslant$ t $\leqslant$ n
      If $A_k > A_t$ Then exchange($A_i, A_t$)
      increment k
   enddo
      If not A4 then ERROR
      If not A5 then ERROR
   end;  (* SORT *)
```

Example 2: The following is the design of a SCP that finds the Eigenvalues and Eigenvectors of a real symmetric matrix. The program is based on the Jacobi method. The fundamental approach of this technique is to annihilate, in turn, selected off-diagonal elements of the symmetric matrix by "elementary" orthogonal transformations. For more details on this method, interested readers are referred to [22].

```
Procedure EIGEN_SCP(A,S,N);
   AI= A;  (* Save the input *)
   Generate an identity matrix S;
   Compute initial norm $\nu_I = \sqrt{\sum A_{ij}^2}$;
   Initialize current norm $\nu = \nu_I$;
   Compute final norm $\nu_F = (\epsilon/N)\nu_I$;
   (* $\epsilon$ represents zero off diagonal *)
   Repeat
      $\nu = \nu/N$;
      For Q:= 2 to N Do begin
         P:= 1;
         Repeat
            OFFNOTZERO:= false;
            If A[P,Q] > $\nu$ then begin
               OFFNOTZERO:= true;
               Compute $Sin\,\theta$ and $Cos\,\theta$;
               Compute the transformation matrix R;
               Perform the transformations:
                  A= $R^T \cdot A \cdot R$  and
                  S= $S \cdot R$;
            end;  (* if *)
            P= P + 1;  (* go on to next row *)
         until P > (Q-1);
      end;  (* for Q *)
   until ($\nu \leqslant \nu_F$) and (not OFFNOTZERO);
   Assertion:  If not (A = $S \cdot AI \cdot S^T$) then ERROR;
   Eigenvalues= diagonal elements $A_{ii}$'s;
   Eigenvectors= columns of S;
end;  (* EIGEN_SCP *)
```

As we did in Example 1, we can derive the following loop invariants by symbolic execution:

$$A/m/ = R/m/^T \cdot A/m-1/ \cdot R/m/$$
$$S/m/ = S/m-1/ \cdot R/m/$$

Upon the completion of the repeat loops, we therefore have the following assertion:

$$A/f/ = S/f/^T \cdot A/i/ \cdot S/f/ \qquad (8)$$

where A/f/ is the final matrix A
A/i/ is the initial matrix A
S/f/ is the final matrix S.

Equating the elements of the matrices in (8), we obtain

$$\sum_{k=1}^{N} A/i/_{mk} \cdot S/f/_{ki} = \sum_{k=1}^{N} S/f/_{mk} \cdot A/f/_{ki}$$

Let us denote by $X^{(i)}$ the ith column of a matrix X, we then have:

$$A/i/\cdot S/f/^{(i)} = S/f/\cdot A/f/^{(i)}$$

Since the design of the control structure ensures that A/f/ is a diagonal matrix , the above equation can be written as:

$$A/i/\cdot S/f/^{(i)} = A/f/_{ii}\cdot S/f/^{(i)}$$

The diagonal elements of the matrix A/f/ therefore must be the Eigenvalues, and the columns of the matrix S/f/ then must be the corresponding Eigenvectors. Since the assertion (equation (8)) generated implies the output specification of the program, it guarantees the self-checking properties of the program.

2.3 Design of Large Self-Checking Programs

In this subsection, we describe the procedure for the design of large SCPs. This is a three-phase process. Given a programming problem, we first do the system design using some hierarchical modular design technique: top-down, bottom-up or more often a mixture of the two. This process converts the functional requirements of the program into a hierarchy of program modules. For each module, we then do the detailed design using a procedural design language. The design language used should be precise and contain enough detail so that symbolic execution of the design is possible. In Example 1, we used a program design language similar to that presented in [20].

To make the program self-checking, we then apply the global assertion or local assertion techniques presented in Section 2.1 and Section 2.2 respectively to each module. During this process, a procedure call is considered as an abstract executable instruction of a virtual machine implemented by the program modules in the lower levels of the hierarchy. Since the called modules (procedures) are themselves self-checking, the calling modules can assume that the results returned from the called modules are correct.

Since the assertions for each module are derived independently, the set of assertions generated thus far could be excessive. A final inspection is therefore needed to remove the redundant assertions.

Definition 1: An assertion AI is said to *dominate* another assertion AJ if all the properties checked by AJ are also checked by AI.

Definition 2: If an assertion AI dominates another assertion AJ, then AI is called the *dominating assertion* and AJ is called the *dominated assertion*.

In a SCP, some of the assertions may be redundant and should be removed for better system performance. As an example, if a module that computes solutions of linear equations calls a different module to do the Gaussian elimination and it computes the solutions by back substitutions, then the assertion, in the calling module, that checks whether the solution satisfies all the equations dominates all the assertions in the called modules. The dominated assertions therefore can be removed without impairing the self-checking properties of the software system. Usually, we prefer to save the dominated assertions since they can help to identify faulty modules, and system malfunctions can be detected early. Nevertheless, in some cases the dominating assertions are preferred for the sake of better system performance.

We summarize the three phases of the proposed self-checking programming methodology in the following:

PHASE 1: Design
- Use hierarchical modular design techniques to design a hierarchy of program modules.
- Do the detailed design for each module in some computable program design language.

PHASE 2: Make assertions
- Generate assertions for each module using either global or local technique as appropriate.

PHASE 3: Remove redundant assertions
- Identify dominating and dominated assertions
- Remove the redundant ones.

Although we described the methodology as three separate processes, in practice PHASE 2 and PHASE 3 can be overlapped to reduce design time.

The proposed methodology is compatible with the traditional software development process in the sense that the process of making the program self-checking by adding assertions is a totally separate phase in the software development process. The job of making the program self-checking can be done by a specialist, and the program designers and the programmers need not change their traditional practicing styles. Nevertheless, the assertions can serve as a communication tool between the designers and the programmers. They provide additional information to guide the programmers in programming. They are also very useful for the programmers to informally verify their code quickly.

3. ADDING ADDITIONAL PROPERTIES TO SCPs

In Section 2, all the assertions generated imply the output specifications of the programs. If the assertions are true, the outputs computed must be correct. This class of assertions therefore guarantee the self-checking properties of the programs. In this section, we will present a less rigorous technique in which the assertions do not imply the output specifications. Nevertheless our experiments (to be described) show that this technique can also be very effective.

Since some of the programming problems have their inputs and outputs share the same memory space, the outputs therefore "destroy" the input information. In order to check the correctness of the outputs we must save an extra copy of the inputs so that relationships between the inputs and outputs can be validated. Sorting is one such example. Since we need to check whether the sorted list is a permutation of the original list, the input list must be available for the validation. For this class of problems, if the size of the input data is large, then the assertions based on the input/output relationships are not very efficient.

Alternatively, we may add to the input data some properties that are supposed to be preserved by the operations of the program. The correctness of the outputs can then be verified by checking whether the outputs meet those properties. Although an incorrect algorithm may also preserve the added properties, this is unlikely if the appropriate properties are used. Usually the properties used are those which are very sensitive to small changes in the program. Since programmers do not create their programs at random, it is reasonable to assume that if a program is incorrect, it is almost correct. Assertions that are sensitive to simple coding errors therefore usually provide very good fault coverage.

One technique that adds additional properties to data is *data encoding*. A program can be viewed as a function over an input domain with values in an output range. If a program is designed to process information encoded in some error detecting code, then we can detect faults by observing the program's coded output. This technique has been used in designing self-checking circuits [1]. In the remainder of this section, we will discuss its application to self-checking programs.

A checksum technique is presented in [2] for detecting and correcting errors when matrix operations are performed on a multiple processor system. The checksum encoding is done as follows. given an n by m matrix A, we transform it into an (n+1) by (m+1) matrix. The elements $a_{i,m+1}$ for $1 \leqslant i \leqslant n$ and $a_{n+1,j}$ for $1 \leqslant j \leqslant m$ are the row checksums and column checksums, respectively and they are computed as follows:

$$a_{i,m+1} = \sum_{j=1}^{m} a_{i,j} \quad \text{for } 1 \leqslant i \leqslant n$$

(row checksums)

$$a_{n+1,j} = \sum_{i=1}^{n} a_{i,j} \quad \text{for } 1 \leqslant j \leqslant m$$

(column checksums)

The other elements are information elements and are the same as the corresponding elements in the original matrix. The strategy used in the checksum technique is to encode the input matrices; after manipulation, the output matrix is expected to be in the encoded format, otherwise, errors have occurred.

This encoding scheme can be used to design SCPs for matrix operations. It is shown in [2, 3] that the encoding scheme is preserved by the following matrix operations: addition, multiplication, scalar product, LU-decomposition, transposition, Gaussian elimination and inversion.

We show in the following a template for SCPs that use data encoding techniques:

```
procedure coding_scp(var input,output: data-type;
                     var error: boolean);
begin
    Reading input;
    Encoding input;
    output ← Data manipulation;
    if coded(output)
        then begin
            error:= false;
            output:= decoding(output)
        end   (* then *)
        else error:= true
end;  (* coding-technique *)
```

Depending on the error detecting code, the decoding of the output may not be necessary. A SCP for matrix multiplication that uses the checksum technique is given in the following. The SCPs for other matrix operations can be obtained similarly using the given template.

```
program mat_mul;
var A: array[1..l+1,1..m] of real;
    B: array[1..m,1..n+1] of real;
    P: array[1..l+1,1..n+1] of real;
    i,j,k: integer;
begin
    (* reading input *)
    for i:= 1 to l do
        for j:= 1 to m do read(A[i,j]);
    for i:= 1 to m do
        for j:= 1 to n do read(B[i,j]);
    (* encoding input *)
    call column_chksum(A,l,m);
        (* compute column checksums *)
    call row_chksum(B,m,n);
         (* compute row checksums *)
    (* matrix multiplication *)
    for i:= 1 to (l+1) do
        for j:= 1 to (n+1) do
            begin
                P[i,j]:= 0;
                for k:= 1 to m do
                    P[i,j]:= P[i,j] + A[i,k] * B[k,j]
            end;
    (* check coded output *)
    if coded(P,l+1,n+1)
        then print P
        else print "faults detected"
end;   (* mat-mul *)
```

The data encoding technique can also be used in non-numeric problems. For instance, suppose we want to sort a list of n elements:

$$L = (x_1 \; x_2 \; \ldots \; x_n).$$

We can encode the list L to get LC as follows:

$$LC = (\hat{x}_1 \; \hat{x}_2 \; \ldots \; \hat{x}_{n-1} \; \max \; C)$$
$$\text{where } \max = \text{maximum}(x_1 \ldots x_n)$$
$$C = \left(\sum_{i=1}^{n} x_i \right) \mathit{mod} \; \max$$
$$\hat{x}_i = x_j \quad \text{for some i and j.}$$

Once the list is in its encoded form, we need to sort only the first n-1 elements of the encoded list because the nth element is already in its correct location.

When the sorting process is done, we can check its correctness as follows:

(a) Check the sorting relationship of the elements in the sorted list.

(b) Compare C with $\left[\left(\sum \textit{sorted elements} \right) \mathit{mod} \; \max \right]$

The first check is to make sure that the sorted list is a monotone sequence. The second check ensures that the sorted list is a permutation of the original list.

Since most, if not all, sorting algorithms use a loop structure to process the list, each list element is processed by the same set of instructions. If a software fault in the set affects a list element, then it likely affects most other list elements as well. The same is true for permanent hardware faults because the same set of hardware components are used for each iteration of the loop. Since the probability that most elements in the output list are incorrect yet the list satisfies the proposed assertions should be almost zero, the assertions are very effective for detecting software faults and permanent hardware faults. Although a transient error might affect only one list element, we can prove that it will be detected by the assertions. The proof is given in the following.

Let L be a list and $\hat{L}$ = sorted(L) = $(x_1 \ \ldots \ x_i \ \ldots \ x_n)$. Suppose that there is a change in the value of the element at position i due to a transient error that occurred during the sort, we then have:

$$\tilde{L} = (x_1 \cdots \tilde{x}_i \cdots x_n), \qquad where\ \tilde{x}_i = x_i + e$$

$$\tilde{x}_i \text{ is undetected} <==> \left[\left(\sum_{i=1}^{n} x_i\right) + e\right] mod\ x_n = C$$

$$<==> \left[\left(\sum_{i=1}^{n} x_i\right) mod\ x_n + \left(e\ mod\ x_n\right)\right] mod\ x_n = C$$

$$<==> \left[C + \left(e\ mod\ x_n\right)\right] mod\ x_n = C$$

$$<==> e = 0$$

Comparing this technique with that of Example 1, we see that this technique does not require the inputs for the validation process. This is particularly important when the list is long.

4. EMPIRICAL RESULTS

4.1 Methodology

In this subsection, we present a fault simulation technique for the evaluation of self-checking properties with respect to both software and hardware faults. The central idea is to introduce simulated faults by applying mutation transformations to produce mutant programs. These mutants are then executed to measure the ability of the set of assertions to distinguish the program from its mutants.

The technique of mutation transformation has been used in software testing [23]. A set of 25 mutant operators that represent common programming errors are used in [24, 25]. These mutant operators can be used for our purposes to evaluate the effectiveness of the SCPs with respect to the software bugs.

For the study of hardware fault coverage, we could write a program to simulate the system for which the SCPs were written. However, since hardware systems are designed differently, this approach would require a different version of the simulator for each system. This is not very cost-effective. Alternatively, we can simply compile the source program to obtain the assembly code. Simulated physical faults are then introduced into the assembly code by applying appropriate hardware mutation transformations (to be described). The mutant programs are then executed and the effectiveness of the assertions against hardware faults can be determined. This technique is diagrammed in Figure 3. Since this scheme performs the simulation on the real system, it is therefore also more reliable than the former approach.

The proposed fault simulation technique is an error-based testing method. It requires the definition of a set of mutation transformations and classes of faults that are considered. The set of faults that constitute the fault model must correctly characterize possible physical failures in the hardware system. Our hardware fault model is based on the earlier functional fault models for complex processors [26,27]. We extend the existing fault model to include faults in the memory. The fault model is given in the following:

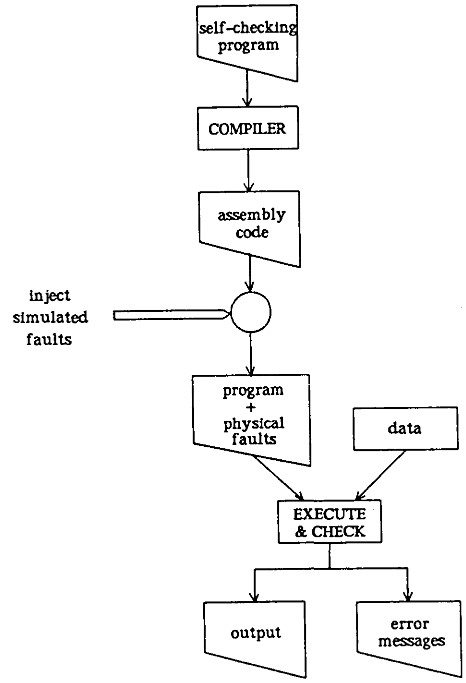


Figure 3. Strategy for evaluating SCPs with respect to hardware faults

(A) **Addressing faults:** Faulty decoding circuits can cause the following faults:
- no storage location is selected
- a wrong storage location is selected
- more than one storage location is selected.

(B) **Instruction decoding faults:** Failures in instruction decoding can cause the following faults:
- no instruction is executed
- a wrong instruction is executed
- more than one instruction is executed.

(C) **Faults in storage elements:** Some of the bits in the storage elements are stuck-at-0 (s-a-0) or stuck-at-1 (s-a-1). Transient failures can also change the contents of registers or memory locations arbitrarily.

Definition 3: A storage element (a register or a memory location) is a *sink* for an instruction if the execution of the instruction modifies its contents.

Definition 4: A storage element is a *source* for an instruction if it provides an operand for the execution of the instruction.

We now describe in the following the mutation transformations for the three classes of hardware faults described in the fault model:

(A) Addressing error:

a) No storage element is selected
- Delete from the program the one address instructions (e.g., LOAD, STORE, ...) that involve the affected storage element.
- Delete from the program the instructions that have the affected storage element as their sinks.
- If an instruction has the affected storage element as a source, replace the source with a constant zero, or a different storage element whose content is zero.

b) Selection of a wrong storage element
- Replace the correct operand by the incorrect one.

c) More then one storage elements are selected
- If the storage element is a sink, add instructions to copy its contents to the incorrectly selected storage elements.
- If the storage element is a source, add before the affected instruction instructions to logically AND its contents with those of the incorrectly selected ones.

(B) Instruction decoding errors:

a) No instruction is selected
- Remove the affected instructions.

b) Wrong instruction is selected
- Replace the affected instructions by the incorrectly selected one.

c) More than one instruction is selected
- Add instructions to logical AND the affected sources and/or sinks

(C) Faults in storage elements:

Stuck-at faults are simulated by setting or clearing the corresponding bits in the affected storage elements.

In the experiments, we assumed that shorts between wires are equivalent to the wired AND of the affected lines. Although this is true in nMOS circuits, it is technology dependent. We also assumed that when more than one instruction is executed simultaneously, the individual operations are performed correctly, and only the inputs and outputs of those operations are affected by the wired-AND faults. This is not always true. Some of the operations may share parts of the circuitry and the interference can cause the operations to be unreliable. We, however, believe that a more complex simulation will show that these faults will also be detected by our technique.

4.2 The Experiments

The experiments were done on the VAX 11/780. They included hardware faults, transient faults and software faults. Four Pascal programs were selected for the experiments based on their popularity:

(1) Sorting
(2) LU decomposition
(3) Solution of linear equations
(4) Eigenvalues and Eigenvectors of a real symmetric matrix

The first two programs use the data encoding technique described in Section 3. The third and fourth programs employ the techniques discussed in sections 2.1 and 2.2 respectively.

In order to be able to quickly perform the fault simulations reliably, we implemented a tool for the injection of simulated physical faults. For each test case, we specified a mutant operator. In response, the program asked for additional relevant information, and then performed the mutation transformation automatically by scanning the assembly code and modifying the affected instructions accordingly. For transient faults, we used the debugger ADB [28] available on the VAX. ADB is a general purpose debugging program. It provides capabilities to examine files and a control environment for the execution of UNIX programs. We ran the SCPs under the control of ADB, and transient faults were injected by setting break points and modifying the object files. For software faults, we manually modified the source code for each test case. The following relevant mutation transformations were studied in our experiments of software fault detection:

- change operators
- modify the indices of loops
- change the loop termination conditions
- change operands
- alter control structures
- modify predicates

We tried about 12,000 test cases of permanent hardware faults, 3,600 test cases of transient faults and 220 test cases of software faults. The results of the experiments are summarized in Table 1. The tables in the Appendix give the types of faults and the resulting errors detected during the experiments. We found that essentially all errors that affected the data manipulation were detected. Undetected errors were due to permanent hardware faults that caused infinite execution or which affected the execution of the assertions. For instance, if an instruction decoding fault changes the incrementing of a loop index to a decrement, then the loop will be executed indefinitely. In this situation, if the assertions are located outside the loop, then the execution will never reach them and the errors will not be detected. For another example, if an addressing fault causes the assertion to compare the wrong values, then faulty results might not be detected.

There were a few cases where the errors due to hardware faults were detected by the operating system, for example,

- tried to read pass end of file
- illegal instruction
- memory fault
- bad data found on real read
- floating exception
- bus error, etc...

before the program had a chance to check its output (counted as "detected"). Particularly, permanent s-a-1 faults, especially those at the more significant bits tend to cause floating exception. The errors due to this class of faults are generally detected by the operating system.

Table 1. Coverages of errors due to faults

Program	Type of faults		
	Hardware	Transient	Software
Sort	94.47%	100%	100%
LUdecomp	94.38%	100%	100%
Linear eqtns	98.60%	100%	100%
Eigen	99.11%	100%	100%

There are few cases of errors due to transient and software faults that are not detectable by the CED technique. They are some of those faults that affect only the input/output operations of the programs. If a transient or software fault affects only the input operations, it effectively changes the input data and the SCP cannot recognize the faulty data, unless the data were originally stored in some encoded format. Nor can a SCP detect transient and software faults that affect the outputs after the correctness of the outputs has been verified by the assertions. These classes of faults are beyond the scope of the proposed CED technique, and are not considered in the computation of the error coverages. In fact, most of transient faults that occur during the I/O operations are likely to cause memory faults, bus error, etc... and will be detected by most operating systems. We also do not consider those faults (mostly transient) that have no effect on the outcomes of the computations. It is quite interesting to know that many errors, particularly due to transient and software faults, are primarily detected by the CED technique.

5. HARDWARE SUPPORT

Although added assertions can improve significantly the reliability of the computing systems, the software redundancy also degrades the system performance. The performance overhead is from 7.27% to 32.75% for the programs used in our experiments. However, this is the price paid for the added reliability. Other reliability techniques such as duplication [29], RESO [30] or N-version programming [31] have much higher hardware or performance overhead.

To improve the performance, a dual processor architecture an be used in conjunction with SCPs. This architecture can also improve the reliability of the CED system. In a single processor system, if the processor is faulty, then the checks performed on the faulty processor may not be reliable. In our experiments, a large number of the undetected errors are of this class. The dual processor system can solve this problem by having each process and its checks executed on different processors. For instance, suppose we want to multiply two checksum encoded N by N matrices; every two rows of the product matrix can be computed simultaneously on the two processors, and each row checksum can also be checked in parallel on the other processor (i.e., if processor 1 computes row i, then the checksum properties of row i should be checked by processor 2). This scheme ensures that if one processor fails, the faulty results it computes will be detected by the good processor.

Comparing this technique with the traditional dual redundancy systems, the traditional technique has the advantages of being simpler and having slightly better hardware fault coverage. The proposed technique, however, has benefits not available in the traditional approach. Most noticeably, the traditional technique dose not detect software faults. Besides, if one of the processors fails, our experiments shows that the remaining processor of the proposed dual processor system can still rely on the self-checking capabilities of the software to perform considerably more reliable computations in a degraded manner. The performance overhead is also better in the proposed scheme. Extending this concept further, we see that the reliability of SCPs can also be improved by having the data manipulation and checks done on different processors of a distributed system. The proposed CED technique is therefore very attractive for many critical applications, particularly those which demand highly reliable software. The scheme is thus a novel type of design diversity [32].

6. CONCLUSIONS

In this paper, we presented a methodology for the design of self-checking programs in which the assertions are systematically generated from the design of the program. Software developed using the proposed methodology is guaranteed to be self-checking with respect to software faults. A less rigorous technique that adds additional properties to programs for the self-checking purposes was also discussed. This technique is particularly useful when the number of inputs is large.

We also described in this paper a fault simulation technique that can be use to study the hardware and software fault coverage of SCPs. This technique was used in our empirical study. The experimental results suggest that SCPs provide excellent coverage against both hardware and software faults. The effectiveness of higher level versus lower level error detection technique is a controversial issue. The proposed fault simulation technique allows us to study the effectiveness of the SCPs with respect to the whole spectrum of computing faults: low level faults such as stuck-at faults in the storage elements, functional level faults such as those that cause decoding errors, and system level faults such as bugs in the software.

We also proposed a dual processor technique for SCP processing. This scheme can improve both system performance and reliability. We believe that the proposed approach is superior in many aspects to the traditional redundancy methods such as duplication, N-version programming or signatured instruction streams.

REFERENCES

[1] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, New York: North-Holland, 1978

[2] K-H Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Comput.*, vol. C-33, pp. 518-528, June 1984.

[3] J-Y Jou and J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of the IEEE Special Issue on Fault Tolerance in VLSI*, vol. 74, no. 5, pp. 732-741, May 1986.

[4] P. Banerjee and J. A. Abraham, "Fault-Secure Algorithms for Multiple-Processor Systems," *Proc. IEEE Int'l Symp. on Computer Architecture*, pp. 279-287, June 1984.

[5] S. S. Yau and F-C Chen, "An Approach to Concurrent Control Flow Checking," *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 126-137, Mar. 1980.

[6] T. Sridhar and S. M. Thatte, "Concurrent Checking of Program Flow in VLSI Processors," *Proc. Int'l Test Conf.*, pp. 191-199, 1982.

[7] J. P. Shen and M. A. Schuette, "On-Line Self-Monitoring Using Signatured Instruction Streams," *Proc. Int'l Test Conf.*, pp. 275-282, 1983.

[8] D. J. Taylor, D. E. Morgan and J. P. Black, "Redundancy in Data Structures: Improving Software Fault Tolerance," *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 585-594, Nov. 1980.

[9] D. J. Taylor, D. E. Morgan and J. P. Black, "Redundancy in Data Structures: Some Theoretical Results," *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 595-602, Nov. 1980.

S. S. Yau and R. C. Cheung, "Design of Self-Checking Software," *Proc. Int'l Conf. on Reliability Software*, pp. 405-457, April 1975.

[11] D. M. Andrews, "Using Executable Assertions for Testing and Fault Tolerance," *Proc. 9th Int'l Symp. on Fault Tolerant Computing*, pp. 102-105, June 1979.

[12] A. Mahmood, E. J. McCluskey and D. J. Lu, "Concurrent Fault Detection Using A Watchdog Processor and Assertions," *Proc. Int'l Test Conf.*, pp. 622-628, 1983.

[13] A. Milli, Self-Checking Programs: An Axiomatic Approach to The Validation of Programs by The Use of Assertions, PH. D. dissertation, Dept. Comp. Sci., University of Illinois, Urbana, 1981.

[14] J. P. Black et al., "A Case Study in Fault Tolerant Software," *Software - Practice and Experience*, vol. 11, pp. 145-157, 1981.

[15] S. L. Hantler and J. C. King, "An Introduction to Proving The Correctness of Programs," *Computing Surveys*, vol. 8, pp. 331-353, Sept. 1976.

[16] B. Elspas, "*The Semiautomatic Generation of Inductive Assertions For Proving Program Correctness*," (Interim Report, Project 2686), Stanford Research Institute, Menlo Park, California, July 1974.

[17] S. M. German and B. Wegbreit, "A Synthesizer of Inductive Assertions," *IEEE Trans. On Software Eng.*, vol. SE-1, pp. 68-75, 1975.

[18] S. M. Katz and Z. Manna, "The Logical Analysis of Programs," *CACM*, 19 (4), 1976, pp. 188-206.

[19] B. Weibreit, "The Synthesis of Loop Predicates," *CACM*, 16 (2), 1974, pp. 102-112.

[20] R. W. Floyd, "Assigning Meaning to Programs," *Proc. of the American Math. Soc. Symp. in Applied Math.* (vol. 19), Providence, R. I.: American Math. Soc., 1967, pp. 19-31.

[21] S. Katz and Z. Manna, "A Close Look at Termination," in *Current Trends in Programming Methodology*, vol. II, R. Yeh, Ed., Englewood Cliffs, NJ: Prentice-Hall, 1977, Chap. 10.

[22] John Greenstadt, "The Determination of The Characteristic Roots of A Matrix by The Jacobi Method," in *Mathematical Methods For Digital Computers*, vol. I, A. Ralston and H. S. Wilf, Eds. New York, NY: John Wiley & Sons, Inc., April, 1967, pp. 84-91.

[23] R. A. DeMillo and R. J. Lipton, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, April 1978, pp. 34-41.

[24] T. A. Budd et al., "The Design of A Prototype Mutation System for Program Testing," *National Computer Conference*, pp. 623-627, 1978.

[25] T. A. Budd and F. Sayward, "Users Guide to The Pilot Mutation System," *Yale University Rep. 114*, Dept. of Comp. Sci., 1977.

[26] S. P. Thatte and J. A. Abraham, "Test Generation for Microprocessors," *IEEE Trans. Comput.*, vol. C-29, pp. 429-441, June 1980.

[27] D. Brahme and J. A. Abraham, "Functional Testing of Microprocessors," *IEEE Trans. Comput.*, vol. C-33, pp. 475-485, June 1984.

[28] J. F. Maranzano and S. R. Bourne, "A Tutorial Introduction to ADB," in *Unix Programmer's Manual*, Seventh Edition, vol. 2A, Murray Hill, NJ: Bell Laboratories, Jan. 1979.

[29] J. von Neumann, "Probabilistic Logics and The Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies*, C. E. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton University Press, pp. 43-98, 1956.

[30] J. H Patel and L. Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Trans. Comput.*, vol. C-31, pp. 589-595, July 1982.

[31] L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Proc. 8th Int'l Sump. on Fault Tolerant Computing*, pp. 3-9, June 1978.

[32] A. Avizienis and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, vol. 17, no. 8, pp.67-80, Aug. 1984.

APPENDIX

The tables in this appendix give the types of faults considered in the experiments.

Table 2. Faults and resulting errors in the sort program.

Type of faults		Errors Detected by CED	Errors Detected by Operating System	Undetected Errors
Hardware	Faults in storage elements	310 (40.16%)	419 (54.27%)	43 (5.57%)
	Instruction Decoding Faults	151 (30.26%)	317 (63.53%)	31 (6.21%)
	Addressing Faults	323 (39.10%)	461 (55.81%)	42 (5.09%)
Transient	Control Faults	28 (46.67%)	32 (53.33%)	0 (0%)
	Instruction Decoding Faults	153 (60.00%)	102 (40.00%)	0 (0%)
	Addressing Faults	110 (68.75%)	50 (31.25%)	0 (0%)
Software	Mutations	34 (89.47%)	4 (10.53%)	0 (0%)

Table 3. Faults and resulting errors in the LUdecompose program.

Type of Faults		Errors Detected by CED	Errors Detected by Operating System	Undetected Errors
Hardware	Faults in storage elements	318 (37.86%)	471 (56.07%)	51 (6.07%)
	Instruction Decoding Faults	274 (42.35%)	355 (54.87%)	18 (2.78%)
	Addressing Faults	412 (54.64%)	285 (37.80%)	57 (7.56%)
Transient	Control Faults	23 (43.40%)	30 (56.60%)	0 (0%)
	Instruction Decoding Faults	109 (74.15%)	38 (25.85%)	0 (0%)
	Addressing Faults	89 (60.14%)	59 (39.86%)	0 (0%)
Software	Mutations	37 (94.87%)	2 (5.22%)	0 (0%)

Table 4. Faults and resulting errors in the linear equation program.

Type of Faults		Errors Detected by CED	Errors Detected by Operating System	Undetected Errors
Hardware	Faults in storage elements	154 (26.10%)	432 (73.22%)	4 (0.68%)
	Instruction Decoding Faults	112 (31.28%)	239 (66.76%)	7 (1.96%)
	Addressing Faults	272 (29.69%)	629 (68.67%)	15 (1.64%)
Transient	Control Faults	34 (66.67%)	17 (33.33%)	0 (0%)
	Instruction Decoding Faults	88 (83.81%)	17 (17.19%)	0 (0%)
	Addressing Faults	58 (82.86%)	12 (17.14%)	0 (0%)
Software	Mutations	43 (95.56%)	2 (4.44%)	0 (0%)

Table 5. Faults and resulting errors in the Eigen program.

Type of Faults		Errors Detected by CED	Errors Detected by Operating System	Undetected Errors
Hardware	Faults in storage elements	752 (49.25%)	771 (50.49%)	4 (0.26%)
	Instruction Decoding Faults	288 (37.11%)	472 (60.82%)	16 (2.07%)
	Addressing Faults	1526 (59.63%)	1010 (39.47%)	23 (0.90%)
Transient	Control Faults	57 (78.08%)	16 (21.92%)	0 (0%)
	Instruction Decoding Faults	187 (82.74%)	39 (17.26%)	0 (0%)
	Addressing Faults	108 (81.82%)	24 (18.18%)	0 (0%)
Software	Mutations	87 (96.67%)	3 (3.33%)	0 (0%)

STUCK-AT FAULT DETECTION IN PARITY TREES

Samiha Mourad*, Joseph L. A. Hughes, and Edward J. McCluskey

CENTER FOR RELIABLE COMPUTING
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California 94305

ABSTRACT

Algorithms for the generation of test sets for a parity checker tree of 2-input EXCLUSIVE-OR (XOR) gates are described. A minimal test set (3 patterns) detects all single stuck-at faults at the input leads of the XOR gates. Three test sets that guarantee 100% fault coverage for double stuck-at faults are developed. Their lengths increases logarithmically with the number of inputs of the tree. Two of these tests detect all single and double faults.

1 INTRODUCTION

There is general agreement that single faults do not represent the actual circuit failures, particularly in current IC technologies. The number of faults and the frequency with which some faults occur have been affected by two main changes: the reduction in size of the devices and the increase in circuit complexity. For example, during the fabrication process a single surface defect or a variation in processing parameters can cause multiple faults. As feature size decreases, this problem becomes more severe due to the larger relative size of surface defects.

Experimental investigations to assess the goodness of single stuck-at fault test sets for detecting multiple stuck-at faults were carried out by [Hughes 84,85]. These simulation studies of the 74LS181 4-bit ALU, using 10 single stuck-at test sets and one toggle-only test set, demonstrated very high multiple stuck-at fault coverage. The actual fault coverage was significantly higher than the fault coverage bounds based on an analysis of the network structure [Agrawal 81]. In order to confirm the usefulness of single fault test sets in detecting multiple faults, there is a need to evaluate other circuits that are commonly used in digital systems.

This paper analyzes the effectiveness of single stuck-at fault test sets in detecting multiple stuck-at faults in a very widely used circuit, a parity checker. First, an optimal single stuck-at test set that is suitable for any XOR gate implementation is presented. Then a reduced test set is developed to detect single stuck-at faults at the input pins of XOR gates.

The circuit is simulated to determine the effectiveness of these tests for detecting multiple faults. Test sets that detects all double stuck-at faults are developed.

2 PARITY NETWORKS

Probably the most widely used encoding scheme is parity. Parity checkers are used with memory arrays, registers, and buses. Parity has been the most popular checking scheme in real computer systems such as the IBM 4341 [Ciacelli81], the Sperry Univac 1100/60 processor [Boone 80], and the VAX 11/780 [Swarz 78]. Although parity may not be the most cost effective code for every situation, it is very widely used either by itself or in conjunction with duplication [Sedmak 80]. Self-checking parity checkers also are used in PLA testing [Khakbaz 82].

Since parity checkers are themselves circuit modules, they are also subject to circuit failures. Thus, not only should a parity checker be able to detect errors in its inputs, it also should be testable for faults within itself.

3 STUCK AT FAULTS

3.1 Single Stuck-At Faults

The parity tree in Fig. 1 consists of 2-input XOR gates. It has been shown [Bossen 70] that such circuits are fully tested for single stuck-at faults by an optimum test set that consists of only 4 patterns. This result is based on the following assumptions: 1) no internal realization of the XOR gate is assumed, and 2) all 4 input patterns are needed to fully test the XOR gate.

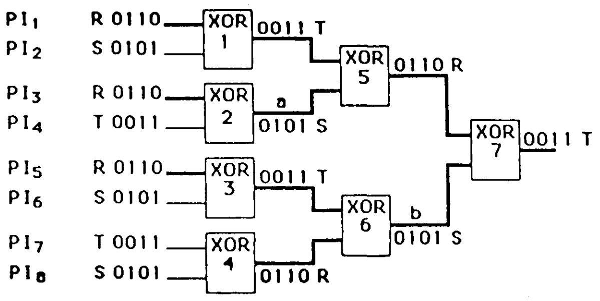


Figure 1. An 8-input parity tree.

* Also, with the Computer Engineering Department San Jose State University, San Jose, CA 95192.

CH2345-7/86/0000/0836$01.00©1986 IEEE

Three vectors, R, S and T consisting each of two '0's and two '1's are used to develop the Bossen test. The addition mod(2) of the vectors is defined by the matrix in Table 1.

Table 1. Addition table of the vectors R, S and T.

$\oplus$	0	R	S	T
0	0	R	S	T
R	R	0	T	S
S	S	T	0	R
T	T	S	R	0

0 is the all zeros vector.

The test is developed by first assigning one of the 3 vectors to the output of the tree. For the 8-input tree shown in Fig. 1, the output vector, T, at gate XOR7, is arbitrarily selected. Then employing the labeling procedure [Bossen 70] shown in Fig. 2(a), the input patterns for gate XOR7 are determined. They are the vectors R and S shown in Fig. 2(b). The patterns R and S are such that they form an exhaustive test set for XOR7. These two vectors are the outputs of gates XOR5 and XOR6, and the input vectors to these two gates are now determined in the same fashion.

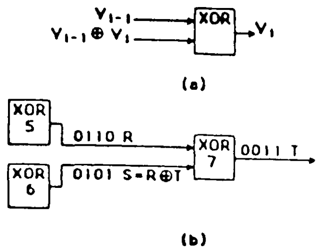


Figure 2. Gate labeling procedure (a) and example of test generation (b) for Bossen test.

The procedure is followed until all vectors have been generated. They are shown at the corresponding nodes in Fig. 1. The optimum test set thus generated (called Bossen throughout the rest of this paper) is formed by concatenating all vectors at the Primary Inputs (PI) 1 through 8. The test set is listed in Table 2. The labeling of an n-input tree may be generalized in the form of the following algorithm which is a variation of the algorithms developed by [Bossen 70] and [Seth 77].

Algorithm I

1. Let Z = {R, S, T}. Assign any vector in Z to the output of the tree.
2. Repeat until all leads are labeled: For each XOR gate whose output lead is labeled, assign to its inputs the two other vactors.
3. The vectors at the primary inputs of the tree form the test set.

Table 2. Bossen test for 8-input parity tree.

	Primary Input							
Pattern #	1	2	3	4	5	6	7	8
1	0	1	0	1	0	1	1	1
2	1	0	1	1	1	0	1	0
3	1	1	1	0	1	1	0	1
4	0	0	0	0	0	0	0	0
Vector Name	R	S	R	T	R	S	T	S

The test set listed above guarantees the detection of all single stuck-at faults at the input leads as well as at all internal nodes of the XOR gates for any gate implementation. If, however, only faults at the XOR gates input leads are to be detected, the first 3 patterns of the test shown in Table 1 are sufficient. Consider a single XOR gate. All 6 single stuck-at faults can be detected with one of the following two tests, UV or U'V', where U = {011} and V = {101}. Note that U $\oplus$ V = W, V $\oplus$ W = U, and W $\oplus$ U = V, where W = {110}, and that U, V and W are obtained from the vectors R, S and T by eliminating the last bit (which is 0 for all of them). Thus, using algorithm I and the set Z={U, V, W}, it is possible to generate a test that detects faults at the leads of the gates for any size tree. This test set will be called Mourad throughout the paper.

3.2 Double Stuck-At Faults

Although the generated test sets, Bossen and Mourad, detect all single stuck-at faults, they are not sufficient to detect all multiple stuck-at faults. This failure to detect double stuck-at faults will be explained next. For this we refer back to the circuit in Fig. 1. If node a and node b are both stuck-at-0, or both stuck-at-1, the sets of all tests that detect these multiple faults is given respectively by the Boolean functions [Breuer 76]

$$ab(d\,^2f/d(ab)) \quad + ab'\ (df/da) + a'b\ (df/db) \qquad (1)$$

$$a'b'(d\,^2f/d(ab)) + a'b\ (df/da) + ab'\ (df/db) \qquad (2)$$

where df/dx, x=a or b, is the Boolean difference of f with respect to x, and $d^2f/d(ab)$ is the Boolean difference of f with respect to a and b. The first term of either equation represents a sensitizing path from a and b, and the other terms, paths from a and from b respectively. These Boolean functions are easily evaluated [McCluskey 86] with the following results:

$$df/dx=1 \text{ for } x=a \text{ or } b, \qquad (3)$$

$$df/d(ab) = 0. \qquad (4)$$

Substituting these values, expressions (1) and (2) reduce to the same expression:

$$ab' + a'b \qquad (5)$$

The corresponding test patterns are 10 or 01. In general form, the test vectors at node a and node b cannot be equal if either of these double stuck-at faults is to be detected. It can be shown in a similar fashion, that if node a is stuck-at-1 (or 0) and node b stuck-at-0 (or 1), the set of test sets to detect these faults is given by the expression

$$ab + a'b' \qquad (6)$$

Here, the corresponding test patterns are 00 and 11. The test vectors at node a and node b cannot be complements of each other if either of these double faults is to be detected.

The constraints dictated by Eqs. (5) and (6) apply to any double stuck-at fault test. In the remainder of this section and in subsequent discussion regarding the effectiveness of single stuck-at faults tests to detect multiple faults, test Bossen will be used as an example.

Parity tree nodes that have the same label receive identical values for all vectors in the Bossen test set. Thus, if two identically labeled nodes are both stuck-at 0 or both stuck-at 1, the multiple fault will not be detected by the Bossen test set. For the example in Fig. 1 nodes PI1, PI3, PI5, and the outputs of gates XOR5 and XOR4 are all labeled with the vector R. Since no two of the Bossen's vectors are complements of each other, all double faults such that one node is stuck-at-1 (or 0) and the other stuck-at-0 (or 1) are detected by this same test.

If the two faulty nodes are along the same sensitizing path, it can be easily shown by a similar analysis that the multiple stuck-at fault will be detected by the Bossen test set because this case reduces to the detection of single stuck-at fault at one of the nodes. For the example in Fig. 1, the multiple fault PI1 stuck-at 0 and XOR5 output stuck-at 0 will be equivalent to the single fault at the output of XOR5.

Using these results, it is possible to calculate a lower bound on the fault coverage for double stuck-at faults in parity trees. This was found to be 83.33% {Mourad 86]. Comparison of these results with those reported in [Hughes84,85], shows that the multiple stuck-at fault coverage of single stuck-at fault test sets is significantly lower for the parity tree than for the 74LS181 4-bit ALU. The next section describes the development of tests that detect all double stuck-at faults.

4 DOUBLE FAULT TEST SETS

In order to detect double stuck-at faults on identically labeled nodes, these nodes of the parity tree must be assigned different vectors. That is, the labels X and Y on any two nodes must satisfy the following two properties

$$X \neq Y \quad (7), \qquad \text{and} \qquad X \neq Y' \quad (8)$$

First, tests that guarantee 100% detection of all double stuck-at faults at the leads of the XOR gates will be developed. Then test sets Bossen and Mourad will be modified and augmented to assign different vectors at the identically labeled nodes and hence to detect all double stuck-at faults.

From expressions (5) and (6) it is clear that a test set that detects double stuck-at faults on the nodes of a single XOR gate, should include a pattern holding both inputs at the same signal {00 or 11} and another pattern holding the inputs at opposite values {10 or 01}. Thus there are four possible test sets to fully detect double stuck-at faults on the input leads of an XOR gate. In vector notation, these test sets are AB, AB', A'B and A'B', where A = {1 1} and B = {1 0}. Notice that Bossen vectors can be expressed as concatenations of these two vectors: R=B'B, S=BB and T=AA'. The 4 vectors form the group, $G(Z_2{}^2, \oplus)$, where $Z_2{}^2$ is the a 2-bit, 2-symbols (0 and 1) code.

For a 4-input tree, the vectors A and B and their complements are not sufficient to uniquely label all its internal nodes. An extra bit can be appended to each vector in an effort to discriminate among them. There are 8 3-bit vectors, of these only 4 can be used because of the constraints given by Eq. (8). Hence, it is necessary to augment the bits in the vectors to 4. All 16 4-bit vectors form a group, $G(Z_2{}^4, \oplus)$. Due to the constraint given by Eq. (8), any 8 of these vectors that form a subgroup could be used in labeling the tree. Arbitrary, vectors 0 through 7 will be used. The set of these vectors will be denoted by Z_4. In general, Z_n is used to denote the set of all labeling vectors of an n-input tree. In a similar fashion it can be shown that at least 5 bits will be needed to generate the labels for an 8-input XOR tree. In general, an n-input tree (n>2) requires at least L bits, where $L=(\log_2(n)+2)$. Note however, that if the all '0' (or all '1') vector is used, the node assigned such a vector will not be tested for single stuck-at-1 (or at-0).

Next, the assignment of the vectors to the nodes will be explained. Assuming the output of an XOR gate is labeled, then the inputs to this gate can be labeled using elements of Z_n. Now these 2 labels can be used to label the inputs of their corresponding gates. However, in selecting the labels, care should be taken in order to avoid duplicating previously used labels.

Algorithm II

```
1. Let $Z_n$ be the set of all labeling vectors.
   Assign any vector from $Z_n$ to the output of the tree.
2. Repeat until all leads are labeled:
    For every XOR gate whose output is labeled with
    a vector, say x in $Z_n$, assign to its inputs
    the 2 vectors y and z such $x=y \oplus z$ and x and
    $y \ ( \{Z_n-Z_i\}$, where $Z_i$ is the set of all labels
    that were already assigned to leads other than
    those in the same sentisising path.
3. The vectors at the primary inputs of the tree
   form the test set.
```

Algorithm II is used to generate test sets for 4-and 8-input parity trees. The tests are listed in Table 5. They guarantee 100% fault coverage for double stuck-at faults. They do not, however, guarantee the detection of all single stuck-at faults.

Table 3. Minimal tests to detect all double and some single stuck-at faults.

Pattern	4-input	8-input
#	1 2 3 4	1 2 3 4 5 6 7 8
1	1 0 0 0	1 1 1 1 1 1 0 1
2	0 1 0 0	1 0 0 1 0 0 0 1
3	0 0 0 1	0 1 0 1 1 0 0 0
4	0 0 0 0	0 1 1 0 0 0 0 1
5	- - - -	0 0 0 0 0 0 0 0
Vector	B B'A'A'	A B B A B B A'A
	A'A'A'B	A'A B'B B A'A'B'
		0 0 0 0 0 0 0 0

It is desirable to develop tests that yield high coverage for both single and double stuck-at faults. The approach is as follow: start with the Bossen test (exhaustive test for every XOR gate) or the Mourad test (detection of faults at the input leads of the gates) then augment this test in such a way that all the nodes are assigned unique vectors.

As was demonstrated earlier [Mourad 86], an n-input parity tree has at most J nodes labeled R, S or T, where J=k or k+1 and 2n-1=3k or 3k+1. Therefore, at least J distinct vectors that pairwise satisfy Eqs. (7) and (8) need to be appended to the Bossen (or the Mourad) vectors in order to distinguish between identically labeled nodes. To form the required J vectors at least $\lceil \log_2 J \rceil = \log_2(n)$ bits are needed. Hence, the minimal test that detect all single stuck-at and all double stuck-at faults on a n-input tree is $\log_2(n) + 4$.

Algorithm II is used to label the tree. Here, the vectors of the set Z_n, are formed by concatenating the Bossen's vectors and Z_2^J. Algorithm II was used to generate test sets for the 4-input and an 8-input parity trees. The test sets are listed in Tables 4 and 5.

Table 4. Tests to detect all single and double faults at the input leads of the XOR gates.

Pattern	4-input	8-input
#	1 2 3 4	1 2 3 4 5 6 7 8
1	0 1 0 1	0 1 0 1 0 1 1 1
2	1 0 1 1	1 0 1 1 1 1 1 0
3	1 1 1 0	1 1 1 0 1 0 0 1
4	1 1 1 1	1 1 0 1 0 0 1 0
5	0 1 1 0	1 0 1 0 0 1 1 1
6		0 1 1 1 0 0 1 1
Vector Name	U V U W	U V U W U W W V
	B'A A B'	A B B'B A'B'A B'
		0 1 1 1 0 0 1 1

Table 5. Tests to detect single and double faults at the input leads as well as the internal nodes of the XOR gates.

Pattern	4-input	8-input
	1 2 3 4	1 2 3 4 5 6 7 8
1	0 1 0 1	0 1 0 1 0 1 1 1
2	1 0 1 1	1 0 1 1 1 1 1 0
3	1 1 1 0	1 1 1 0 1 0 0 1
4	0 0 0 0	0 0 0 0 0 0 0 0
5	1 1 1 1	1 1 0 1 0 0 1 0
6	0 1 1 0	1 0 1 0 0 1 1 1
7		0 1 1 1 0 0 1 1
Vector Name	R S R T	R S R T R T T S
	B'A A B'	A B B'B A'B'A B'
		0 1 1 1 0 0 1 1

A summary of all test sets discussed in this paper is given in Table 7. The fault type is listed in the first column. Then the number of patterns for parity trees with inputs ranging from 2 to 16 are listed in the next columns.

Table 6. Summary of all tests for the parity tree.

	Number of primary input leads			
Type of stuck-at fault	2	4	8	16
Single at pins only	3	3	3	3
Single at pins and internal	4	4	4	4
Double mainly	2	4	5	6
Double and single at pins	3	5	6	7
Double at pins and single at pins and internal nodes	4	6	7	8

5 SUMMARY AND CONCLUSIONS

This paper reports a study on the effectiveness of single stuck-at faults test sets in detecting double stuck-at faults for a parity tree, and the development of double stuck-at fault tests for this circuit. Such circuits are extensively used as code checkers in logic circuits and communication systems. The parity tree is completely tested for single stuck-at faults with a 4-pattern test [Bossen 70]. However, this test yields unacceptable fault coverages for multiple stuck-at faults. For a more practical tree size, 32-input, the coverage is 85%. The coverage is estimated to decrease with increased number of primary inputs to 83.33%.

The results reported here are significantly lower than those obtained for the 4 bit ALU [Hughes 84,85]. However, this is not surprising because the circuit is symmetric, has one output and, naturally, comprises many XOR gates.

Although single stuck fault tests appear to provide adequate multiple fault coverage for some circuits, the analysis of the parity tree demonstrates that such high fault coverage is not guaranteed for all circuits. Thus, further experimentation is needed on other circuits in order to fully evaluate the effectiveness of single stuck-at fault test sets in detecting multiple stuck-at faults.

Three test sets for double stuck-at faults are developed in this paper for parity trees. The first test set guarantees 100% detection of double stuck-at faults but misses some single stuck-at faults. The second test set fully detects single as well as double stuck-at faults but only at the input leads of the gates. Finally, The third test set detects all single stuck-at faults and double faults at the leads. Unlike the results reported in [Seth 77], the length of these test sets increases logarithmically with the number of inputs leads of th

6 ACKNOWLEDGEMENTS

This research was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract no. N00014-85-K-0600. The authors wish to thank Laung-Terng Wang and Jon Udell for their suggestions. Thanks are also due to David and Joseph McCluskey for their assistance in preparation of this manuscript.

7 REFERENCES

[Agarwal 81] Agarwal, V.K. and A.S.F. Fung, "Multiple Fault Testing of Large Circuits by Single Fault Test Sets," IEEE Trans. Comp., Vol. C-30, No. 11, pp. 855-865, November 1981.

[Boone 80] Boone, L.A., H.L. Liebergot, and R.M. Sedmak, "Availability, reliability, and maintainability of the Sperry Univac 1100/60," Digest, The 10th Annual International Symposium on Fault-Tolerant Computing (FTCS-10), pp. 3-8, Kyoto, Japan, October 1980.

[Bossen 70] Bossen, D.C., D.L. Ostapko, and A.M. Patel, "Optimum Test Patterns for parity Networks," Proc. AFIPS Fall 1970 Joint Computer Conference, Vol. 37, pp. 63-38, Houston, Texas, November 1970.

[Breuer 76] Breuer, M.A. and A.D. Friedman, Diagnosis and Reliable Design of Digita Systems,Computer Science Press, 1976.

[Ciacelli 81] Ciacelli, M.L., "Fault handling on the IBM 4341 processor," Digest, The 11th Annual International Symposium on Fault-Tolerant Computing (FTCS-11), pp. 9-12, Portland, Maine, June 1981.

[Hughes 84] Hughes, J.L.A., and E.J. McCluskey, "An Analysis of the Multiple Fault Detection Capabilites of Single Stuck-At Fault Test Sets," Proc. International Test Conference, pp. 52-58, Philadelphia, Pennsylvania, October 1984.

[Hughes 85] Hughes, J.L.A., S. Mourad, and E.J. McCluskey, "An Experimental Study Comparing 74LS181 Test Sets," Digest, COMPCON Spring 85, pp. 384-387, San Francisco, California, March 1985.

[Khakbaz 82] Khakbaz, J. and E.J. McCluskey, "Concurrent error detection and testing for large PLA's," Joint Special Issue on VLSI Systems IEEE Journal of Solid State Circuits, vol. SC-17, no. 2, pp. 386-394; and IEEE Trans. Electron Devices, vol. ED-29, no. 4, pp. 756-764, April 1982.

[McCluskey 86] McCluskey, E.J., Logic Design Principles with Emphasis on Testable Semi--Custom Circuits, Prentice Hall, Engelwood Cliffs, NJ, 1986.

[Mourad 86] Mourad, S., J.A.L. Hughes and E.J. McCluskey, "Multiple Faut Detection in Parity Trees," Digest, COMPCON Spring 86,pp. 441-444, San Francisco, March 1986.

[Sedmak 80] Sedmak, R.M., and H.L. Liebergot, "Fault-tolerance of a general purpose computer implemented by very large scale integration," Special Issue on fault-tolerant computing, IEEE Trans. Comp., Vol. C-29, No. 6, pp. 492-500, June 1980.

[Seth 77] Seth, S.C. and K.L. Kodandapani, "Diagnosis of Faults in Linear Tree Networks," IEEE Trans. Comp., Vol. C-26, No. 1, pp. 29-33, January 1977.

[Swarz 78] Swarz, R.S., "Reliability and maintainability enhancements for the VAX-11/780," Digest, The 8th Annual International Symposium on Fault-Tolerant Computing (FTCS-8), pp. 24-28, Toulouse, France, June 1978.

A TWO LEVEL GUIDANCE HEURISTIC FOR ATPG

Tom Kirkland

MCC
9430 Research Blvd.
Austin, Tx 78759

M. Ray Mercer

Univ. of Texas at Austin
ENS 143
Austin, TX 78712-1084

ABSTRACT

A heuristic for guiding ATPG algorithms is presented based on a two-level view of the circuit, functional and topological. The heuristic eliminates many of the drawbacks of previous guidance measures since it reduces the number of potential conflicts that may arise due to reconvergent fanout. The procedure for calculating the guidance measure is explained and the selection rules are illustrated.

INTRODUCTION

The use of search algorithms for Automatic Test Pattern Generation (ATPG) raises questions concerning the most efficient way to organize the search to produce a test in the shortest possible time[1,2,3]. These questions are usually answered, at least in part, by some type of guidance measure, such as controllability and/or observability[2,4]. However, most of the measures used to date have been derived by a single-level algorithm which computes values based only on locally derived information such as the SCOAP measure[5]. This process is easy to program and executes rapidly, but misses many important details about the overall topology of the circuit. As we will show, the topology affects the potential for conflict in node assignments during ATPG and, therefore should be included in any heuristic measure used for ATPG guidance. First, we must look at the types of error introduced by measures based only on local information. Then we describe a two-level measure that eliminates or reduces the potential for conflict during the backdrive process of ATPG.

We assume that a search algorithm is being used for ATPG that requires choices to be made during backdrive from the fault site. This algorithm could be the D-Algorithm, PODEM, FAN, or any similar method that searches a node space until a test is found or it is determined that none exists. Whenever more than one node is a candidate for inclusion in the search tree at a given point in the process, a choice must be made. These choices can be made randomly, but the use of node weights can provide much better choices, thereby reducing the computing time required to find a test. A node weight based on controllability is often used for this purpose, usually based on the SCOAP measure. We will assume that such a measure is used, although the errors we discuss are not specific to this particular choice, but rather exist in all measures that compute new weights only from local information. This measure will be the first level in the two-level heuristic and will be called the C measure in subsequent discussions. The C measure will attempt to measure the minimum number of inputs that must be set to control a given node.

LOCAL INFORMATION ERRORS

Any weighting based on propagating local measures fails to account for the presence of multiple paths in a circuit. These multiple paths introduce dependencies in the weights that are not accounted for by the local computation process. Consider the circuit shown in Figure 1 which illustrates the computation of controllability values for a partially uncontrollable network. The process begins by assigning to each input a zero and one controllability value of 1. This is denoted by the pair of numbers separated by a slash with the zero value before the slash and the one value after it. The process then computes the controllability of every other gate in the network on the basis of the values of its inputs. Of course, only those gates whose inputs are known are current candidates for computation. Therefore, propagating the 1/1 values from input A through inverter B is done by swapping the zero and one values. However, when we attempt to compute the controllability of AND gate C, we realize that the process will cause an error. Blindly using the input values of C, we compute a controllability of 1/2, but an examination of the circuit reveals that the output of C cannot be controlled to a one at all.

CH2345-7/86/0000/0841$01.00 © 1986 IEEE

The error illustrated in Figure 1 will be called Type 1 error. This error results in a controllability figure that is overly optimisitic. The optimism is caused by the assumption that the input weights of a gate are independent. In the presence of reconvergent fanout, this assumption is not valid. Another type of error that also results from this independence assumption is illustrated in Figure 2. Once again, the controllability weights are given for the same circuit but with the inverter removed. The method will compute the controllability of C to be 1/2, and once again it is wrong since it is just as easy to control C to a one as to a zero. This type of error, which causes a pessimistic value, will be called Type 2 error.

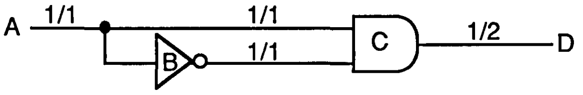


Figure 1. Type 1 Error

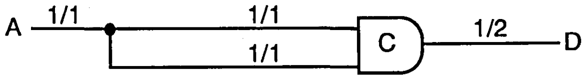


Figure 2. Type 2 Error

While the circuits used to illustrate the errors have been simplified to reveal the error in a graphic way, they are not contrived. Consider the circuit of Figure 3, which illustrates a more complex Type 1 error. The one controllability of gate L is computed to be 2, while in fact it should be 4. This error is due to the fact that the computation assumes that D can be a one and a zero at the same time. This can be seen more clearly by looking at Figure 4, which shows the assumptions made at each stage of the controllability calculation. Capital letters indicate an assumed true value for the variable while small letters indicate an assumed complemented value for the variable. As can be seen by examining the lists of assumptions at the inputs to L, both D and D-bar (d) are present. This means that along one path D was assumed to have a true value (1), while along the other path it was assumed to be false (0). Obviously, both assumptions cannot be true at the same time, and the result is a Type 1 error at L.

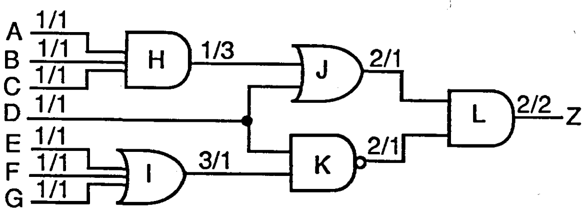


Figure 3. Non Local Type 1 Error

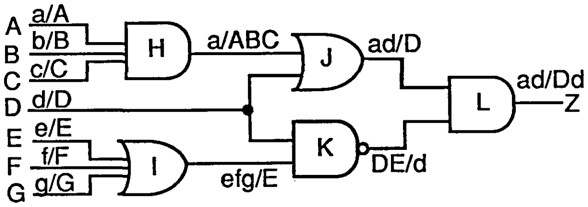


Figure 4. Controllability Assumptions

It is possible to correct for Type 1 error by a process illustrated in Figure 5. By carrying along the assumptions associated with each zero and one controllability value, Type 1 error can be detected whenever an inconsistent set of assumptions is found. In Figure 4, for example, when the one controllability of L is about to be computed, the input assumption lists are compared. Since a one at L requires a one on every input, all the input assumptions must be consistent. As we have already seen, they are not. Therefore, one or more of the input assumptions must be changed before the one controllability of L is computed. To change an assumption requires backtracking in the circuit and selecting a different assumption at a previous node. For our example this backtracking carries us to J, where it was possible to choose either D or H as the source for controlling J to one. Previously we chose D, which caused the conflict. During backtracking we will choose the other alternative, H, since it does not contain any terms that conflict with the other input assumptions of gate L at which the conflict was detected. By carrying this new choice forward to L, we resolve the conflict by controlling L to a one with H and D-bar. The conflict has been resolved and a consistent one controllability has been found for L.

This backtracking process does introduce additional complexity into the controllability computation, but, if efficiently implemented, does not consume much computer time for most circuits. In many systems, this information is at least partially available from other sources such as simulation of manually or randomly generated patterns. Since the elimination of Type 1 error is so essential to the development of a good guidance heuristic it is worth the extra computation time. It must be realized that this backtracking must be done at some point, since, if left uncorrected, the ATPG algorithm will encounter the same conflict and backtrack itself. The difference is that the ATPG may backtrack many times from the same node because of the same conflict. It is better to resolve the conflict once and for all. If we assume that every stuck-fault in the circuit is to be tested, then the ATPG must find a way to control each node to both a zero and a one at some time. It is best to find at least one way to do this in a single computation rather than resolving it over and over again during ATPG. Later, we will give some rules for selecting assumptions during the forward computation which reduce the need for backtracking and the burden of deciding which path to backtrack when a conflict is found.

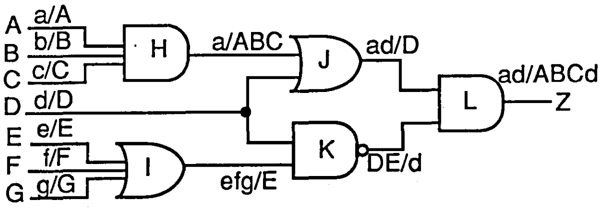


Figure 5. Correcting Type 1 Error

If we carry the assumption lists along during the controllability calculation, we can also use them to correct for certain Type 2 errors. It should be realized that Type 2 errors do not cause as much trouble for ATPG guidance as Type 1 errors, and therefore their correction is useful, but not essential. Type 2 errors result from "double counting" during the controllability calculation when the same variable appears with the same parity in more than one input assumption list. If we simply add the input controllability values the sum will include the same variable more than once, resulting in a pessimistic value for controllability. This pessimism can be partially eliminated by taking the union of the input lists and then computing its length. This value includes each variable only once due to the union operation and thereby eliminates double counting.

It should be noted, however, that this process does not eliminate all Type 2 error. It is possible for some Type 2 error to remain which we call invisible Type 2 error. This is illustrated in Figure 6. Because of the choice of K to control R to a zero, it is not possible to detect the Type 2 error that occurs at T. As can be seen, the zero controllability of T has been calculated as 2. This is clearly an error since holding L to a zero will force T to a zero regardless of the values of any other inputs. Therefore, Type 2 error exists at T which is invisible to the process set out above. This invisible Type 2 error occurred because L was not used in both path choices from L to T. If it had been, then the error would not have remained invisible. Due to the rules for selection of nodes which we will give later, it is possible for invisible Type 2 error to exist in the final result. This is not serious, however, since the presence of Type 2 error only causes slight inefficiency when used to guide an ATPG algorithm. Let us examine the ATPG's use of values containing Type 2 error to see why this is so.

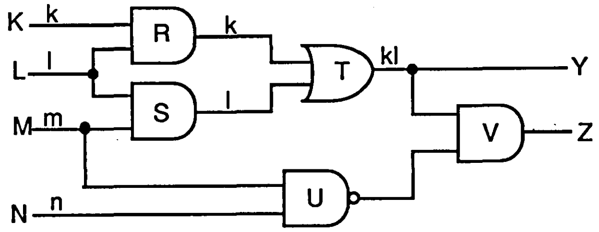


Figure 6. Invisible Type 2 Error

Once again consider the circuit of Figure 6, and assume that the ATPG is trying to set T to a zero. Since both inputs to T must be set to zero, the ATPG will establish its goal of T-0 and attempt to achieve it by trying each input path in turn. Assume that it first tries to set a zero on R. The guidance heuristic will cause it to do this by setting K-0. Since this is a primary input, the backdrive will stop and check if K-0 achieves the goal, which it does not. Therefore, assume that K-0 remains and the ATPG tries along the path from S. To set S-0, the guidance heuristic will indicate that L must be zero. Once again we have reached a primary input and the ATPG will now try to determine if K-0 and L-0 will achieve the goal. This time the goal is satisfied and a way to set T-0 has been found. If the guidance heuristic had indicated that it would be better to backdrive along S to begin with, then the goal could have been satisfied with L-0 alone. Therefore, our measure caused an extra input to be set to achieve the goal. We will return to this point later, but for now it is sufficient to note that the ATPG was not misled by the invisible Type 2 error; the value was pessimistic but still achievable. The fact that a better way existed did not cause backtracking, merely a slight loss of efficiency.

FREE LINES AND HEADLINES

The second level of the heuristic is based on an extension of the concept of free lines and headlines[3]. A free line is a node in a circuit whose predecessors are fanout free. Obviously all primary inputs are free lines. Also, the output of any gate which has only free lines as inputs is also a free line. The free lines of a circuit form trees with only one path from any primary input to any point in the tree. Looking again at Figure 3, we can see that A, B, C, D, E, F, and G are free lines. Also, since all inputs to H and I are free lines, their outputs are also free lines. The outputs of J and K are not free lines since at least one of each of their inputs is a reconvergent fanout branch.

A headline is a free line that enters a reconvergent fanout loop. Input D in Figure 3 is a headline, as are H and I. The importance of headlines can be appreciated when it is noted that outputs of a circuit can be completely characterized on the basis of the values at the headlines of the circuit. Therefore, headlines can be thought of as a reduced but complete set of independent inputs to a circuit. Each headline can be assigned an arbitrary value, independent of all the other headlines, and a complete specification of headline values will completely determine all output values of the circuit.

Since the headlines of a circuit are independent, it is possible to terminate the backdrive process of ATPG at headlines with complete confidence that the value assigned can later be justified, regardless of any other values assigned later to another headline. This can reduce the computation in ATPG significantly if the headline "covers" many inputs. Headline H in Figure 3, for example, "covers" inputs A, B, and C. Therefore, once a value has been determined for H, the backdrive process can defer work along this path, thereby saving the cost of assigning values along the input paths of H. Later, after all other tasks have been resolved by the backdrive, these values can be justified by assigning values backward from their inputs without fear of conflict. The larger the cover of a headline, the larger the savings. Unfortunately, the cover of headlines is usually small.

PSEUDO HEADLINES

It is possible to extend the concept of headlines, however, to allow for larger covers[6,7]. Consider the circuit of Figure 3 again, but assume that it is a subcircuit of a much larger circuit. If no fanout branches exist in this subcircuit other than the one shown, then it is possible to consider the output of L as a Pseudo Headline with respect to the part of the circuit driven by L, since L has all the salient properties of a headline with respect to the things it drives. Further, L covers all the inputs shown in Figure 3, a total of seven. The subtree represented by these seven inputs contains 128 possible binary input combinations. Clearly, if an ATPG algorithm is atempting to find a way to justify a goal which requires L to assume a value, and it can stop as soon as L is reached, a great deal of search time is saved. If an efficient way can be found to identify pseudo headlines it could be used to greatly improve the guidance heuristic for an ATPG. We now illustrate such a procedure.

Identifying Constrained Headlines

A process similar to the computation of the C measure mentioned earlier can be used to identify constrained headlines. To do this two pieces of information are needed: (1) which fanouts actually reconverge, and (2) with respect to which outputs do they reconverge. The calculation of the number of reconvergent branches is done by means of a backward pass to propagate output tags as illustrated in Figure 7. Each output is assigned a unique tag; these tags are then propagated backward, computing the tags of each fanin, until the primary inputs are reached. When a fanout stem is encountered the union of all the branch tags is assigned to the stem. It can be seen that this will yield a measure of the number of reconvergent branches at each fanout stem, as well as the outputs with respect to which those fanouts reconverge. Determination of these tags is called RFO analysis and is used in the calculation of the R measure which we will now describe.

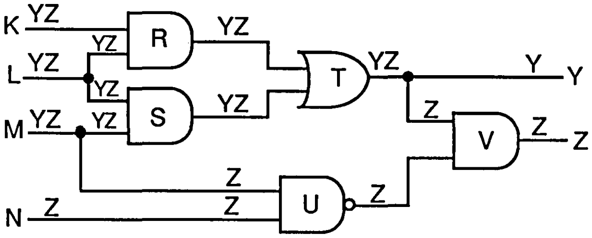


Figure 7. RFO Measure

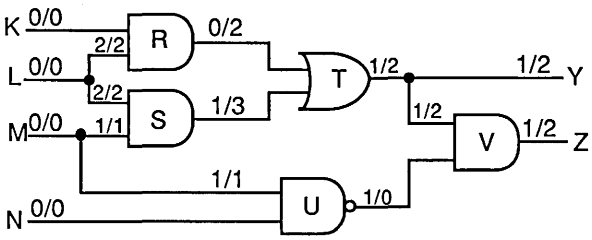


Figure 8. R Measure Computation

Once the RFO analysis is made for each fanout, it is possible to determine the R measure. This process is illustrated in Figure 8. The notation employed is similar to that used earlier where the figure before the / is the zero value and that after the / is the one value. To compute the R measure values, inputs are given an initial assignment of 0/0 instead of the 1/1 of the C measure. The same equations are used to compute the R measure of a gate as the C measure, except when fanouts are encountered. At each fanout, values are assigned to each branch based on the sum of the count on the stem plus the number of occurrences of reconvergent output tags in the pairwise intersection of the branch of interest and every other branch. This yields an approximate measure of the potential conflict that this branch will encounter downstream, and is called the R measure. For example, fanout L in Figure 8 has two branches, which contain output tags for both Y and Z. The stem count of L is 0/0 and the intersection of the branch tags contain 2 entries (See Figure 7). Therefore, each branch is assigned a 2/2 value. This results in a 0/2 value on the output of R.

In circuits with multiple outputs, the computation of the R measure requires that more attention be paid to the value assigned to a particular fanout branch. Consider fanout M in Figure 8; the value of 1/1 is assigned to each branch since only one potential conflict exists downstream, and that with respect only to output Z. This results in a 1/3 assignment to the output of S, based on its input values. The remainder of the calculation uses the same procedure as that for the C measure computation, with appropriate adjustment for the additional output values. As can be seen, the final R measure of a line is proportional to the potential conflict of that line and the number of outputs that the conflict affects.

Once the R measure has been determined for each node in the circuit, it is possible to identify headlines by simply looking for nodes during backdrive that have a 0 R-value. For example, in the circuit of Figure 8 if a value of zero is required by the ATPG backdrive at the output of gate R, the backdrive can immediately defer further work along this path since the R-zero value for that line is zero. Similarly, if backdrive wished to set the output of gate U to a one, it could defer further work since its R-one value is zero. These points in a circuit where one logic value can be independently assigned but the other logic value cannot will be called constrained headlines. In Figure 8, R is a constrained zero headline while U is a constrained one headline. This concept allows us to move the headlines further into the circuit than previous methods. After the constrained headlines have been determined we proceed to identify an additional property which we will call pseudo headlines.

Identifying Pseudo Headlines

A pseudo headline is defined to be a line that can be treated as a headline during backdrive since it can be independently assigned with respect to all other pseudo headlines. Pseudo headlines contain headlines as a subset. The determination of pseudo headlines is similar to the concept of identifying dominators in flow graphs[8,9]. A similar idea has been used to identify points of maximal reconvergence in networks[10]. Since the procedure was originally developed for single output graphs, it must be modified in our case to allow for the calculation of reconvergence with respect to each circuit primary output. The calculation is done by means of a set of counters associated with each fanout that potentially reconverges. One counter per output is used at each fanout. In the circuit of Figure 8, for example, two counters would be assigned to fanout L since two output tags appear there (See also Figure 7). One counter is for output Y and one counter is for Z. Similarly, only one counter is needed at M, since only Z is affected by M's reconvergence. This is known since the intersection of M's branch tags contains only Z. Fanout T, on the other hand needs no counters since none of its branches reconverge.

When the proper number of counters has been assigned each is initialized to the number of reconvergent branches for the particular output. A forward pass is then begun from each reconvergent fanout, in level order, to propagate fanout tags and increment/decrement the counters until the primary outputs are reached or the counters reach zero. When a subsequent fanout is encountered, the counters are incremented by the number of fanout branches less one. When a point of reconvergence is reached, the counters are decremented by the number of inputs that contain the same fanout tag. When all counters associated with a given fanout reach zero, this tag is dropped from further propagation. The gate at which a particular counter reaches zero is the point of complete reconvergence for that fanout with respect to that output. When all such counters at a node reach zero the R value for that node can be set to zero as shown in Figure 9. The propagation of the fanout tags for this process of identifying pseudo headlines is illustrated in Figure 10, and the actual counter manipulation is shown in Table 1. As can be seen, gate T is the point of complete reconvergence for L with respect to Y and Z, while gate V is the same for M with respect to Z.

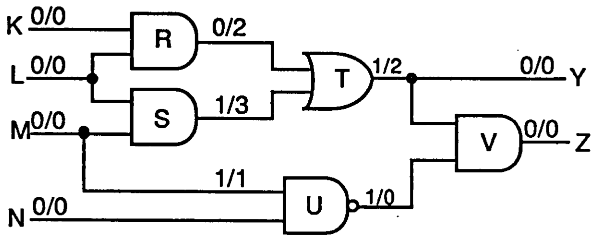


Figure 9. R Measure Adjustment

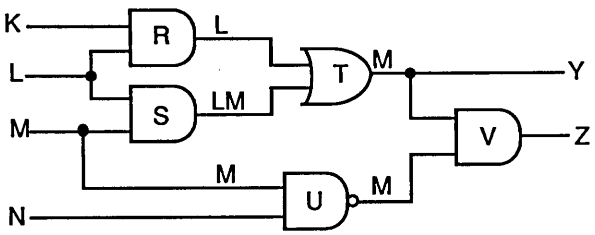


Figure 10. Fanout Tag Propagation

TABLE 1. Pseudo Headline Calculation

STEP	GATE	L-Y	L-Z	M-Z
Initialize	-	2	2	2
1	R	2	2	2
2	S	2	2	2
3	T	0	0	2
4	U	-	-	2
5	V	-	-	0

Once the points of reconvergence are found for each output it is possible to determine if a node is a pseudo headline. To be a pseudo headline requires that the node be a point of complete reconvergence for all fanouts with respect to all outputs of interest. For example, in Figure 10, if the backdrive wished to know whether gate T is a pseudo headline, it needs to know if the backdrive arrived at T along the path from Y or Z, since T is a point of complete reconvergence for all fanouts with respect to Y but not with respect to Z. Thus, T with respect to Y is a pseudo headline, while V with respect to Z is a pseudo headline. When a node is found which is a pseudo headline with respect to all outputs, its R-value can be set to 0 for both zero and one guidance since that node must be independent of all other headlines and pseudo headlines that might be encountered in backdrive with respect to any possible backdrive path.

CHOOSING ASSUMPTIONS

So far we have deferred discussion of the actual method of choosing which assumptions to propagate. Now that the C and R measures have been explained it is possible to state the rules for such choices. As we do, an attempt will be made to justify each rule on the basis of the effect on ATPG backdrive guidance. Obviously, since headlines are completely independent nodes, they should be chosen over any other node when a choice is available. Similarly, pseudo headlines that are completely reconverged with respect to all outputs are almost as good as true headlines and should be given high priority. Also, constrained headlines should be chosen if no headline is available since they are an extension of a headline or pseudo headline. But what if a pseudo headline is not completely reconverged with respect to all outputs? In this case if a pseudo headline can be found that is reconverged with respect to a particular path, then that one should be chosen. If no such pseudo headline can be found, then the pseudo headline with the fewest unreconverged outputs should be chosen on the assumption that it represents the smallest potential for conflict downstream. Finally, if ties result from the rules as given, they are resolved on the basis of the C measure.

Therefore, the priority of rules is:

1. Choose a true headline.
2. Choose a pseudo headline completely reconverged with respect to all outputs.
3. Choose a constrained headline for the value desired.
4. Choose a pseudo headline with smallest unreconverged output list.
5. Choose a node with smallest output list.
6. Choose a node with smallest C measure.

(Ties are broken on the basis of the C measure)

It should be noted that rules 1, 2 and 3 all cause a node with an R value of zero to be chosen while rule 4 chooses a node that is a pseudo headline with respect to the largest number of outputs and rule 5 chooses a node with the smallest number of reconvergent output paths. If all of these rules produce a tie, the C measure is used to break the tie by choosing the node with the smallest C measure.

RESULTS

The heuristic described has been programmed on a Symbolics 3600 in conjunction with an ATPG system using a modified version of the FAN algorithm. While the performance of a heuristic can only be truly measured over a large number of examples, preliminary results with several combinational circuits indicates significant improvement over the traditional controllability guided ATPG. The specific circuits used for this analysis include the 74181 ALU and seven of the circuits proposed at ISCAS 85 as benchmarks for ATPG. The characteristics of these example circuits are given in Table 2. The results of these experiments as well as theoretical analysis indicate that the two-level heuristic becomes more efficient as the size of the circuit grows since the depth of the search stack and thereby the number of remade decisions at each backtrack grows exponentially with circuit size.

TABLE 2. Circuit Characteristics

#	CKT	GATES	PI's	PO's	FAULTS
1	74181	63	14	8	212
2	C432	160	36	7	524
3	C499	202	41	32	758
4	C880	383	60	26	942
5	C1355	546	41	32	1574
6	C1908	880	33	25	1879
7	C3540	1669	50	22	3428
8	C7552	3512	207	108	7550

The results of using the new heuristic to guide ATPG are given in Table 3. While it should be stressed that these results are not completely representative since they represent a small sample of circuits, they are interesting. The number of backtracks is probably the better indicator of the value of the heuristic, but normalized run times on the 3600 are given for comparison as well. The run times should not be taken too literally, however, since overhead such as garbage collection and paging greatly influences these figures, while the number of backtracks is machine independent.

TABLE 3. Heuristic Performance

	C/O MEASURE		C/R MEASURE	
#	BKTRKS	TIME	BKTRKS	TIME
1	167	2.1	78	1
2	355	1.7	126	1
3	567	2.7	178	1
4	714	3.1	223	1
5	1283	3.7	327	1
6	1526	3.8	390	1
7	2774	4.1	693	1
8	6374	4.2	1613	1

REFERENCES

1. J.P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," IBM Journal of Research and Development, Vol. 10, pp. 278-291, July 1966.

2. P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," IEEE Trans. on Computers, Vol. C-30, March 1981, pp. 215-222.

3. H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," IEEE Trans. on Computers, Vol. C-32, December 1983, pp. 1137-1144.

4. F. Brglez, P. Pownall, and R. Hum, "Applications of Testability Analysis: From ATPG to Critical Path Tracing," Proc. 1984 Test Conf., pp. 705-712, October 1984.

5. L.H. Goldstein, "Controllability/Observability Analysis of Digital Circuits," IEEE Trans. on Circuits and Systems, Vol. CAS-26, September 1979, pp. 685-691.

6. M. Abramovici, J.J. Kullikowski, P.R. Menon, and D.T. Miller, "Test Generation in LAMP2: Concepts and Algorithms," Proc. 1985 Test Conf., pp. 49-56, November 1985.

7. V.D. Agrawal, S.C. Seth and C.C. Chuang, "Probabilistically Guided Test Generation," Proc. of ISCAS 85, pp. 687-690, June 1985.

8. R. Tarjan, "Finding Dominators in Directed Graphs," SIAM Journal of Computing, Vol. 3, pp. 62-89, 1974.

9. T. Lengauer and R.E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," ACM Trans. on Prog. Lang. and Systems, Vol. 1, No. 1, July 1979, pp. 121-141.

10. K.S. Hwang and M.R. Mercer, "Derivation and Refinement of Fanout Constraints to Generate Tests in Combinational Logic Circuits," Proc. of ICCAD 85, pp. 10-12, November 1985.

AUTOMATIC INTRA-DEVICE PIN & ELEMENT REASSIGNMENT (AIDPER) ALGORITHM

H. Alan Hershey and Tunde A. Onitiri*

AT&T Bell Laboratories
Crawfords Corner Road
Holmdel, New Jersey 07733

ABSTRACT

The completion rate of printed circuit board (PCB) routing depends on the complexity of net interconnections presented to the router. The PCB layout functions before routing, namely, partitioning and placement, and the order that they are performed help simplify this problem. We took a constructive approach, where partitioning is divided into two functions: the assignment of elements to parts before placement; and repartitioning of elements within a part (or intra-device) following placement. This paper focuses on the intra-device reassignment and it describes our AIDPER algorithm for achieving this function. The innovation here is that AIDPER ensures not only that each element within a part is reassigned to its optimum position, but also that it constraints are satisfied. The algorithm has been incorporated in the IDS Design Station, AT&T Bell Laboratories internal PCB CAD system, and accrues these benefits: emphasizes horizontal and vertical connection, forces nesting of long connections, reduces cross-overs, and increases intra-device connections. The illustrative problem presented, shows the contributions of AIDPER algorithm in simplifying interconnection problems.

1. INTRODUCTION

The ability to obtain higher completion rate of printed circuit board (PCB) routing depends on the complexity of net interconnections that are presented to the router. The PCB layout functions before routing are partitioning and placement. These functions help simplify the interconnection problem, and the order that they are done is important. At AT&T Bell Laboratories, we have taken a constructive approach in the development of these functions for our internal PCB CAD system, IDS Design Station. In the approach, we divided partitioning into two functions: the assignment of elements to part instances before placement; and repartitioning of elements to positions within a part instance (i.e., intra-device reassignment) following placement. As a result, we create synergism among element assignment, placement, and repartitioning functions.

This paper focuses on the intra-device reassignment and it describes the AIDPER algorithm developed for this function. The AIDPER algorithm consists of two assignment models: the classical linear assignment model, applied to reassign elements, and a directional assignment model, based on sorting routines, to reassign pins of an element. The innovation of AIDPER algorithm is that it accounts for the positional constraints imposed on the elements and their associated pins; and it considers the need for elements to co-exist based on the interconnection requirements.

2. BACKGROUND

Several algorithms have been developed for element (or gate) assignment[1], and[2], and for pin assignment[3],[4],[5]. It is not our intent to dissect the algorithms developed by the above authors, rather, to briefly review a couple of them, to introduce the concepts used in our AIDPER algorithm.

Leah Mory-Ranch developed a pin assignment algorithm[3] that is invoked after placement. The objective of Mory-Ranch's algorithm is to reduce the amount of wire cross-overs. It is deficient in two areas. First, the algorithm does not account for the positional constraints imposed on the elements and their associated pins. Second, Mory-Ranch's algorithm does not consider the need for elements to co-exist based on interconnection requirements.

Ikuo Nishioka et al developed a gate assignment algorithm[1] used before placement, and a pin assignment algorithm that is invoked after placement. Referring to their pin assignment algorithm, it is an application of the linear assignment model. The shortcomings of their algorithm are the same as that of Leah Mory-Ranch. In our approach (i.e., AIDPER algorithm), we not only addressed the above pin assignment shortcomings, we also reassign elements within a part.

3. AIDPER ALGORITHM

The automatic intra-device pin and element reassignment (AIDPER) algorithm is a combination of two models: element reassignment which is an application of the linear assignment algorithm, and pin reassignment that is a directional assignment model based on sorting routines. The assignment algorithm and its intricacies have been fully addressed in[6,7,8,9,10]. For a thorough understanding of the modified version of the linear assignment algorithm, let us briefly review its application to the **machine assignment problem.** This will show our departure from the classic linear assignment algorithm. Consider the assignment of m jobs (or workers) to n machines. A job $i(=1,2,\ldots,m)$ when assigned to machine $j(=1,2\ldots,n)$ incurs a cost C_{ij}. The assignment **cost matrix** shown in Table 1 provides the general cost of processing job i on machine j. The objective, then, is to assign the jobs to the machines (one job per machine) to minimize total cost.

* Now with AT&T Information Systems, 60 Columbia Turnpike, Morristown, New Jersey 07960
Telephone (201) 829-7294

CH2345-7/86/0000/0848$01.00 © 1986 IEEE

		Machine					
		1	2	.	.	.	n
	1	C_{11}	C_{12}	.	.	.	C_{1n}
	2	C_{21}	C_{22}	.	.	.	C_{2n}
Job	3	C_{31}	C_{32}	.	.	.	C_{3n}
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	m	C_{m1}	C_{m2}	.	.	.	C_{mn}

Table 1: Cost Matrix

The linear assignment model is mathematically expressed as follows:

Let X_{ij} = 1, if job i is assigned to machine j
= 0, otherwise.

Objective function:

$$\textbf{minimize} \quad X_0 = \sum_{i}^{m}\sum_{j}^{n} C_{ij}X_{ij}$$

subject to:

$$\sum_{i} X_{ij} = 1, \quad j = 1,2,\ldots,n$$

$$\sum_{j} X_{ij} \leq 1, \quad i = 1,2,\ldots,m$$

$$X_{ij} = 1 \text{ or } 0$$

$$(n \geq m)$$

These constraints will assure that each machine is assigned no more than one job and all jobs will be assigned. They also require that the number of jobs not exceed the number of machines for a feasible solution.

Now consider the assignment problem in Table 2, with three jobs and three machines.

		Machine		
		1	2	3
	1	5	8	10
Job	2	6	4	9
	3	17	13	11

Table 2: Cost Matrix

The optimal dual solution is the X_{ij} path where the reduced costs entries ($\bar{C}_{ij}$) all equal zero. Our contrived example is simple with an optimum solution $C_{11} + C_{22} + C_{33} = 20$ as depicted in Table 3. However, applying the linear assignment algorithm to reassign elements and their pins within a part is not that simple. The difficulties arise in constructing the assignment **cost matrix** and in determining the **zero path** for the optimum solution. The difficulties are attributed to the positional constraints applied to the part, and its elements with their associated pins.

		Machine		
		1	2	3
	1	0	3	5
Job	2	2	0	5
	3	6	2	0

Table 3: Reduced Cost Matrix

We developed the AIDPER algorithm to emphasize the intra-device reassignment of elements and pins to achieve as many horizontal and vertical connections as possible. Long connections are forced to nest with shorter and/or assigned connections. As a result, AIDPER depends on the knowledge of the direction of each span (or edge) leaving a part. The AIDPER algorithm uses a "cellular" direction function for determining the spans' directions. Referring to Figure 1, with a part at the center (i.e., direction 9), a span will connect in one of nine directions. Note that direction '9' implies a connection within a part (i.e., intra-connection).

We developed the following mathematical expression for determining the x- and y-directions of each span leaving a part:

$$CELL_x = (X_t - X_c)\,\frac{|X_t - X_c| + 0.5\delta x}{\delta x}$$

where $CELL_x$ = cell count in x-direction

X_t = x-coord of target node

X_c = x-coord of part body center

δx = cell size (i.e., mesh) in x-direction

Note:

a) Span length = $CELL_x + CELL_y$

b) Span direction = $(X_t - X_c) + (Y_t - Y_c)$

3.1 ELEMENT REASSIGNMENT

Although the AIDPER algorithm is an applied classical linear assignment model, the development of its **cost matrix**, and the determination of the optimum assignment differ from the classical approach.

3.1.1 AIDPER Cost Matrix. The cost of reassigning element i to position j is the sum of two cost functions, namely, constraint and positional. Mathematically:

If S_{ij} = cost of spare element i in position j.

F_{ij} = cost of fixed element i in position j.

I_{ijk} = cost of connection k of element i in position j.

For example, an intra-connection between an input and an output pin of two elements, within a part, may require that the pins of these two elements be adjacent or opposite to one another. Figure 2 depicts the hierarchy and the priorities used by AIDPER to process interconnections.

Z_{ijst} = zone cost of span t, pin s, element i in position j.

The penalty for a span not being in the ideal zone (or position). Note that there are four zones, generated by dividing the region around a part, about its center, into four quadrants.

M_{ijst} = length cost span t, pin s, element i in position j.

The distance penalty of a span, determined by its cell counts.

D_{ijst} = direction cost of span t, pin s, of element i in position j.

The nine span directions (Figure 1) are weighted to penalize connections to the immediate regions that cannot be achieved in one wiring track, and/or penalize long connections that cannot nest to existing ones.

CON_{ij} = constraint cost of element i in position j.

Related, but independent to the constraint cost is the formulation of a **co-existence matrix** for specifying co-existence requirement(s) amongst elements. For example, if an input pin element y has an intra-connection to an output pin of element z, for optimum reassignment, it may be required to assign the two elements such that their pins are adjacent to one another for horizontal connection. Therefore, if element y is in position '1' then element z must be (say) in position '2' to satisfy co-existence requirement. And this data is reflected in the **co-existence matrix** accordingly.

POS_{ij} = positional cost of element i in position j.

The **cost matrix** is developed by first seeding it with the constraint costs, i.e.,

$$CON_{ij} = S_{ij} + F_{ij} + \sum_{k} I_{ijk}$$

The positional costs are then added to the **cost matrix**, i.e.,

$$POS_{ij} = \sum_{ijs} \sum_{ijt} (Z_{ijst} + M_{ijst} + D_{ijst})$$

3.1.2 AIDPER Optimum Assignment. From the **cost matrix** we have to determine the optimum reassignment of the elements by minimizing the objective function:

$$EFOM_o = \sum_{i} \sum_{j} X_{ij} \sum_{ijs} \sum_{ijt} (Z_{ijst} + M_{ijst} + D_{ijst})$$

$$+ \sum_{i} \sum_{j} (S_{ij} + F_{ij} + \sum_{k} I_{ijk}) X_{ij}$$

subject to:

$$\sum_{i} X_{ij} = 1, \quad j = 1,2,\ldots,n$$

$$\sum_{j} X_{ij} = 1, \quad i = 1,2,\ldots,m$$

$$X_{ij} = 1 \text{ or } 0$$

$$(n = m)$$

These constraints will assure that each element is assigned to one and only one position and the number of elements is equal to the number of positions.

In the classical linear assignment algorithm, the optimal dual solution is when the reduced costs of the basic variables is zero. In contrast, our algorithm applies the co-existence requirements (or *co-existence matrix*) to the reduced **cost matrix** to determine the minimum reassignment cost. The **co-existence matrix** ensures that not only should each element be reassigned to its optimum position, but also that its constraints are satisfied. This matrix is necessary to properly handle the intra-device connections; particularly, the adjacent I/O condition with no external leads continuing the net.

3.2 PIN REASSIGNMENT

The objective here is to constructively refine the interconnections associated with pins from the element reassignment. The pin reassignment model is a series of four sorting routines that results in reassigning connections to pins to reduce cross-overs, and to promote connection nesting. First, the pins are sorted based on their x-coordinates and y-coordinates for x-oriented and y-oriented parts respectively. Second, the pin connections are sorted by their direction priorities. This sort ensures that spans in directions '1' and '5' (see Figure 1) are at the extreme *LEFT* and *RIGHT* respectively. And the remaining spans' directions are between these extremes. Third, the pin connections in one direction are sorted by span length (in cell counts). This sorting routine also helps break ties in spans of the same direction. Lastly, the spans with the same direction and same span length are further sorted by the opposing pin location criterion which allows the reassignment of element pins that are in opposing sides of a part.

4. AIDPER FLOW

The AIDPER flow consists of three phases, namely, preparation, point source, and refinement. This flow helps simplify the net interconnection problem to achieve a higher percentage of final automatic routes. We will use an illustrative problem depicted in Appendix A to present AIDPER flow and its results.

4.1 PREPARATION PHASE

The preparation phase formulates the reassignment problem of interest. This involves: 1) scope definition, i.e., the entire PCB, a region of the PCB, specific parts within a region, or a single part; 2) determination of each part's relative position within a scope; 3) specification of each placed part's properties, i.e., is the part fixed, or are its terminals and/or elements fixed; and 4) determination of the order which the parts are to be processed. The part order is based on the degrees of freedom for each part and are processed most constrained to least constrained.

For our illustrative problem, we are processing a single Quad 2-Input NAND gate, with swappable elements and input pins, see Figure A-1.

4.2 POINT SOURCE PHASE

The objective of this phase is to process each part as a point source, where all the net interconnections emanating from a part are considered to be from the part's body center. The point source phase is necessary since we must assume that the element and pin assignment have been arbitrary to this stage of the layout process (i.e., element assignment assigned an element to a part instance; but not to a particular position within the part, and the placement algorithm treated the part as a point source). The flow of the point source phase is as follows:

1. Determine the spanning tree of all the nets in the realization scope.
2. Based on the order which the parts are to be processed, derived in the preparation phase, each part is processed as follows:
   - Determine the direction(s) of all the spans leaving a pin of a part. Use the "cellular" directional function described in the previous section.

     Following our illustrative problem, the direction of the spans are shown in Figure A-1.
   - Reassign the elements within a part by applying the AIDPER algorithm. Note that both the elements and their associated pins are reassigned at this stage.

     Continuing with the problem, Figure A-2a shows element assignment only, with no regard to element positional constraints and co-existence requirements. Figure A-2b depicts the advantage of AIDPER's approach.
   - Reassign the pins of an element for refinement using the pin reassignment algorithm described in the previous section.

     Figure A-3a illustrates only pin reassignment condition; without element reassignment, and no regard to positional constraints and co-existence requirements. On the other hand, Figure A-3b shows further simplification of the problem when AIDPER's pin reassignment is done in addition to element reassignment.
   - Store the results of the point source phase for use in the next phase. Note that the point source phase has reformulated the routing problem by performing a crude improvement of the interconnections.

4.3 REFINEMENT PHASE

The objective of this phase is to improve on the results of the point source phase. The refinement phase flow is similar to the point source with the following exceptions:

- The exact direction and location of each span associated with a pin is determined. As a result, we can determine whether the direction suggested at the point source phase as RIGHT is actually going to the top of the part on the RIGHT, or the bottom of the target part.

  Following our illustrative problem, Figure A-4 shows the reduced interconnection problem presented to the refinement phase with exact direction of each span. Referring to the span directions in Figure 1, using 'T' and 'B' to designate the top and bottom edges of a part; an exact span direction of (say) '5T' implies connection going to the LEFT and terminating at the TOP edge of the target part.
- The cellular size used for calculating the direction is finer.
- Recalculate the spanning tree on completion of this phase.

  Figure A-5 shows the final simplified interconnection problem that will be presented to the router.

5. FUTURE WORK

Like any system, the automatic intra-device pin and element reassignment algorithm has its limitations. Therefore, the following areas of further work surfaced during the design and implementation of the AIDPER algorithm.

1. The algorithm can only handle components such as DIPs, RES, and CAPs. Therefore, there is need to investigate how the algorithm can be extended to process surface mounted components, connectors and bus bars.
2. Finally, future work is needed for the automatic reassignment of pins and/or elements of parts considering internal connections.

6. CONCLUSIONS

A description of the algorithm used for automatic pin and element reassignment function that exists in the AT&T Bell Laboratories internal PCB CAD system, IDS Design Station, has been presented. The AIDPER algorithm provides the following benefits:

- Improves the nesting of connections, thus, promoting more row connections that is particularly beneficial when row routers are employed.
- Significantly reduces cross-over of nets, thus, reducing the number of vias for interconnections.
- Improves intra-device connections, while considering inter-device connections.

As demonstrated in Appendix A the algorithm is producing expected results and it supports the synergism among element assignment, placement, and repartitioning functions.

REFERENCES

[1] Ikuo Nishioka, Takuji Kurimoto, Seiji Yamamoto, Toru Chiba, Isao Shirakawa and Hiroshi Ozaki, "An Approach to Gate Assignment and Module Placement for Printed Wiring Boards," *IEEE Transaction On Computers*, Vol. C-29, No. 8, August 1980, pp.681-688.

[2] L. Mah, and L. Steinberg, "Techniques of Gate Assignment," *Proceedings of the Ninth Design Automation Workshop*, June 1972, pp. 63-67.

[3] Leah Mory-Ranch, "Pin Assignment on a Printed Circuit Board," *The Fifteenth Design Automation Conference Proceedings*, June 1978, pp. 70-73.

[4] N. L. Koren, "Pin Assignment in Automatic Printed Circuit Board Design," *Proceedings of the Ninth Design Automation Workshop*, June 1972, pp. 72-79.

[5] C. S. Hing, "Pin Assignment of Circuit Cards and the Routability of Multilayer Printed Wiring Backplanes," *Proceeding of the Tenth Design Automation Workshop*, June 1973, pp. 33-43.

[6] E. A. Dinic and M. A. Kronrod, "An Algorithm for the Solution of the Assignment Problem," *Soviet Math. Dokl.*, 10, 1969, pp. 1324-1326.

[7] M. Malek-Zavarei, "Application of Graph Theory to the Solution of a Nonlinear Optimal Assignment Problem," *The Bell System Technical Journal*, Vol. 61, No. 8, October 1982, pp. 1863-1870.

[8] R. Silver, "An Algorithm for the Assignment Problem," *Commun. Assoc. Comput. Mach.* 3, 1960, pp. 603-606.

[9] Hamdy A. Taha, *Operations Research An Introduction*, Macmillam Publishing Co., Inc., New York, pp. 138.

[10] Sheldon B. Akers, "On The Use of The Linear Assignment Algorithm in Module Placement," *Proceedings of the 18th. Design Automation Conference*, 1981, pp. 137-144.

4	3	2
5	9	1
6	7	8

Figure 1: Span Directions

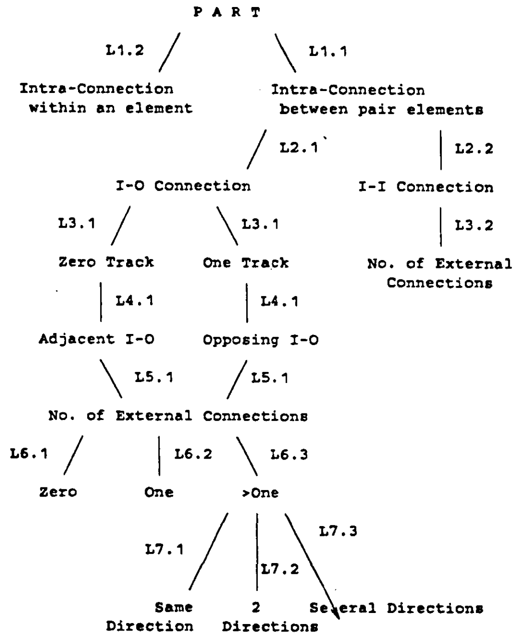
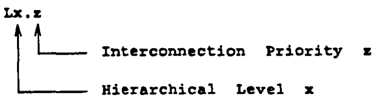


Figure 2: Hierarchy and Priorities of Interconnections
Key:



APPENDIX A: AIDPER Illustrative Problem

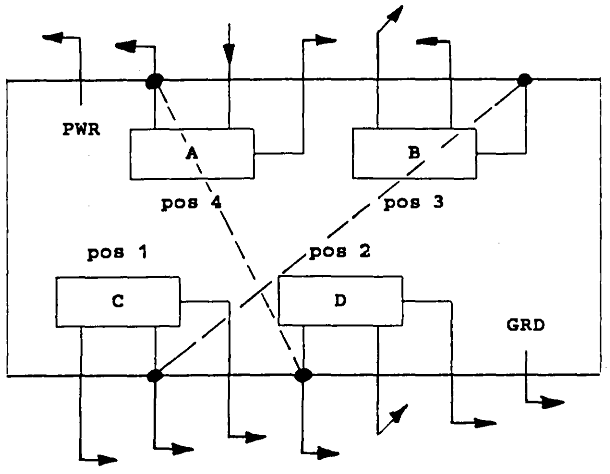


Figure A-1: Original Interconnections With Span Directions

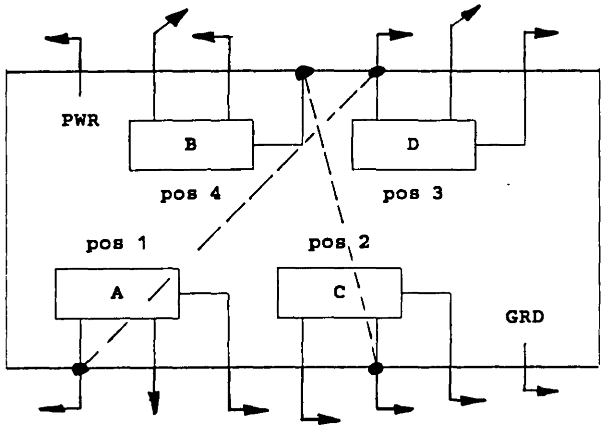


Figure A-2a: Element Reassignment Only, Without Positional Constraints and Co-existence Requirements

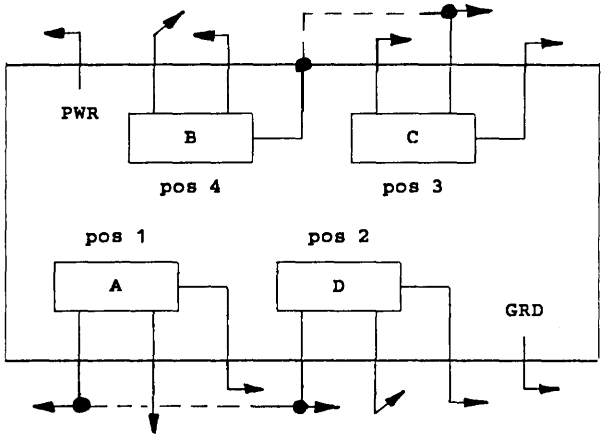


Figure A-2b: AIDPER's Point Source Element Reassignment

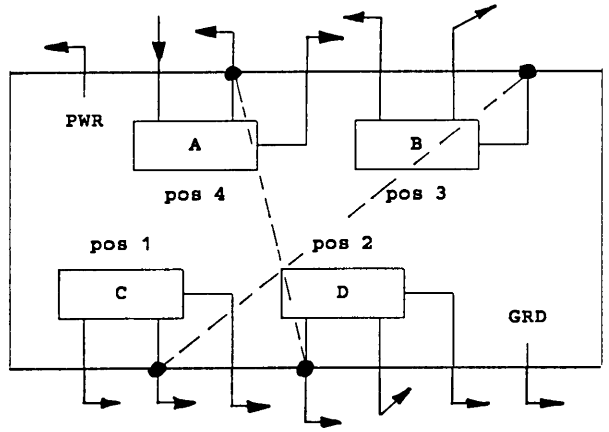


Figure A-3a: Pin Reassignment Only Without Element Reassignment, and No Regard to Positional Constraints and Co-existence Requirements

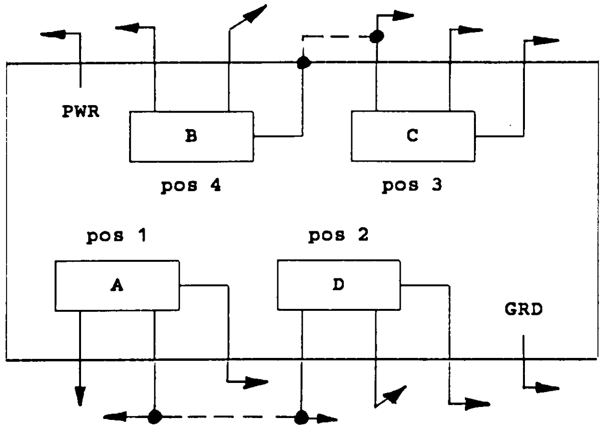


Figure A-3b: AIDPER's Point Source Pin Reassignment After Element Reassignment

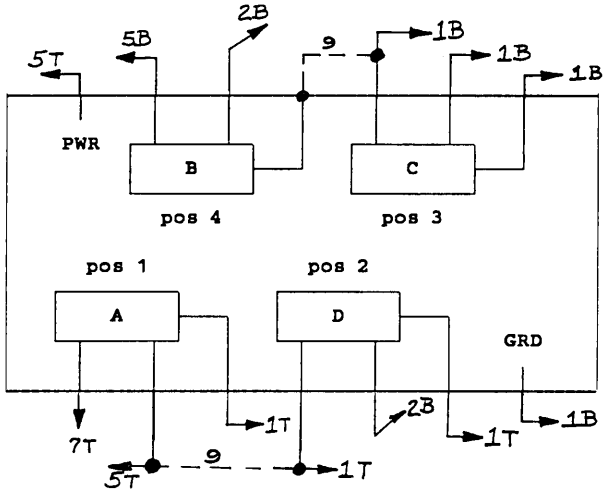


Figure A-4: AIDPER's Refinement Phase Interconnection Problem With Exact Directions of Spans

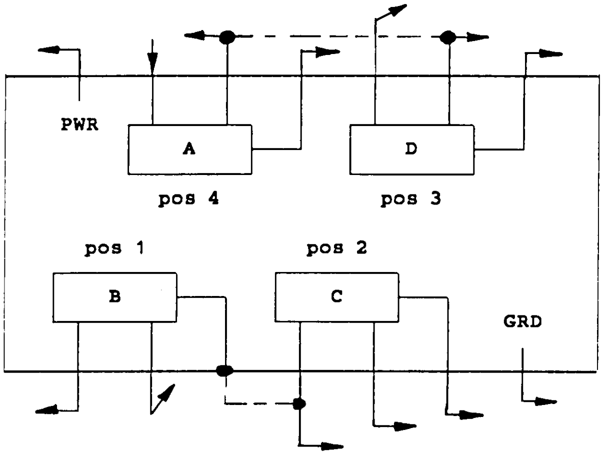


Figure A-5: Final Simplified Interconnection Problem.

A Knowledge Based TDM Selection System

Xi-an Zhu and Melvin A. Breuer[1]

Department of Electrical Engineering-Systems
University of Southern California, Los Angeles, CA 90089-0781

Abstract

In designing a testable VLSI circuit, numerous testable design methodologies (TDMs) can be used, such as LSSD and BILBO. A designer is thus faced with the problem of selecting the appropriate TDM to match his goals and constraints. One way to aid the designer in this selection process is to employ a knowledge based system. In this paper we will describe a prototype of a knowledge based consultant system for TDM selection, called PLA-TSS (Programmable Logic Array TDM Selection System), which emulates a human expert in assisting a designer in selecting a suitable TDM for a given PLA and a set of requirements. Key factors affecting selection are identified. A *knowledge base* containing the TDMs for PLAs has been constructed, where TDMs are represented by frames. Based on this knowledge, *ramification analysis* is carried out, which reveals the relationship among requirements and TDM attribute values. A *score function* is defined to evaluate and compare TDMs. A dynamic selection process is controlled by *reason directed backtracking*. The principles used in PLA-TSS are also applicable to other selection problems of a similar nature.

1 Introduction

In order to solve the VLSI test problem, many testable design methodologies (TDMs) for different type of digital circuits have been developed[1, 2, 3]. As a typical example, we will mainly consider programmable logic arrays (PLAs). More than 30 TDMs for PLAs have been proposed in the last decade. An exclusive survey can be found in[4]. Note that in making a circuit testable under given design constraints, a designer often employ some well known technique for design for testability rather than create his own TDM. The ever-growing number of TDMs for PLAs introduces a new problem to PLA designers and users who want to make a PLA testable: how to select the most suitable TDM for a given PLA and application? Intensive study concerning the TDMs shows that they have multiple and conflicting attributes. The performance of TDMs vary with a PLA's parameters. No TDM is absolutely better than others in every aspect for all PLAs and requirements. Simply applying one TDM to all PLAs will sometimes result in poor designs. Therefore in order to obtain a good testable design, dynamic selection of TDMs for specific circuit and application is necessary. This kind selection problem will soon confront most digital system designers, and the genetic version of this problem can be found in everyday life.

1.1 Main factors affecting selection

Usually a selection problem involves a selector, a receiver and a selection domain.

The selector. A **selector** is an element which initiates and controls a selection. In TDM selection, the selector may be a designer or a synthesis program. The main functions of the selector are to specify a set of requirements for the selection, search for a solution in the domain, and either decide which alternative to select or end the selection process with no solution. A selector's personal preference and knowledge about the solution space greatly affects the result of selection.

The receiver. A **receiver** is the element for which the selection is made. In TDM selection the receiver is a digital circuit. A circuit influences selection because it may partially determine the goodness of the TDMs, since many attribute values of TDMs depend on a circuit's size, aspect ratio or personality.

The domain of selection. The collection of alternatives from which a choice can be made is called the *domain* D of the selection. The domain for TDM selection is a set of available TDMs. In addition to the large number of alternatives, the size of D is usually not fixed. The size of the TDM selection problem for PLAs is about 30 and increasing each year. Without being known by every selector, new TDM may be added to the domain as they are created or made available. Therefore the selector often does not have a complete knowledge of the domain.

The TDM domain is characterized by a set of attributes which include the testability characteristics, effects on the original design, requirements on test environment, and design cost. The *dimension of the solution space* refers to the number of attributes of the domain.

Let ATT $= (att_1, att_2, ..., att_n)$ be a set of attributes of the TDM domain. Every TDM can be represented in terms of the n attributes. An evaluation vector

$$TDM_i = (v_{i1}, v_{i2}, ..., v_{in})$$

can be derived for every TDM from knowledge about the domain attributes and the TDM, where v_{ij} is the value of TDM_i with respect to att_j, and each v_{ij} is either a constant or a function of the receiver's parameters. For a specific receiver, the value of each v_{ij} can be determined.

Each attribute has a *unit* of measurement, and all values for an attribute use the same unit. Most attributes have incommensurable units, hence their values are not directly comparable. Translating these attribute values into a comparable scale is an important step in making a selection.

[1] This work was supported in part by the National Science Foundation under Grand MCS-8203485, and by the Defense Advanced Research Projects Agency, and monitored by the Office of Naval Research under Contract No. N00014-84-K-0649.

CH2345-7/86/0000/0854$01.00 © 1986 IEEE

In an n-dimensional solution space, if the individual dimensions are mutually independent, a selection problem can be divided into n sub-selection problems in n unidimensional subspaces. Unfortunately, the multiple attributes often interact and conflict with each other. Thus finding a globally optimal solution becomes a hard problem.

Requirements. Requirements represent the criteria for selection in terms of the desired attribute values of the selection domain, which contain both the goals and constraints of selection. The selection process is requirement-driven. There is no reason for a TDM to be good or bad unless a requirement is specified. Since the domain is multi-dimensional, the requirements consists of multiple subrequirements, each for a subdimension. A set of requirements can be represented by a *requirement vector*

$$R = (r_1, r_2, ..., r_n),$$

where r_i is the subrequirement for the ith attribute. In practice, not all attributes need to appear explicitly in R, and a requirement may be single valued or a value range $r_j = [r_{j1}, r_{j2}]$. However in the following we will mainly consider single valued requirements.

Priorities of attributes. In general, attributes of TDMs do not have the same importance to all designers and applications. Priorities among attributes will greatly affect the result of selection. To indicate these priorities, a weight vector

$$W = (w_1, w_2, ... , w_n)$$

should be specified by the designer, where w_j is the relative weight of the jth attribute. The w_i's are non-negative numbers, and $w_i > w_j$ implies that att_i is more important than att_j. The relative value of the weights w_j's can be converted into normalized weights nw_j's using the formula

$$nw_j = \frac{w_j}{\sum_{j=1}^{n} w_j}. \qquad \text{Thus } 0 \leq nw_j \leq 1.$$

1.2 The selection problem and selection process

Given the TDM domain D, an initial requirement vector R, and a circuit C, a TDM_i is called a solution if TDM_i satisfies R. A good solution is a $TDM_i \in D$ such that TDM_i satisfies R, and for any other $TDM_j \in D$ such that TDM_j satisfies R, TDM_i is no worse than TDM_j. The precise definitions of "satisfy" and "worse" require a deep discussion, and will be given in the next section. The selection problem S can be stated as follows: select a $TDM_i \in D$ such that TDM_i satisfies R and TDM_i is a good solution for R and C. If no solution exists, adjust R until a good solution is found and as long as R remains acceptable to the selector. If no such R can be found, no solution exists.

Selection is usually not a simple straightforward task, but rather a dynamic process which involves exploring the solution space, changing goals and constraints (requirements), and backtracking. We call this a selection process. Usually the selector first specifies an initial requirement vector $R_0=(r_{01}, ...,r_{0n})$. Search in the solution space is carried out using these requirements. If a satisfactory solution is found, the selection terminates with success. Otherwise, alternative solutions (if any exists) are examined by the selector. If there is no acceptable solution and the selector wants to continue, one or more of the requirement values must be changed, and the search restarts using the new requirement vector $R_1=(r_{11}, ...,r_{1n})$. This process is repeated until a satisfactory solution is found or the selector decides to quit with no solution.

A selection process is mainly controlled by the selector and constrained by the requirements. There is no deterministic algorithm which guarantees finding the most satisfactory solution when conflicts exist. The general characteristics of a selection process are as follows.

- The result of selection is dependent on the requirements specified by the selector, the characteristics of the receiver, and the alternatives in the domain. There is usually no universal or absolutely optimal solution.

- Requirements are changeable. The initial requirements may often be too difficult to satisfy. However as we frequently experience in shopping, failing to satisfy initial requirements does not mean no selection can be made. A selector can modify the requirements until a satisfactory result is obtained. Intelligent decisions on how to change the requirements is crucial to the efficiency of a selection.

- In a multi-dimensional solution space, satisfying one requirement may invalidate another. Tradeoffs between different requirements are often necessary.

- Complete domain knowledge is required to make the best choice. However most selectors are not experts in the domain of selection. Therefore advice from expert consultants concerning the domain is often needed.

- Human factors, such as a selector's preference, are important in a selection.

These characteristics make the selection problem unique and non-trivial. It is difficult for a designer to make a good selection because of his lack of knowledge about TDMs and limited ability to handle a large amount of complicated information involved in a selection. People usually solve such problems with the help from domain experts. Unfortunately, such experts are not always available. Our goal is to build a knowledge based expert consultant system, as shown in Figure 1, which helps to solve selection problems. By introducing the consultant, the function of the selector is greatly reduced, and a better result can be expected if the knowledge based system is suitably constructed.

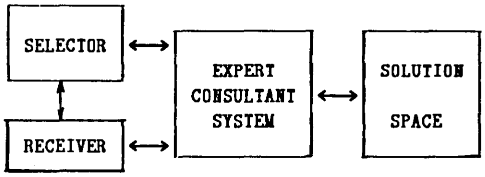


Figure 1: Selection with help of an expert consultant system

Basically, the consultant system has two kinds of knowledge.

1. **Domain specific knowledge** which includes knowledge about the attributes of the selection domain and the properties of the attributes, and knowledge about all alternatives in the domain of selection.

2. **Domain independent knowledge** such as knowledge about the art of controlling a selection process.

Domain specific knowledge is contained in a knowledge base. Domain independent knowledge is incorporated in a controller. A prototype system, PLA-TSS has been built which helps a designer to select a TDM for a PLA. In the following we will discuss the main features of PLA-TSS. In section 2 we briefly describe the knowledge base of PLA-TSS. Then in the next three sections we will elaborate on the major functions of PLA-TSS, namely how to evaluate TDMs, how to reason about the interactions among attribute values, and how to control backtracking. Finally we will give a overview of PLA-TSS.

2 The knowledge base

The knowledge base is an important part of an expert consultant system. The knowledge base should be designed in such a way that it contains all information needed for both the expert and the designer, and it is easily accessible, understandable and changeable.

The knowledge base of PLA-TSS is a knowledge repository which contains all information about the TDM domain. The main body of the knowledge base is the TDMs. It appears that TDMs are most suitably represented by frames. A frame provides a number of *slots* which contain various pieces of information concerning a TDM. The slots correspond to the generic concepts which define the domain. Every TDM in the domain can be characterized by the same set of concepts. The multi-dimensional domain of selection naturally matchs into a frame, in which each attribute corresponds to a slot in the frame, and its value is the entry in the slot which may be a constant or a function of a PLA's parameters. The TDM frame defines a general representation of a TDM. It may not be unique because slots can be organized into different hierarchies. It may not be fixed because new slots can be added to the frame. However, once a frame is defined, all TDMs should be represented in a uniform way. Each particular TDM_i defines a specific *instance* of the general TDM frame. The number of instances is not limited.

For a given PLA, the attribute values of each TDM can be determined. Thus a completely specified evaluation matrix

$$\text{EM} = \begin{array}{c} \begin{array}{ccccc} \text{att}_1 & \cdots & \text{att}_j & \cdots & \text{att}_n \end{array} \\ \left(\begin{array}{ccccc} v_{11} & \cdots & v_{1j} & \cdots & v_{1n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ v_{i1} & \cdots & v_{ij} & \cdots & v_{in} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ v_{m1} & \cdots & v_{mj} & \cdots & v_{mn} \end{array} \right) \end{array} \begin{array}{c} \\ \text{TDM}_1 \\ \cdots \\ \text{TDM}_i \\ \cdots \\ \text{TDM}_m \end{array}$$

can be built in which rows correspond to TDMs and columns correspond to attributes. Each entry v_{ij} is the value of TDM_i which respects to att_j for the given PLA. This evaluation matrix forms the solution space for a specific selection process.

3 Evaluation functions

The general goal of TDM selection is to choose a "good" TDM for a set of given requirements and a given circuit. What is a "good" TDM? Given a set of TDMs with multiple attribute values, how does one judge which is better? To answer these questions, a function for evaluating TDMs in the solution space must be defined.

3.1 Properties of attributes

Since TDMs have multiple attributes, to compare different TDMs we must first define how to compare values for each attribute which corresponds to one subdimension in the multi-dimensional solution space. The ordering relation in each subdimension is dependent on the properties of the corresponding attribute. In the TDM domain there are several different types of attributes.

An attribute is a **simple attribute** if it only takes on single values. Non-simple attributes are called **complex attributes**, whose values are sets which may be a singleton.

A simple attribute may be either numeric or binary. A simple attribute is **numeric** if its values are in the real domain $\mathcal{R}$. Attributes which only take binary values (YES or NO) are called **logical attributes**.

A numeric attribute att_i is an **upper-bound** attribute if the requirement r_i specifies an upper bound for its value. A numeric attribute att_j is a *lower-bound* attribute if the requirement r_j specifies a lower bound for att_j's values.

The value of a logical attribute att_i may be either YES or NO. One value is stronger than the another if the former is more favorable. A logical attribute att_i is an upper-bound attribute, if for att_i YES is stronger than NO. Similarly, a logical attribute att_j is a lower-bound attribute, if for att_j NO is stronger than YES.

A complex attribute att_i is an upper-bound attribute, if, for any requirement r_i, acceptable values of att_i are all subsets of r_i. A complex attribute att_j is a lower-bound attribute, if, for any requirement r_j, acceptable values of att_j contain r_j.

3.2 The comparison function

Knowing the properties of attributes, a partial ordering relation can be defined for each subdimension of the domain which indicates how to compare any two values of the same attribute. In a selection process, another type of comparison is more important, namely the comparison between a value and a requirement for an attribute, because the real goodness of a value is relative to the requirement. A function *compare* is defined which returns the *distance* between a value v_{ij} and a requirement r_j. *Compare* is a complex function due to the fact that attributes of a domain have various properties.

Definition 1: Let R_j be a set of possible requirement values for attribute att_j, and V_j be a set of possible values of att_j which the TDMs may possess. *Compare* is a function from $R_j \times V_j \rightarrow \mathcal{R}$ defined as follows, where $\mathcal{R}$ is the domain of real numbers.

For any $r \in R_j$, $v \in V_j$ and any j,

$$\textit{compare}(r, v) = 0, \text{ if } r = \text{don't care}.$$

Otherwise,

$\textit{compare}(r, v) = r - v$, if att_j is upper bound numeric;

$\textit{compare}(r, v) = v - r$, if att_j is lower bound numeric;

If att_j is upper bound logical,

$$\textit{compare}(r, v) = \begin{cases} -1, \text{ if } r = \text{YES}, v = \text{NO}; \\ 0, \text{ if } r = v; \\ 1, \text{ if } r = \text{NO}, v = \text{YES}; \end{cases}$$

If att_j is lower bound logical,

$$\textit{compare}(r, v) = \begin{cases} -1, \text{ if } r = \text{NO}, v = \text{YES}; \\ 0, \text{ if } r = v; \\ 1, \text{ if } r = \text{YES}, v = \text{NO}; \end{cases}$$

If att_j is upper bound complex,

$$\textit{compare}(r, v) = \begin{cases} -1, \text{ if } r \cap v = \emptyset; \\ 0, \text{ if } r \cap v \subset v; \\ 1, \text{ if } r \cap v = v. \end{cases}$$

If att_j is lower bound complex,

$$\textit{compare}(r, v) = \begin{cases} -1, \text{ if } r \cap v \neq r; \\ 0, \text{ if } r = v; \\ 1, \text{ if } r \subset v. \end{cases}$$

If $compare(r_j, v_{ij}) \geq 0$, we say v_{ij} satisfies r_j. If $x < 0$, v_{ij} does not satisfy r_j. A $TDM_i = (v_{i1}, v_{i2}, ..., v_{in})$ satisfies a set of requirements $(r_1, r_2, ..., r_n)$ if for all $1 \leq j \leq n$, $compare(r_j, v_{ij}) \geq 0$.

3.3 The penalty-credit function

The *compare* function determines the distance between a requirement and a value in a subdimension of the solution space. In many cases, however, not the magnitude of the distance but the effect of the difference dictates the selection. The actual effect may not be proportional to the distance.

A selection process is not simply matching the requirements and attribute values of TDMs. The key consideration in making a selection is how much each attribute value affects the final solution. Usually, if a value v just satisfies a requirement r, it will be accepted. If v is better than r, it may be accepted with extra credits. Such credits may increase with the positive distance between v and r but saturate after some point. If v does not satisfy r, it may still be partially acceptable with some degree of penalty. However after certain point, v may become absolutely unacceptable. To describe the impact of an attribute value with respect to a requirement value, a **penalty-credit function** (PCF) should be established for every attribute.

Definition 2: Let PC_j denote the PCF for att_j. PC_j is a function from $R_j \times V_j$ to $\mathcal{R}$. $PC_j(r_j, x1) > PC_j(r_j, x2)$ if and only if x1 is a more preferable value than x2 for att_j.

Definition of PCFs are completely dependent on the selector and the application. There is no universal PCF that suits every case. However knowledge about the general forms of PCFs can be used to help designers in defining appropriate PCFs. In PLA-TSS, there are three ways for a designer to specify PCFs.

1. Define own PCFs.
2. Modify the standard PCFs.
3. Use the default PCFs.

These methods differ in how to divide the possible values into groups, and what functions are used in each group to produce the desired penalty or credit.

3.3.1 Custom-defined PCFs

The main features of a custom-designed PCF is that the value range of an attribute can be divided into any number of regions, and any meaningful function can be used in each region.

Example 1. Consider a PCF for fault models. Assume there are four fault models: a, b, c and d. If the designer wants faults in class a to be definitely detected, and prefers faults in class b and c to be detected as well, but does not care about faults in class d, the PCF for fault models can be defined as follows.

$$PC_{fm}(r, v) = \begin{cases} 40, & \text{if } \{ a, b, c \} \subseteq v; \\ 30, & \text{if } \{ a, b \} \subseteq v \text{ or } \{ a, c \} \subseteq v; \\ 20, & \text{if } a \in v; \\ -300, & \text{if } \{ b, c \} \subseteq v \text{ and } a \notin v; \\ -500, & \text{if } b \in v \text{ or } c \in v\text{, but } a \notin v; \\ -2000, & \text{if } v \text{ does not contain a or b or c.} \end{cases} \quad (1)$$

Example 2. The PCF for fault coverage can be defined as follows.

$$PC_{fc} = \begin{cases} A + compare(r,v) \times 10, & \text{if } v \geq 97; \\ A + compare(r,v) \times 20, & \text{if } v < 97. \end{cases} \quad (2)$$

Note that a custom defined PCF may be independent of the requirement, although it should imply the requirement. It may include the weight factor in it, since the value of PC(r, v) is arbitrary. This provides a way of non-linear weighting.

3.3.2 Standard PCFs

Since there is usually some commonality in most PCFs, a parameterized standard form of a PCF is defined for each type of attribute. A designer can use his own parameters to modify these standard forms. For example, let us consider numeric attributes. Assume r_j is the requirement for att_j. The designer should specify a *saturation point* (SP) and an *unacceptable point* (UP). The entire value range of att_j is then divided into five segments, namely,

1. the saturation segment consisting of values better than or equal to SP,
2. the credit segment consisting of values from r_j to SP,
3. the acceptance segment defined by r_j,
4. the penalty segment consisting of values from r_j to UP,
5. the unacceptable segment consisting of values worse than or equal to UP.

The designer can then choose the function to be used in each segment, which can be a constant or a function of r_j and values for att_j.

Example 3. Assume SP = 10 and UP = 50. Using the standard PCF for numerical attributes, a PCF for area overhead can be defined as follows.

SEGMENT	VALUE RANGE	FUNCTION USED WITHIN THE SEGMENT
SATURATE	[0, 10]	C
CREDIT	[10, r]	$A \times e^{\ln(C/A) \times compare(r,v)/compare(r,10)}$
ACCEPT	[r, r]	A
PENALTY	[r, 50]	$- [compare(r,v)]^2$
UNACCEPTABLE	> 50	D

C > A > 0, D << 0.

This function is plotted in Figure 2.

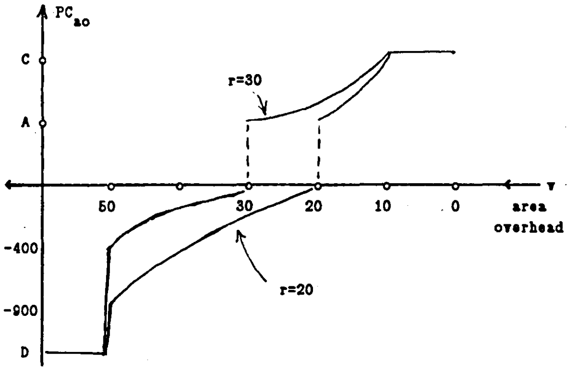


Figure 2: The PCF for area overhead.

3.3.3 Default PCFs

Being an expert consultant, default knowledge is necessary. PLA-TSS has a procedure for automatically defining default PCFs according to the user's requirements. The system defined PCFs are designed to accommodate general situations and be fair to all attributes. For numeric attributes, the default PCF is defined below, where A, B, C and D are constants and can be changed by the user or an expert. As an example the system's default PCF for the requirement "extra I/O pins is 5" is shown in Figure 3.

$$PC_j(r_j, v) = C, \text{ if } compare(r_j, v) > 0.5r_j.$$

$$PC_j(r_j, v) = A + \frac{C-A}{0.5r_j} \times compare(r_j, v),$$
$$\text{if } 0 < compare(r_j, v) \leq 0.5r_j.$$

$$PC_j(r_j, v) = A, \text{ if } compare(r_j, v) = 0.$$

$$PC_j(r_j, v) = B - \frac{B-D}{0.5r_j} \times compare(r_j, v),$$
$$\text{if } 0 > compare(r_j, v) \geq -0.5r_j.$$

$$PC_j(r_j, v) = E,\ E << 0, \text{ if } compare(r_j, v) < -0.5r_j.$$

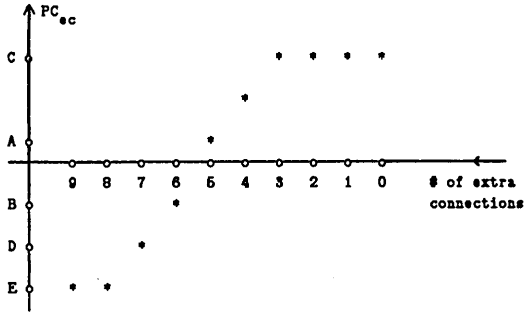


Figure 3: The PCF for extra connections

For logical attributes, there are only four combinations of requirements and values. The differences among PCFs lie in the choices for penalty or credit for each of the four cases. For complex attributes, numerous combinations exist. However they can be classified into three cases, namely a value either exceeds, satisfies or fails a requirement. The default PCFs for logical and complex attributes are defined uniformly as follows.

$PC_j(r_j, v) = D$, if $compare(r_j, v) < 0$.
$PC_j(r_j, v) = A$, if $compare(r_j, v) = 0$.
$PC_j(r_j, v) = C$, if $compare(r_j, v) > 0$.
Here A, C and D ($D < 0$) are the same as for the numeric attributes.

Example 4. The default PCF for self-testing is shown in Table 1. For this case a large credit is given to a value YES when r = NO to increase the attractiveness of self-testing even when not specified as a requirement.

Default PCFs do not imply any priorities among attributes. Priorities are specified via the weight vector. The three methods for defining PCFs provide enough capability and flexibility for a designer to conveniently specify most meaningful PCFs.

requirement	value	credit	penalty
YES	YES	A	
YES	NO		D
NO	YES	C	
NO	NO	A	

Table 1: The default PCF for self-testing

PCFs define the tightness or looseness of the requirements, and play a multiple role in making a selection.

- PCFs convert incommensurable attribute values into a common measure — the resulting penalty-credit on the final solution.

- PCFs translate attribute values of different magnitudes into a comparable scale.

- PCFs specify the fuzziness of the requirements and provide more precise information about the criteria for selection which cannot be represented by the requirements alone.

It is the combination of requirements and PCFs that completely defines the criteria for selection. It should also be noted that in defining the PCFs we assume that each PC_j can be assessed independent of other attribute values.

3.4 The score function

So far we have only considered evaluation of values in individual dimensions. In practice, a solution space is multi-dimensional. Comparisons among TDMs and between TDMs and requirements take place in multiple dimensions. A problem then is how to combine n attributes with different characteristics, meanings and units. Our solution is to first convert each of the n attribute values of a TDM into a numeric value representing the contribution of that attribute to the global solution, then combine the n unitless values into a single numeric *score* representing the overall quality of the TDM. The first problem is solved by the PCFs defined in the last subsection. The second problem is solved using a score function which should have the following properties.

1. For any two TDMs, TDM_i and TDM_k, $score(c_i) \leq score(c_k)$ if and only if $(v_{k1}, v_{k2}, ..., v_{kn})$ is more or equally preferable than $(v_{i1}, v_{i2}, ..., v_{in})$.

2. The score should reflect precedences among attributes. The attributes with higher weights make more positive or negative contribution to the total score.

3. The score should reflect tradeoffs in satisfying different requirements. Each TDM's advantages and disadvantages tend to cancel each other in the score function. The TDMs which offer more weighted credits than penalties should have higher scores.

Using the normalized weights, the PCFs and the compare function, many evaluation functions can be defined to combine multiple features of a TDM into a single score. Two score functions are used in PLA-TSS.

1. The total score is defined by the equation

$$score(TDM_i) = \sum_{j=1}^{N} PC_j(v_{ij}, r_j) \times nw_j.$$

The total score reflects tradeoffs between different features of a TDM so as to represent the overall performance of the TDM.

2. The loss score (LS) is defined by the equation

$$LS(TDM_i) = \sum_{j=1}^{N} \delta(compare(r_j v_{ij})) \times$$
$$[PC_j(v_{ij}, r_j) - PC_j(r_j, r_j)] \times nw_j,$$

where $\delta(x) = 1$ if $x < 0$, otherwise $\delta(x) = 0$.

The loss score of a TDM indicates the degree to which a TDM does not satisfy the requirements. It is the summation of the weighted negative distance between a TDM and the requirement vector. If TDM_i satisfies all requirements, $LS(TDM_i) = 0$, otherwise, $LS(TDM_i) < 0$. The larger the value of $LS(TDM_i)$, the closer the TDM is to the requirements.

These two score functions have different properties. Under the assumption that the goal of selection is to choose a TDM with the best overall performance, the total score is used to compare and rate TDMs. The total score indicates a relative ranking of TDMs. It not only helps identify the "best" TDM, but provides a broad range of choices. In case a designer wants to make a minimal change to the requirements in order to obtain a solution, the loss score will suggest the best TDM.

Example 5. Suppose there are five TDMs and five attributes as shown in Table 2. We will use the PCFs defined in Table 1, equation (1) and (2), and Figures *2* and *3*, respectively. Let A = 20, C = 30, B = -5 and D = -2000. For the given requirements shown in row "Req." of Table 3, and the relative weights given in row "R.W.", the normalized weights (N.W.) and scores are as shown. Each entry corresponding to TDM_i and att_j is the value of $PC_j(r_j, v_{ij})$. The total scores for the TDMs are listed in the last column.

	att_1	att_2	att_3	att_4	att_5
TDMs	self-testing (st)	fault model (fm)	fault coverage (fc)	area overhead (ao)	extra i/o connections (ec)
TDM1	no	{a, b}	99	0	0
TDM2	yes	{a, c}	95	35	2
TDM3	yes	{a,b,c}	100	42	4
TDM4	yes	{a, d}	98	17	7
TDM5	no	{a,b,c,d}	100	300	5

Table 2: A small fully specified evaluation matrix

	self-testing	fault model	fault coverage	area overhead	extra i/o connections	total score
Req.	yes	{a}	97	30	5	
R.W.	3	1	6	8	4	
N.W.	0.125	0.042	0.25	0.333	0.167	
TDM1	-2000	30	40	50	20	-218.75
TDM2*	20	30	-20	-25	20	-6.225
TDM3	20	40	50	-144	15	-28.767
TDM4	20	20	30	26.03	-2000	-314.49
TDM5	30	40	50	-2000	10	-646.40

Table 3: Calculation of scores [I]

These scores show that no TDM completely satisfies the requirements, though TDM2 comes closest. Now if we change the weight of fault coverage and area overhead to 9 and 5, respectively, and re-calculate the scores as shown in the Table 4 (a), the score of TDM3 becomes the largest. This means that if fault coverage is more important than area overhead, TDM3 is the best. If the constraint for extra connections is not very tight, and we change a part of the PCF for extra I/O connections to

$$PC_{ec} = - (B * compare(r,v)), \text{ if } v > 5 ,$$

then the scores are as shown in Table 4 (b). In this case, TDM4 is the best.

TDM	Total score
TDM1	-200
TDM2	-5.6
TDM3*	-4.517
TDM4	-313.996
TDM5	-390.15

(a)

TDM	Total score
TDM1	-200
TDM2	-5.6
TDM3	-4.517
TDM4*	18.334
TDM5	-390.15

(b)

Table 4: Calculation of scores [II]

This example shows how the scores change with weights, requirements and PCFs, and aids in identifying the most suitable TDMs. In general, the better a TDM satisfies a requirement vector, the higher will be its score. Scores are the main criteria used by the consultant system to evaluate a TDM, to sort the TDMs to identify potential solutions, and to select a TDM with optimal overall performance.

4 Ramification analysis

The domain of selection has multiple attributes with various relations existing among them. These interrelations cause conflicts between requirements, which then lead to failures in finding a solution. To be intelligent and efficient in selection, it is important to understand the relations among various attribute values. This is the task of ramification analysis. We developed a systematic method for finding interactions among attribute values in a given evaluation matrix. The basic concept can be illustrated by **ramification trees**.

4.1 Ramification trees

A ramification tree is a knowledge structure for representing relationships among attribute values upon which ramification analysis can be carried out.

Definition 3: Let $T = \{TDM_1, TDM_2, ...\}$ be a set of TDMs where $TDM_i=(v_{i1}, ..., v_{in})$, and (att, v) represent an attribute value pair where v is a value of att. The direct consequence of (att_j, y) on att_p is (att_p, x), if (1) all v_{ip}'s are known, (2) $y \neq$ unknown, and (3)

- att_p is logical, and $\forall$ TDM_i, $TDM_k \in \{TDM_l \mid TDM_l \in T$ and $y <= v_{lj}\}$, $v_{ip} = v_{kp} = x$.

- att_p is numeric, and $x = range\{ v_{ip} \mid TDM_i \in T$ and $y <= v_{ij}\}$, where $range\{n_1,...,n_x\} = [\min\{n_1,...,n_x\}, \max\{n_1,...,n_x\}]$.

- att_p is complex and $x = worst\{ v_{ip} \mid TDM_i \in T$ and $y <= v_{ij}\}$.

Here for numeric and complex attributes x "<=" y if *compare*(x,y) $\geq$ 0, while for logical attributes, "<=" stands for "=". Otherwise, the direct consequence of (att_j, y) on att_p is $(att_p$, unknown).

Definition 4: Given a set of attributes $ATT = \{att_1, ..., att_n\}$ and a set of TDMs T. A ramification bush (R-bush) for an attribute att_j, denoted by $RB(att_j)$, is a two level directed tree in which

- there is a root node att_j which points to all internal nodes;

- there are t $(1 < t \leq k)$ internal nodes labeled (att_j, v_i), for $1 \leq i \leq k$, where k is the number of different known values $v_1, ..., v_k$ which att_j can take, i.e., $\forall\, 1 \leq l \leq k$, $v_l = v_{ij}$ for some $TDM_i \in T$;

- each internal node of the R-bush links to at most n leaf nodes corresponding to the n attributes in ATT;

- each leaf node connecting to an internal node (att_j, v_i) is labeled (att_p, x_{ip}), where (att_p, x_{ip}) is the direct consequence of (att_j, v_i) on att_p. In particular, (att_j, x_{ij}) is a leaf node if and only if $x_{ij} \neq v_i$.

Example 6. For the evaluation matrix in Table 2, the complete R-bush for self-testing is given in Figure 4. The leaf nodes represent the direct consequences of the possible values of self-testing on other attributes.

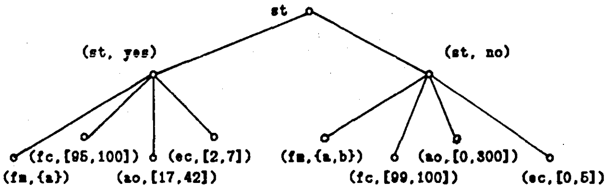


Figure 4: The complete R-bush for self-testing

Definition 5: A conditional R-bush RB(att_j, v) is an incomplete R-bush for att_j which only contains one internal node (att_j, v).

Definition 6: Let RB_1(att_j, v_1), ..., RB_t(att_j, v_t) be distinct conditional R-bushes for att_j. The union of RB_1, ..., RB_t is the R-bush for att_j, which has t internal nodes (att_j, v_1), ..., (att_j, v_t).

Conditional R-bushes are very useful when one is only interested in a particular value of an attribute. A complete R-bush is the union of all conditional R-bushes for the same attribute. Each internal node of an R-bush defines a *branch* of the R-bush. For the R-bush shown in Figure 4, the conditional R-bush RB(st, no) is the right branch of the R-bush defined by the internal node (st, no).

A ramification bush indicates the impact of one attribute's value on others. Individual R-bushes for different attributes can be concatenated together into a *ramification tree* (R-tree) to show the ramification of multiple attribute values on remaining attribute values.

Example 7. Again consider Table 2. Suppose we start with the attribute self-testing (st). Initially the R-tree is the complete R-bush for st as shown in Figure 4. Since the complete R-tree is large, we only expand one branch of the R-tree. First expand the leaf node (fc,[95,100]) under (st,yes) in the left branch. Since under the condition that st = yes, fc has only three possible values 95, 98 and 100, we replace the leaf node (fc,[95,100]) in Figure 4 by a 3-branch R-bush RB = RB(fc,95) ∪ RB(fc,98) ∪ RB(fc,100) which is for the TDM set T1 = {TDM2, TDM3, TDM4} (since only these three TDM satisfies st = YES) and the attribute set {fm, fc, ao, ec} since st has been considered. In the similar way, we continue to expand the leaf node for ec under (fc,98) until no more expansion is possible along alone that branch of the R-tree. A fully expanded branch of the R-tree is shown in Figure 5.

Definition 7: A *stage* of an R-tree is a two-level subtree of the R-tree which forms an R-bush. A *branch* of an R-tree is a subtree of the R-tree that (1) contains the root, (2) consists of no more than one branch of the R-bushes from each stage, and (3) has leaves all of which are leaf nodes of the R-tree.

Every branch of a static R-tree defines a set of consistent requirements. A completely specified requirement vector R = (r_1,..., r_n) can be satisfied if there exists a branch in which there is no node (att_j, x_j) such that x_j does not satisfy r_j, for $1 \leq j \leq n$.

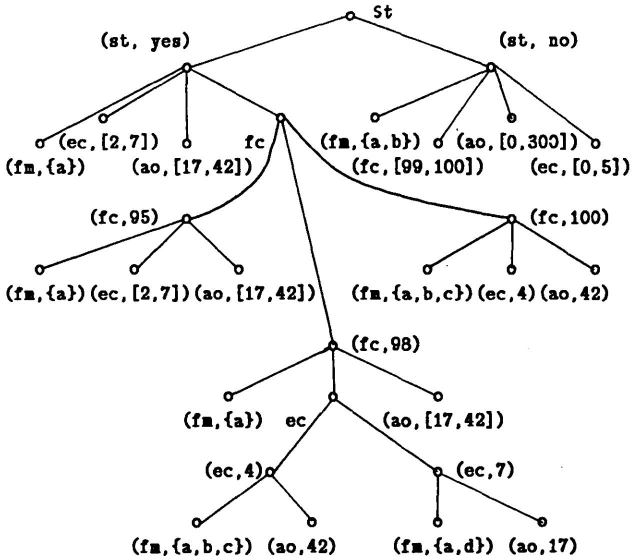


Figure 5: A static R-tree

Static R-trees provide information concerning possible combinations of feasible requirements. In a real selection process, however, only one requirement vector is active at a time, and most parts of the complete R-tree may never be accessed, therefore there is no need to generate a complete R-tree. To represent the useful relations among attribute values in a dynamic selection process, dynamic ramification trees are defined.

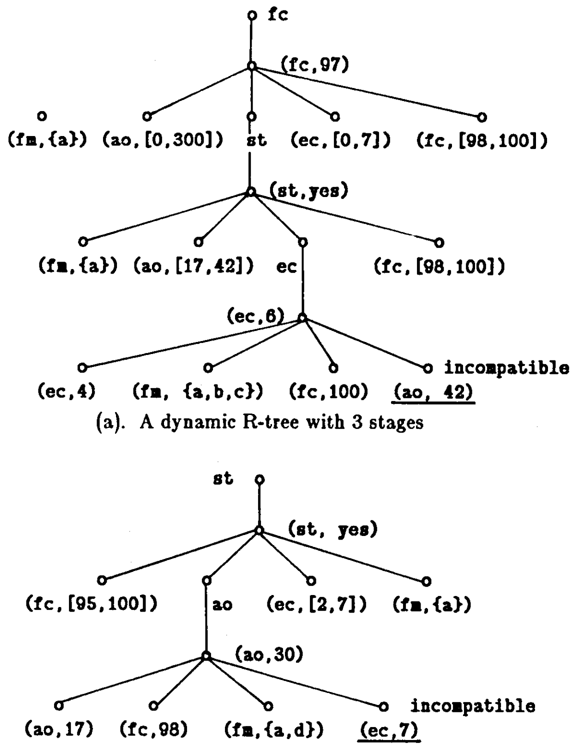
Definition 8: A dynamic R-bush (DRB) for a requirement r_j, is an R-tree having only one internal node (att_j, r_j).

A dynamic ramification tree (DRT) corresponding to a requirement vector R=(r_1, ..., r_n) is a single-branch R-tree consisting of a cascade of DRBs in which internal nodes are in the set {(att_1, r_1), ..., (att_n, r_n)}.

Definition 9: A set of requirements is consistent if there exists a TDM which satisfies it, otherwise the requirements are inconsistent.

Example 8. Consider the evaluation matrix in Table 2. Let the requirement vector be (st, fm, fc, ao, ec) = (yes, {a}, 97, 30, 6). Two different dynamic R-trees are shown in Figure 6. The first stage of the dynamic R-tree in (a) corresponds to a dynamic R-bush DRB(fc, 97). At the third stage the R-tree has a node (ao, 42) which implies that if the requirements fc=97, st=yes and ec=6 are satisfied, area overhead must be 42%. This, however, is incompatible with the requirement ao=30. Both R-trees contain incompatible nodes, since this set of requirements is inconsistent.

The size of a dynamic R-tree is a function of the order in which attributes are selected. Heuristics can be employed to generate dynamic R-trees having different attributes. For example, if satisfying some requirements is important, one can sort the attributes by their weights, and then expand the R-tree using the sorted attribute order. Thus the ramifications of the more important requirements will be processed first, and the less important requirements may be adjusted using the information from the dynamic R-tree. Several procedures for constructing various R-trees for any given evaluation matrix have been defined[4]. Next, we will discuss some applications of ramification trees.



(a). A dynamic R-tree with 3 stages

(b). A dynamic R-tree with 2 stages

Figure 6: Two dynamic R-trees for a requirement vector

4.2 Ramification analysis

One problem in a selection process is that the consequences of the various requirements are not obvious, hence it is difficult to avoid inconsistent requirements or to determine the best way for making tradeoffs. Ramification analysis is aimed at revealing relationships among different attribute values and helping a selector in specifying and changing requirements.

To test if the values in a requirement vector $R = (r_1, r_2, ..., r_n)$ are consistent, a dynamic R-tree based on R can be built. For a domain of n attributes, an R-tree has at most n stages. There may be up to N! different dynamic R-trees for the same requirement vector, each corresponding to a permutation of the n attributes. An important question is: in order to detect consistencies among a set of requirements, does it matter which R-tree is generated? To answer this question, we present the following theorems. Proof for these theorems can be found in[4].

Theorem 10: Any dynamic R-tree DRT(R,T) contains incompatible node(s) or is empty if and only if the requirement vector R is inconsistent for the set of TDMs T.

Definition 11: Let Level[*node*] equal the length of the path from the *node* to the root in an R-tree DRT. Let SN be the set of nodes in the DRT. A node (att_j, x_j) in a DRT is an essential node if

$$\text{Level}[(att_j, x_j)] = \max\{ \text{Level}[(att_j, x)] \mid (att_j,x) \in SN \}.$$

Definition 12: Let EN1 and EN2 be the set of essential nodes for DRT1 and DRT2, respectively. DRT1 and DRT2 are said to be equivalent if EN1 = EN2.

Theorem 13: If $R = (r_1, r_2, ..., r_n)$ is a consistent requirement vector, all dynamic R-trees DRT(R,C,A) are equivalent.

The above theorems indicate that although there are many dynamic R-trees for the same requirement vector, TDM set and attribute set, it is sufficient to generate only one of the R-trees in order to detect inconsistency among requirements. If the R-tree has "incompatible" nodes, then these requirements are inconsistent, otherwise they are consistent. The ramifications of a requirement vector $R = (r_1, r_2, ..., r_n)$ can be specified by the set of essential nodes of a dynamic R-tree. When requirements are inconsistent, tradeoffs have to be made in order to obtain a solution.

Ramification trees can also be used to predict the consequences of changing requirements, and to aid in incremental requirement specification, i.e. starting from the most important attribute and trying to infer as many other implied requirements as possible. In this way, some requirements can be automatically defined, inconsistency among requirements can be detected while the requirements are specified, and a set of totally qualified TDMs are found at the end of requirement specification, if any exists.

5 Reason directed backtracking

Selection is a process which often requires backtracking. In a selection process, failures are usually caused by one or more components of the requirement vector. Backtracking always involves modifying requirements. There are two primary kinds of failures:

1. **search failure**: no solution is found in a search,
2. **confirmation failure**: a solution is rejected by the designer.

Example 9. Consider the evaluation matrix in Table 2. If the requirements are (st, fm, fc, ao, ec) = (yes, {a}, 97, 30, 6), no TDM is completely satisfied. Let

$$\text{FAIL}(\text{TDM}_i) = \{ (att_j, v_{ij}) \mid \text{TDM}_i \text{ fails to satisfy } r_j \}.$$

The FAIL lists associated with the five TDMs are shown in Table 5.

TDM	FAIL(TDM$_i$)
TDM1	(self-testing, no)
TDM2	(area overhead, 35), (fault coverage, 95)
TDM3	(area overhead, 42)
TDM4	(extra connections, 7)
TDM5	(self-testing, no), (area overhead, 300)

Table 5: The FAIL lists

Each row in Table 5 represents a reason for failure. For example, the third row indicates that the requirement for area overhead is too high. If it is changed to 42, TDM3 will be a solution. The fourth row implies that the requirement for extra connections causes the failure. If one more extra connection is possible, TDM4 will be a solution. However the first row suggests that if self-testing is not required, TDM1 will be a solution without increasing the requirements on area overhead or extra I/O connections.

This example shows that there usually many reasons for a failure, and there are many ways to revise requirements. It is necessary to identify the most critical reason for failure, and find out how to backtrack such that a good solution can be found. In PLA-TSS, **reason analysis** is used to classify failures into a few categories and then identify the most critical failure category containing the current failure mode. For each failure category there is a pre-defined *remedy*. Once the main reason for a failure is understood, the system informs the designer of the reason, suggests possible ways to recover from the failure, then restarts a search process if the designer accepts the advice and/or adjusts the requirement values. We call this control structure *reason directed backtracking*. To demonstrate this method, we will show how search failures are handled next.

5.1 Reason analysis for search failures

A search failure occurs when no TDM completely satisfies a requirement vector R. Since TDM attribute values are fixed for a given circuit, the only way to resolve a search failure is to change R such that a solution can be obtained. How to change R depends on why the failure occurred.

Five classes of search failures are defined, namely three types of critical conflicts, a single conflict and multiple conflicts. There is a specific way for resolving each class of failure.

Critical Conflicts

1. First class critical conflict: There is a requirement r_j that no TDM can satisfy.

Remedy: When such a critical conflict occurs, the only way to continue the selection process is to relax the corresponding requirement to below the best value for att_j. The system will request that the designer modify the critical requirements shown in a conflict table, and then restart the search process. If the designer does not change the requirements, the selection process ends with no solution.

2. Second class critical conflict: There is a requirement r_j such that no TDMs which are close to the requirements can satisfy it.

Given a circuit and a requirement vector R, the TDMs can be divided into the two sets

$T1 = \{ TDM_i \mid score(TDM_i) > L \}$, and

$T2 = \{ TDM_k \mid score(TDM_k) \leq L \}$,

where L is a preset lower bound for TDM scores. T1 contains TDMs which we refer to as being relatively close to R, and therefore is referred to as the set of potential solutions. When T1 is not empty, a TDM will most probably be chosen from within T1, and TDMs in T2 are very unlikely to be selected unless some requirements are significantly changed. The difference between first and second class critical conflicts is that the critical requirement r_j can not be satisfied by any TDM in T1, although it can be met by some TDMs in T2.

Remedy: In this case, there are two ways to solve the problem. First, relax the requirement on att_j to below the best value in $\{v_{ij} \mid TDM_i \in T1\}$, such that it can be satisfied by some TDMs in T1 and a solution may be found. Secondly, change other requirements such that some TDMs in T2 which satisfy r_j will become close to the new requirements. However, since all TDMs in T2 have poor scores, any such TDM which satisfies r_j must have some very bad or unacceptable value in at least one attribute other than att_j. Comparing these two choices, the second one requires radical changes in order to lead to a solution, while the first one may only involve small changes. The system will inform the user of these two ways of changing requirements, and suggest the user change r_j.

3. Third class critical conflict: No TDM is close to R.

Remedy: This is the case where every TDM has a very bad score. Major changes to some requirements are necessary, and these may involve changing an unacceptable value to an acceptable one. The system first determines the necessary changes which have to be made, and then suggests the easiest change in order to obtain a solution using information from reason analysis and the scores.

Single conflict: There exists a TDM which meets all but one requirement.

Remedy: If a single conflict occurs, a solution can be obtained by changing a single r_j. In this case, a conflict table is displayed which shows which TDM fails which requirement. The designer is advised that if he is willing to change the value of a requirement to the extent shown in the table, a solution is obtained. If several single conflicts occur simultaneously, the system can suggest which change leads to the best solution. Since the designer can see how close he is to a solution, he may decide which requirement to change or which TDM to choose.

Multiple conflict: A TDM fails to satisfy at least two requirements.

Remedy: If a multiple conflict occurs, many alternatives exist, especially when all TDMs have multiple conflicts. The designer is informed that tradeoffs must be made among the various attributes. A conflict table is presented which informs the designer as to what conflicts exist. Several TDMs with high scores are suggested as possible solutions. Advice about the most beneficial tradeoffs is given, which is based on the TDM scores. The designer can select a TDM, or ask the system to select the best one, or change some requirements and let the system search again.

5.1.1 Ordering of failure classes

It is not enough to only classify failures, because several classes of failures may be present at the same time and the failure classes may not be disjoint. Precedences among failure classes must be defined such that an intelligent decision about which failure to solve first can be automatically made. In PLA-TSS, the search failures are ordered and processed according to the following rules.

1. Critical failures are dealt with first. Among them, the precedence is first class critical conflicts, followed by the second class, and finally third class.

2. If there are no critical failures, single failures are handled before multiple failures. But if the failure class associated with the best (highest score) TDM is a multiple failure, it is treated together with single failures, since such a multiple failure may lead to a minimal degree of changes.

Example 10. Consider the evaluation matrix in Table 2 and the requirements and weights specified in Table 3 and the PCFs used at the beginning of Example 5. In searching for a solution, four single failures and a multiple failure occur simultaneously. After reason analysis, the system will inform the user that he may change a single requirement on area overhead from 30 to 42 in order to obtain a solution. However if he can lower the requirement on fault coverage a little bit from 97 to 95 and relax the requirement on area overhaed from 30 to 35, the best solution can be obtained. At this point it is up to the user to decide which way to go.

The prototype PLA-TSS has demonstrated that reason analysis directed backtracking is a suitable control strategy for the selection process. This is because a selection process is often failure-driven. Actions to take depend on what kind of failure has been encountered. Reason analysis, combined with scores and ramification analysis, provides intelligent information on where and how to backtrack after a failure has occurred, and thus lead to a efficient selection process.

6 PLA-TSS overview

The basic structure of PLA-TSS is given in Figure 7.

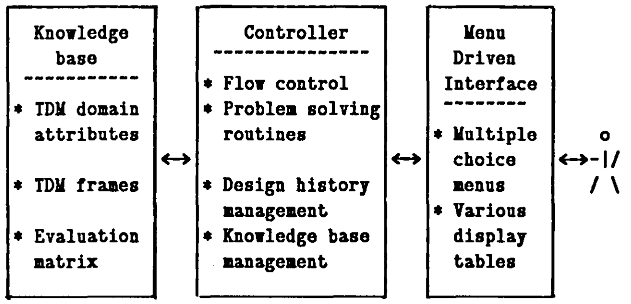


Figure 7: Overview of PLA-TSS

The knowledge base

An important part of PLA-TSS is a knowledge base which contains information about the domain of TDMs. To insure changeability and expandability, the knowledge base is separated from the controller, and is relatively static. TDMs are represented by TDM frames. Currently, there are fifteen TDMs in the knowledge base. New TDMs or attributes can be readily added to the knowledge base without changing the program.

The controller

The controller of PLA-TSS gets information from the knowledge base, reasons with this knowledge, and interfaces with the user. It has four basic functions.

1. Control a dynamic selection process. The controller is responsible for determining when to do what. The basic control flow is shown in Figure 8.

2. Make a selection. To simulate a human expert, the controller embodies the knowledge about the art of selection in a set of problem solving procedures, such as ramification analysis, search routines, and the comparison, PC and score functions for evaluating TDMs. These procedures are called at the appropriate time to allow the system to make a good decision or give intelligent suggestions.

3. Keep track of design data and history. In order to allow backtracking in a selection process, the controller keeps a record of the entire design history. A history consists of a sequence of states in a selection process. Each state is represented by the values of global variables. At every decision point or when the system state is changed, the old state is recorded so that a designer can trace back (review) or back up to any earlier state and undo some actions.

4. Manage the knowledge base. Tools are provided for maintaining the knowledge base, such as changing information, reviewing the contents of the knowledge base, deleting information or acquiring new TDMs.

The interface

An important feature of PLA-TSS is that the consultant system and the designer act at the same functional level, i.e., the control of the selection process may switch between the two parties. This feature implies that the control flow in a selection process cannot be unidirectional. To implement such a system, PLA-TSS uses a combination of sequential control, reason directed control and multiple branching. Interaction between a designer and the system is through status report plus menu driven interface. Many different tables are used to display system states in various situations. Menus with multiple choices are shown which allow a designer to select his most preferable actions.

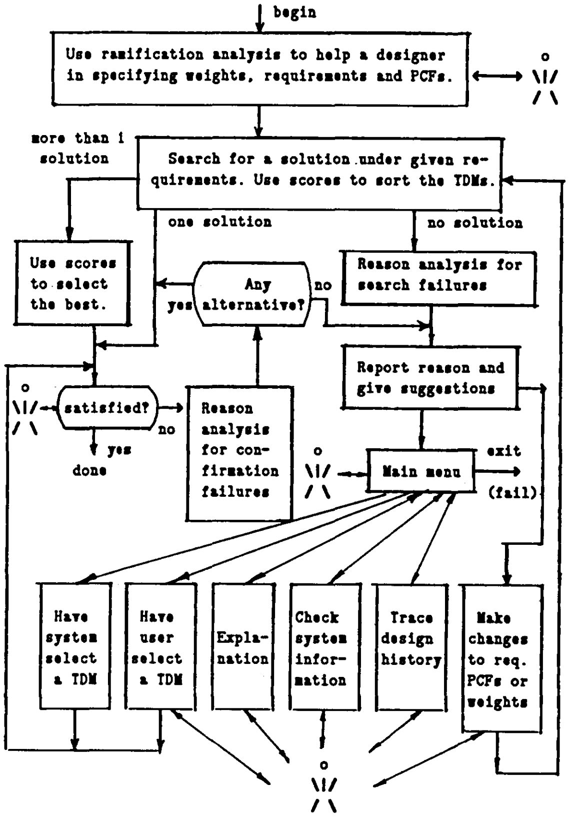

begin
Use ramification analysis to help a designer in specifying weights, requirements and PCFs.
o
\|/
/ \
more than 1 solution
Search for a solution under given requirements. Use scores to sort the TDMs.
one solution
no solution
Use scores to select the best.
Any alternative?
yes
no
Reason analysis for search failures
Report reason and give suggestions
o
\|/
/ \
satisfied?
no
yes
done
Reason analysis for confirmation failures
o
\|/
/ \
Main menu
exit
(fail)
Have system select a TDM
Have user select a TDM
Explanation
Check system information
Trace design history
Make changes to req. PCFs or weights
o
\|/
/ \

Figure 8: Basic control flow in PLA-TSS

7 Conclusion

In this paper we described a knowledge based system for TDM selection. Such a system contain two kinds of knowledge, namely knowledge about the alternatives to be selected and knowledge about how to make a selection. In our prototype system PLA-TSS, the domain specific knowledge is represented by frames, and the domain independent knowledge is represented by procedures. By loading in different domain specific information into the knowledge base, the same scheme can be used to select TDMs for other circuits, or select other objects for different purposes. PLA-TSS is implemented in LISP and runs on a TOPS-20 machine. This research shows that requirement constrained selection is a good field for applying knowledge based systems, because extensive knowledge and heuristic reasoning, which often fall beyond the capability of human selectors, are required to make a good selection.

References

1. Abadir, M.S. and M.A. Breuer, "A knowledge based system for designing testable VLSI chips", *IEEE Design & Test of Computers*, Vol. 2, No. 4, August 1985, pp. 56-68.

2. Breuer, M. A., "A methodology for the design of testable VLSI chips", *Proc. IEEE Workshop on Test Environments*, 1985, pp. 7-38.

3. Williams, T.W. and K.P. Parker, "Design for testability - A survey", *IEEE Trans. on Computers*, Vol. C-31, No. 1, Jan 1982, pp. 2-15.

4. Zhu, X., *A Knowledge Based System for Testable Design Methodology Selection*, PhD dissertation, Dept. of EE-Systems, University of Southern California, August 1986.

Deriving Module Interconnectivity from Behavioral Specifications, and Coupling a VLSI Layout Editor for Error-free Routing

Ganesh C. Gopalakrishnan, Nai Chi Lee, David R. Smith, and Mandayam K. Srivas

Department of Computer Science, State University of New York, Stony Brook,
NY, U.S.A.

ABSTRACT

The problem of inconsistencies among multiple, and independently originated manual specifications is a serious one in VLSI design. This paper proposes an integrated VLSI design system that solves this problem.

High-level behavioral specification is the major source of design information in our system. We can infer from it many pieces of information needed for low-level tools, including module interconnectivity ("netlist") information for layout routing. All manual input will be checked against the derived information.

In this paper, we outline an algorithm to infer module interconnectivity among the submodules of a module from (a) the behavioral specifications of the module, and (b) knowledge of the names of the ports used by submodule-operations. We also describe a VLSI layout editor that is driven by the automatically generated netlist.

1 Introduction

Most current VLSI design methodologies are "tool-kit" based. Human designers are required to provide inputs to the individual tools separately. The problem with this approach is that independently originated manual specifications can have inconsistencies among them. For instance, using layout editors such as [SMHO85] and [MCN], users can inadvertently create layouts that would behave quite differently from the original high-level behavior specifications. Although complete design automation using a "silicon compiler" seems to be the long-term solution, we don't have any practically viable tool of that nature yet.

With the tool-kit based approach, the above-mentioned errors would get discovered only very late during the design. Inconsistencies may even remain undetected because extensive (or exhaustive) simulation may be unacceptably expensive.

We argue that all the activities of designing a circuit module ought to stem from one high level specification written for it. In order to do this, the traditional role of high-level specification languages must be extended beyond design documentation and high-level simulation to include the roles of (i) automatic control synthesis, (ii) providing information for low-level simulation tools, (iii) providing module interconnectivity information for layout tools, etc. More importantly, the high-level specification language itself ought to be designed with formal verification of design correctness and extraction of information for low-level tools in mind. Unfortunately, most of today's high-level specification languages do not meet these objectives.

We have designed a high-level specification language with these objectives in mind. Our language (called "Structure and Behavior Language" (SBL)) supports viewing hardware modules as *abstract data types* [GH78,GTW78]. Modules are specified in a purely applicative language. SBL supports the formal verification of high-level specifications, and high-level simulation. In addition, we have developed algorithms for extracting information from SBL specifications and passing it on to low-level tools that perform controller synthesis and chip layout. In future, a switch-level simulator will also be driven (in part) by the information generated from SBL.

In this paper, we present an algorithm to infer module interconnectivity and module hierarchy from behavioral specifications. We also describe a VLSI layout editor that is guided by the human designer, and that verifies the actions of the designer against the automatically generated hierarchy and interconnectivity information. In addition, our system operates hierarchically. This means several things:

- Behavioral specifications are written and verified hierarchically;
- Module interconnectivity is generated hierarchically;
- Routing is performed hierarchically.

With these provisions, we hope to demonstrate a design system that guarantees correctness of both behavior and layout, as well as performing the complex task of routing efficiently, by operating hierarchically.

Figure 1 illustrates our design approach. Assume that a module M is to be designed. Two specifications are writ-

This research was supported in part by NSF via grants DCR-8401624 and DCR-8319966

CH2345-7/86/0000/0864$01.00© 1986 IEEE

ten for it: (a) An *abstract* specification, that specifies the desired behavior of M without assuming anything about its implementation; (b) A *realization* specification that specifies the submodules (M_1 and M_2 in this case) and an algorithm to implement the operations of M using the operations supported by M_1 and M_2. All the interconnections needed to support the algorithm of the realization specification in hardware—specifically, all the wires needed to support all the data movements between M_1 and M_2—will be automatically generated by the parser of the realization specification. (The boxes labeled 'C_x' are controllers. In our system, these controllers will also be derived automatically from realization specifications. Once synthesized, the controllers are similar to the other modules as far as routing goes.)

The main advantages of this approach are the following. Assuming that M_1 and M_2 are correct, the structural correctness of M depends only on the realization specification of M. (We have shown in [GSS86] how a realization specification may be shown with respect to an abstract specification. Once we show a realization specification to be correct, it can be used as a basis for deriving the interconnections among M_1 and M_2.

For example, consider a FIFO module. The abstract specification of the FIFO (not shown) specifies the invariants to be satisfied by any implementation of the FIFO. Figure 2 shows the realization architecture and the algorithms to implement the submodule operations, chosen by the designer. The realization specification will be verified against the abstract specification.

In our approach to designing VLSI systems, we would refine a module's specification enough times so that the leaf modules of the hierarchy are present in a module library, or are simple enough to be built outside our framework. It is using these library modules that the circuit of M is constructed. The actual routing would be carried out bottom-up in the hierarchy.

Details of our research have been reported in [GSS85,GSS86]

Organization

The algorithm to infer module connectivity will be illustrated on a simple example. The human-interface to the layout editor is then explained. Finally, we present our concluding remarks.

2 The Connectivity Inference Algorithm

We illustrate the connectivity inference algorithm with the aid of a simple example—that of the implementation of a First In First Out (FIFO) queue. Explanation will be simplified, and made at a high level, with emphasis only on the port and interconnection related details.

The FIFO supports three operations:

1. `ins`, which enqueues a data item coming in via input-port `FIFO.?d` (input port `?d` of module FIFO) and creates a new queue state (line 10);
2. `rem`, which removes the front element and returns a new queue state (line 13);
3. `front`, which reads the front element and brings it out via port `FIFO.!d` (line 14).

We will study the implementation of the `ins` operation, and derive a netlist for it. The implementation uses the submodule memory `MEM` as a circular buffer into which two pointers (contents of `WC` and `RC`) point, indicating the location to be next written (when items are inserted into the FIFO) and the location to be next read (when the front of FIFO is to be examined). The unit `SB` is a status block which (i) records whether the last operation performed on the fifo was *ins* or *rem*; (ii) compares the outputs of `RC` and `WC`; if these are equal, it sets status output `!full` to true if the last operation performed was *ins*, and `!empty` to true if the last operation performed was *rem*.

The meaning of *ins* as declared on lines 10 and 11 may be understood as follows:

> Given a state $<< M, R, W, S >>$ (a tuple of the states of the submodules) of the FIFO and given a value *val* to be inserted, the *ins* operation creates a new state of the FIFO in which:
>
> - The memory's state has advanced to one in which *val* has been written into it, at an address obtained by reading the write counter W.
> - The state of W advances to $up(W)$.
> - The state of R is unaffected.
> - Information regarding the last operation is recorded in the status block.

Port-Operation Association

The key feature of SBL is that each module is characterized solely via its operations. This means the following:[1]

- The operations ("services") provided by a module are captured by associating one operation for each service. In the present example, `ins`, `rem` and `front` are some of the operations provided by the FIFO;
- The ports used by module operations for data I/O are associated with the operations.

For example, the `ins` operation will have two pieces of information associated with it:

[1] Timing characteristics are also associated with module operations

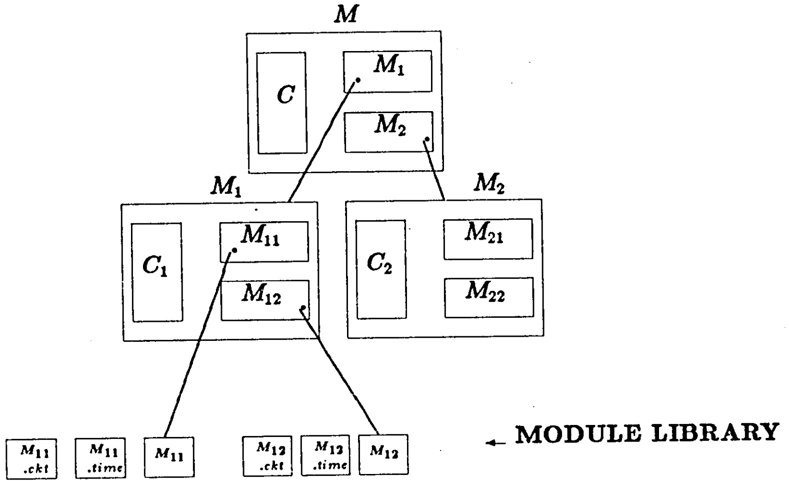
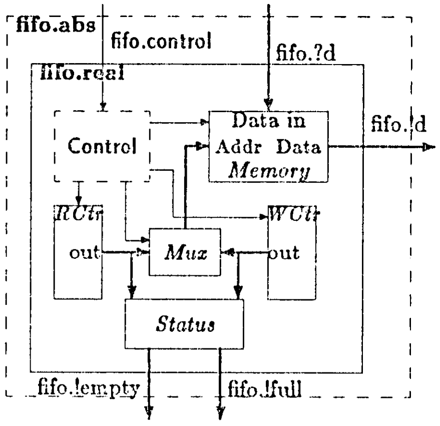


Figure 1: Hierarchical Design of Controllers and Datapaths



```
REAL_MODULE FIFO (capacity, maxdata : INTEGER ) : fifo
CONST naddrbits = log2 (capacity)
SUBMODULE
    MEM    : mem    ( naddrbits, maxdata ) % creates an instance of 'mem'
    MUX    : mux21  ( naddrbits )
    RC, WC : ctr    ( naddrbits )   % both RC and WC are instances of 'ctr'
    SB     :  sb    ( naddrbits )
STATE   MEM, RC, WC, SB
DEFUN                               % Error checking omitted for clarity
ins ( << M, R, W, S >> , v ) <==
    << write (M, sel2(MUX, read(W)), v), R, up(W), last_op_ins (S) >>
rem ( << M, R, W, S >> )
            <== << M, up(R), W, last_op_rem (S) >>
front ( << M, R, W, S >> )
            <== read(M, sel1(MUX, read(R)) )
END FIFO
```

Figure 2: Schematic and Realization Specification of FIFO

MODULE	OPERATION	I-PORTS	I-PORT-TYPES	O-PORTS	O-PORT-TYPES
mem	write	mem.?addr mem.?data	addr-type data-type	none	
mux21	selw	mux21.?w	addr-type	mux21.!o	addr-type
ctr	read	none		ctr.!o	addr-type

Figure 3: Port-Operation Association for the Submodules of FIFO

- The way it is implemented: This is as given on line 11 of figure 2;
- The port(s), and the type or the port(s) that it uses: This information will be specified in the abstract specification of the FIFO, as shown below:

```
PORT
 ?d : ARRAY [8] OF BIT
OPERATION
 ins : FIFO, ?d ==> FIFO
```

  In this example, we assert that the port ?d of FIFO carries items of type ARRAY [8] of BIT, and further, the operation ins is associated with the physical module FIFO, and that it consumes a data item through the physical port ?d of FIFO and returns a new FIFO state.

In general, the type of a port such as ?d would depend on the *size-parameters* of a module. For example, the size parameters of FIFO are datasize and capacity (figure 2, line 1). This helps us to *resize* an entire realization by changing the value of one or more size-parameters and deriving a new netlist.

We now list the port-operation association for the operations of all the submodules used in FIFO (figure 3). This will be the basis for the connectivity inference algorithm presented in the next section.

Connectivity inference for ins

The connectivity inference will be demonstrated for the expression on line 11 of figure 2, consulting figure 3 in the process.

Let us consider the subexpression `write(M, selw(MUX, read(W)))` . This expression uses specific instances of modules (the instantiation done on line 4, for example, says that MEM is a module that is of type mem tailored for sizes addrsize and datasize). We can convert the subexpression being considered into one in which the types of the modules (and not instances) are used. Doing so gives us the subexpression `write(mem, selw(mux21, read(ctr)))`

At this point, we can consult figure 3 and start deducing connectivity. For instance, the expression read(ctr) produces its result on port W.!o. This expression forms the argument to operation selw on module mux21, which expects its input to come via port mux21.?w. At this point we do the following:

- Check that the type of the port W.!o matches that of port MUX.?w; in this example, both of them are of type "addr-type", and hence the types agree;
- Infer that port W.!o must be connected to port MUX.?w

In a similar fashion, we can infer the entire connectivity of FIFO as illustrated in the schematic shown in figure 2.

Notes:

1. The user can (for documentation purposes) provide connectivity assertions in a realization specification by providing "CONNECT" declarations. These will be checked against the inferred connectivity, and only if these agree will the user-given connectivity be used for routing.

2. The "CONNECT" construct also gives the user a facility to introduce sharings of busses—for instance, ports SM1.!o and SM2.!o of two submodules can be attached to a common bus by declaring that they are both connected to an node "BUS".

3. There are additional details involved in connectivity inference. They are: (a) Propagating connectivity assertions through "helping" functions and recursive definitions. (b) Detecting (rare) inconsistencies that a user can inadvertently cause. We have worked out (but do not present) such details.

An excerpt from a terminal-session that shows the inference of module interconnectivity at a high level is shown in figure 4.

3 Interface With Layout Editor

In order to make use of the information in SBL's realization specification during the layout stage, a correspondence between SBL descriptions and the layout has to be established. The following relationships exist:

- There is a one-to-one association maintained between its physical layout of a module and the module instance declared in an SBL specification (for example MEM, on line 4, figure 2);
- The hierarchy of modules correspond in the layout and the SBL descriptions;
- External ports in these realms also correspond.

We generate an intermediate file, *CELL.info* from the SBL specification of a module, *CELL.abs* and *CELL.real*. This *.info* file contains essential information about submodules, ports, inter-module connections and hierarchy. The format of *.info* file is shown in figure 5.

We are currently using *magic* [SMHO85] version 4.1 with the following local enchancements that help in interfacing with SBL.

Hierarchy enforcement

In our modified layout system, any magic command that may cause changes in the current cell's hierarchy (such as :getcell and :array) are first verified against the *.info* file before execution. Checks are also performed at the end of each layout editing session to warn the user about missing subcells, if any. By this, we ensure that the current cell *M.mag* includes a subcell *A.mag* if and only if *A* is a submodule of *M* (as specified in *M.info*).

Port labeling

Magic by itself lacks the concept of *ports.* To map the SBL port specification of a module M onto its magic file, the user has to manually mark each port on the layout. Complex ports in SBL (such as arrays and records) are first converted into simple bit-ports. In general, the type-information associated with a module's port is used in determining the actual number of wires needed in hardware to convey objects of that type.

We have added a :port command which makes use of the label construct in Magic to denote ports. This command ensures that port names and indices (if any) marked on the layout agree with that specified in the *.info* file. By modifying magic's :label command, we can also avoid conflict between SBL port names and other user-defined labels.

This port labeling scheme is utilized by magic's router. In future, we plan to make this information available to other CAD tools such as *crystal* [SMHO85] or *rnl* [Ter83], which at present require the user to provide the port-labels separately.

Routing Assistance (currently being implemented)

Magic comes with a powerful switchbox router, *detour*, that is suitable for hierarchical routing. What is currently missing is the capability to specify the netlist required by this router in high-level description language. The process of generating a netlist manually (either by writing a netlist file or interactively marking it on the layout) is tedious and error-prone. In our layout system, we overcome all these shortcomings by using the inferred netlist in conjunction with the above mentioned port-labeling scheme.

Routing involves a top-down phase and a bottom-up phase. During the top-down design stage, the interconnections between the datapath submodules at each level of the hierarchy are generated. A magic-compatible netlist can be compiled from this. During the bottom-up phase, detour is used to perform routing. Routing would proceeds bottom-up over the hierarchy.

Note that the controller is not one of the datapath submodules. In our system, the controller specification (in some FSM description language such as *meg* [SMHO85]) is compiled from SBL realization specifications and timing requirements. The netlist involving the control path is also generated during this compilation process. A locally modified version of *mpla* [SMHO85] can then be used to produce the layout of the controller in PLA form. The resulting controller module is now ready to be used in the layout, just like one of the subcells.

A Typical Routing Session

In the following session, it is assumed that the user is attempting to layout the cell *FIFO.* We assume that all subcells of *FIFO* are available as library cells (since layouts are built bottom-up), and that all essential SBL information have been already extracted and stored in the *FIFO.info* file (since SBL specifications are top-down).

Once the user loads *FIFO.mag* into magic, the connectivity information is extracted from *FIFO.info.* A magic-compatible netlist file is then created as *FIFO.net,* which includes netlist for data-path, controller, clock and power. The user may now issue the command :netlist FIFO in magic to use this netlist.

The user then has to perform manual placement of subcells in FIFO, and manually routes the power nets first before invoking magic's router (these may not be necessary in the future when tools for automatic cell placement and power net routing become available). Our routing approach is incremental and interactive. The user can selectively route groups of special nets, *e.g.* busses, or manually draw the wire for certain critical nets.

When the user is ready to invokes the *detour* router via the :route command, our routing assistant has to first verify any existing layout connections against *FIFO.net* and construct a new netlist consisting of only those nets that are not yet connected. The assistant then call up *detour* to route the layout.

As with any other router, there is a chance that *detour* may fail to complete some of the connections. Therefore the user should always examine the correctness of routing results by using the :verify command in magic. To an experienced user, a visual inspection on the layout can usually suggest possible problem areas and means to improve routing quality.

If the result of routing is not satisfactory, the user must either completely undo the previous routing, or selectively remove some of the nets from layout (there is a :ripup command in magic just for this purpose). The user must now offer some help to the router by
— improving cell placement,
— allocating larger routing area, and/or
— manually routing some critical nets
before invoking the :route command again.

The cycle then continues until the user is satisfied with the layout of *FIFO.mag*

4 Concluding Remarks

We have presented a VLSI design system that infers connectivity from high-level behavior specifications. Its main features (combining type-checking, maintaining consistency with behavior and error-free routing) were pointed out. In addition, the following features will also be present in our implementation:

```
<> read "FIFO.real"          % Compile the realization spec.
   ...compilation messages...
<> inst FIFO (8,15)          % supply size parameters to FIFO
<> netlist

FIFO.?d --> MEM.?data    : (SUBRANGE-TYPE (INT-TYPE 0) (INT-TYPE 15))
WC.!o --> MUX.?w         : (SUBRANGE-TYPE (INT-TYPE 0) (INT-TYPE 7))
MUX.!o --> MEM.?addr     : (SUBRANGE-TYPE (INT-TYPE 0) (INT-TYPE 7))
RC.!o --> MUX.?r         : (SUBRANGE-TYPE (INT-TYPE 0) (INT-TYPE 7))
MUX.!o --> MEM.?addr     : (SUBRANGE-TYPE (INT-TYPE 0) (INT-TYPE 7))

        % The above interconnections and their types were inferred
        % during compilation. This will be converted to Magic's
        % netlist and used for routing. Type of data is 0..15
        % and address is 0..7 (since first param of FIFO = capacity)
<>
```

Figure 4: Terminal session that demonstrates connectivity inference

- Arrays of modules are ubiquitous in VLSI. Behavior specifications that use arrays of modules are written in SBL by using recurrence equations. We plan to infer connectivity among the arrays of modules directly from the recurrence equations.

- It is a common practice in hardware systems to split a bus and connect it to two different modules. Although this may seem to be purely a structural operation, it has an important bearing on *behavior* as well. For example, a connection shifted one by mistake will introduce an un-intended shift operation into the behavior, undermining the behavioral correctness.

  Due to the above danger, we absolutely prohibit manual bus-splitting at the structural level. Instead, the user is required to ascribe a record type to the bus, and then use *field selection operations* on record values conveyed to the module (via the bus). From the field selection operations in the behavioral specification, we will infer the necessary bus splittings required. Thereby the interconnections and the behavioral specification would remain consistent.

We are at the moment able to run the SBL compiler on non-array modules, generate their net-list and route their submodules in accordance with this net-list. The main tasks that lie ahead of us include generating netlists for arrays of modules and the implementation of the controller synthesis procedure, which would allow us to route the generated controllers also, to make complete chips.

References

[GH78] John V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[GSS85] Ganesh C. Gopalakrishnan, David R. Smith, and Mandayam K. Srivas. An algebraic approach to the specification and realization of VLSI designs. In C. J. Koomen and T. Moto-Oka, editors, *Proc. Seventh International Symposium on Computer Hardware Description Languages*, pages 16–38, North Holland, 1985.

[GSS86] Ganesh C. Gopalakrishnan, David R. Smith, and Mandayam K. Srivas. *From Algebraic Specifications to Correct VLSI Circuits.* Technical Report 86-13, Department of Computer Science, State University of New York at Stony Brook, 1986. An extended version of the paper to be published in the proceedings of the IFIP working conference on "From HDL Descriptions to Guaranteed Correct Circuits, Grenoble, France", North-Holland (1986).

[GTW78] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.* Volume 4, Prentice Hall, Englewood Cliffs, N.J., 1978.

[MCN] MCNC. Vivid: a VLSI layout editor. Product of Micro-electronics Center of North Carolina.

[SMHO85] Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout. *1986 VLSI tools: Still More Works by the Original Artists.* Technical Report UCB/CSD 86/272, Dept. of EECS, Univ. of California at Berkeley, December 1985.

[Ter83] Christopher J. Terman. *Simulation Tools for Digital LSI Design.* Technical Report MIT/LCS/TR-304, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA, September 1983.

```
% This is the simplified format of an info file.
% Everything after a percentage sign is treated as a comment.
%
<< MODULE >> % this is the common name for .real/.mag files.
MODULE-NAME
<< PORT >> % I/O port names, with array dimension declaration.
PORT-NAME1        dimension
PORT-NAME2        dimension
. . .
<< SUBMODULE >> % submodule names, with array dimension delaration.
SUBMODULE-NAME1        dimension
SUBMODULE-NAME2        dimension
. . .
<< CONNECT >> % connection list between ports, with dimension.
PORT-A — PORT-B        dimension
PORT-1 — PORT-2 — . . . — PORT-N        dimension
. . .
% Port specifier has the general form:
%        submodule.portname^record[index]
<< END >>
```

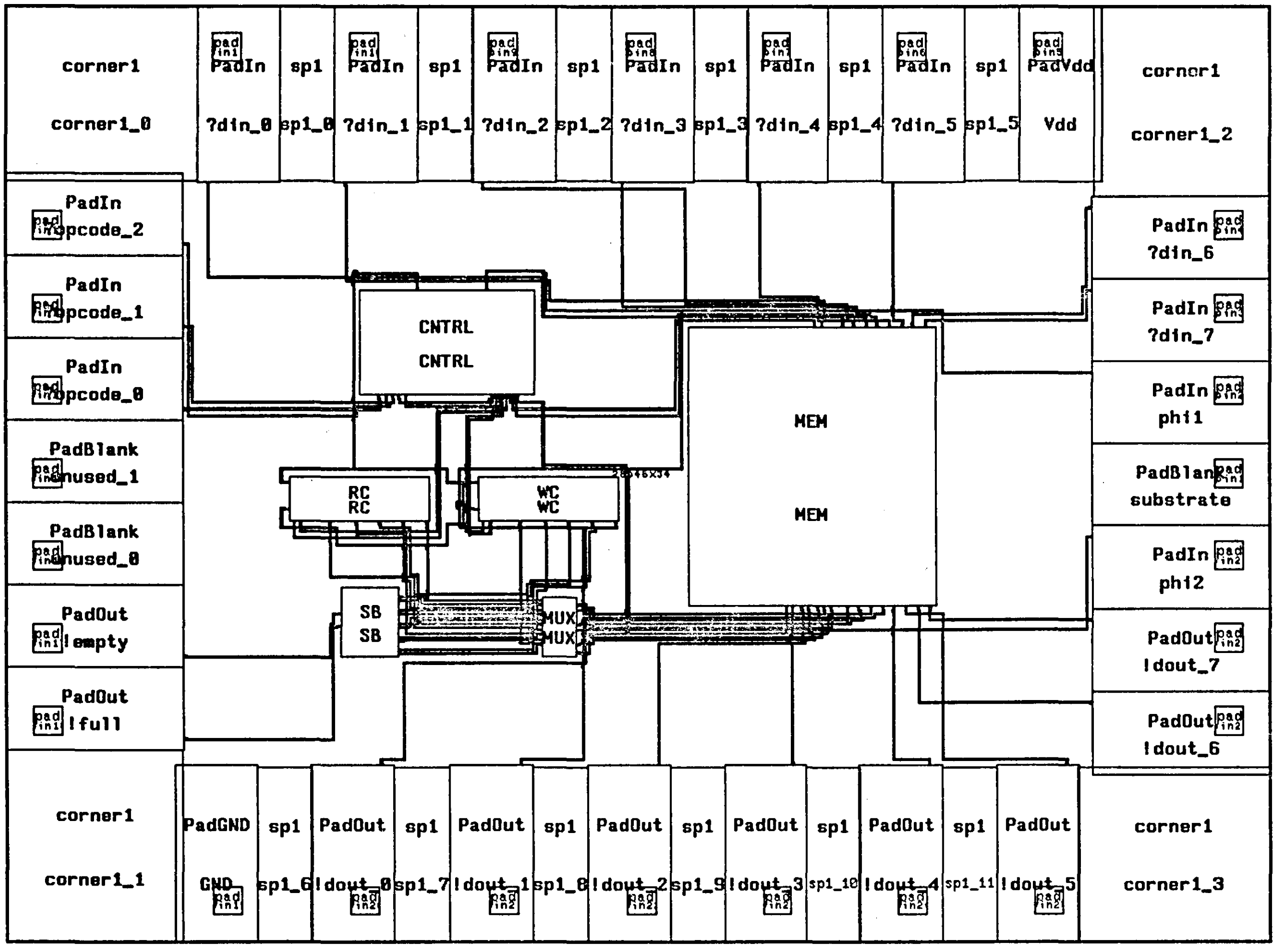


Figure 5: General Format of *.info* file; FIFO's signal-nets routed using Magic

Recent Results in VLSI CAD at MIT

Richard E. Zippel[1], Paul Penfield, Jr.[1], Lance A. Glasser[1], Charles E. Leiserson[1]
John L. Wyatt, Jr.[1], F. Thomson Leighton[2], and Jonathan Allen[1]

[1]Department of Electrical Engineering and Computer Science
[2]Department of Mathematics
Massachusetts Institute of Technology
Cambridge, MA 02139

1. Introduction

Before the mid 1970s, integrated-circuit designs were comparatively simple, and the limits to complexity of an integrated circuit were imposed by semiconductor technology. Advances in semiconductor technology, however, have made it possible to put more on a single chip than can be designed easily and simply by a small group of people. As a result, for the past ten years or so, as the complexity of the designs has increased, the construction of integrated systems has been limited by both the fabrication technology and the cost of full custom design. As today's design crisis is solved, it is expected that the complexity limits will also include our inability to envision the types of systems to deploy in integrated form. The field will become system-limited, as well as technology-limited and design-limited.

MIT has responded to all this with a program of research and education in microsystems aimed at three principal areas: fabrication technology, design, and systems. This paper is devoted to a discussion of some of the recent results in the area of VLSI CAD. This work is carried out in the context of research in other areas of VLSI, and so interactions are both possible and desirable.

This paper discusses only research carried out on the MIT campus. MIT Lincoln Laboratory has a strong program of research in VLSI which is separate.

2. VLSI Research Program

MIT recently opened a state-of-the-art VLSI fabrication facility on campus. It includes a fully equipped 6800 square-foot integrated-circuit laboratory, with about 2800 square feet of class 10 clean space and all the equipment necessary to process wafers from starting material through to final packaging and testing. Its purpose is to provide a stable, base-line process to support research in processing and process control, and to fabricate systems that require unusual process steps. There is also a 3600 square-foot class 100 lab dedicated to novel process technology development, and a 2000 square-foot Submicron Structures Laboratory, including 1000 square feet of class 10 space.

It is convenient to consider the MIT VLSI research program as being composed of seven interrelated parts: submicron technology, electronic materials, semiconductor processing, semiconductor devices, VLSI CAD, VLSI systems, and VLSI theory. The total program is, roughly half devoted to the "electron-oriented" activities, and half to the "bit-oriented."

Much of the research in semiconductor processing and devices could not even be attempted without the new facilities mentioned above. This work includes a program, sponsored by the Semiconductor Research Corporation, on process technology for mixed analog and digital circuits. The research vehicle is a high-accuracy, high-speed A/D converter. It also includes a plan to develop software control systems for IC fabrication facilities. The purpose of the proposed Computer-Aided Fabrication system is to improve flexibility of fabrication, repeatability of processes and experiments, portability of processes, yield and efficiency of process development, thereby reducing overall cost and latency time in IC fabrication. These benefits arise from more complete documentation, elimination of paper in the clean environment, avoidance of human errors, optimal scheduling, environmental monitoring, inventory management, cost accounting, direct equipment drive, and convenient process-development and process-analysis environments—in short by management of all the data associated with VLSI fabrication. This CAF work is part of the VLSI program and it is also part of a broader program of research in manufacturing.

At the other end of the spectrum, there is interest at MIT in highly parallel architectures, since such architectures match the capabilities of VLSI to provide massive amounts of logic. This work includes the development of dataflow machines, and particularly the languages and software systems necessary to make effective use of highly parallel systems, and of algorithms that are particularly well suited for such an environment.

Although most of the VLSI research can be conveniently classified as belonging to one of the seven areas men-

CH2345-7/86/0000/0871$01.00 © 1986 IEEE

tioned above, some of the more interesting projects seem to defy such categorization and are truly interdisciplinary. One of the best examples is the development of a special content-addressable memory whose architecture has been designed to be usable in a wide variety of different applications [24, 29]. Making full use of this architecture is the subject of continuing research. To achieve the desired density and performance goals, novel circuits and a low-resistance interconnect technique have been developed. This effort is funded by a consortium of eight industrial firms, and involves three faculty members from diverse areas (L. R. Reif, processing; C. G. Sodini, circuits; and R. E. Zippel, architecture).

A paper this short cannot discuss research results from all these areas; instead, the focus is on recent results in VLSI CAD.

3. Recent results in VLSI CAD

Nine recent results or projects in VLSI CAD are discussed below. These include particular design tools, and also approaches that support the development of new tools.

3.1 CAD Tool Frame

SCHEMA [30] is an integrated design system for electronic designs being developed by a group led by Richard Zippel. Also involved in SCHEMA's development is a group of researchers at Harris Corp. being led by George Clark. Building VLSI systems is very complex undertaking. The complexity of the task is manifest in two fashions. First, the designs themselves are quite complex and the designers need help managing the complexity of the design. Second, the design tools themselves are becoming more and more complex. SCHEMA has been developed to address both of these types of complexity.

SCHEMA addresses the design complexity issue in three ways. First, SCHEMA provides a complete design structure in which the designer can store schematics, layouts, simulation results and all other artifacts of the design. Thus it captures the *complete* design. In order to understand the impact of different design decisions, the designer needs to examine the design from different perspectives. SCHEMA allows the designer to walk around the design laterally (through the schematics, layouts and simulation results) or vertically (from block diagram through logic and circuit schematics). Furthermore, SCHEMA maintains consistency between the different viewpoints of the design, if possible, and marks the different components incompatible if not.

The second approach SCHEMA uses to simplify large designs is to encourage the development of a library of *simple tools*. Simple tools perform a single, simple task to help the designer. Some examples are a *delay estimator* that calculates the delay between two nodes in a circuit or a *power estimator* that computes the average power dissipated by a set of devices. This information can be easily derived with a simulator or calculated by hand, but in either case requires enough effort on the designer's part that it is not often done. The purpose of the simple tools is to make this information available to the designer whenever and wherever it is needed so that design decisions can be made in a confident fashion. Another simple tool would convert a boolean equation into a pull-down or pull-up structure. Again an easy operation for a designer but somewhat time consuming. None of these tools is a major CAD advance in and of themselves but it is our belief that cumulatively they have a major impact on the designers productivity.

Finally, the big tools: simulators, routers and compactors, almost always have several variants. Simulations can be done using a switch level simulator, a logic simulator or a transient simulator. Routing can be done using any of a number of channel or switch box routers. Each of these techniques is appropriate for a different segment of a design. They should be built to be interchangeable, so that the designer can choose which one to use at different points in the design. The best example of this type of CAD tool currently in use is a mixed mode simulator, though the decision process is controlled by the program not the designer.

To support these types of tools, the components of SCHEMA must be more thoroughly integrated than is of true most other CAD environments. This is accomplished by having the entire design and most design tools coexist in a common address/namespace. To encourage interchangeability in the software tools and to minimize the effort required to build CAD tools, a "Layered Language" software organization is used. Finally, a fairly complete and uniform toolbox of user interface tools is provided. Thus it is relatively rare that a tool designer need be concerned with developing a user interface to his or her tools.

3.1.1 Software Organization

The most common approach used in building a software system is to use a *structured design* methodology. With this methodology, a specification for the problem to be solved is given. Using this specification, the problem is decomposed into sub-problems, which are solved independently to encourage modularity. This approach is then applied recursively to each of the sub-problems. This leads to a tree-like decomposition of the problem into sub-problems. This modular resolution of the problem makes it relatively easy to measure progress on the orignal goal and provides accountability for faults, important issues in developing large software systems.

We feel that this approach is weak for a number of reasons. First and most important, invariably the problem posed is not the problem that should be solved. Either the problem's specification is incomplete or inaccurate, or by the time the system is built the needs will have changed and a slightly different system is required. Second, the modularity imposed by structured design leads to duplication of effort as each team builds the tools necessary to solve their problem. If they choose to share modules with groups working on other subproblems, then hidden dependencies will appear among the modules sharing code, reducing their modularity. Finally, the accountability benefits of structured design are really illusory. Hard problems in the resulting system are more often due to poor decomposition of the problem; thus the responsibility for the problems is collective.

In SCHEMA we have chosen to use what we call a "layered language" approach to building systems. To solve a problem using this approach, one generalizes the problem to be solved into a class of related problems and then develops a "mini-language" for solving problems in this class. This new language has two components: (1) new data types and operations on the data types and (2) new control structures (abstractions). Collectively these tools should make the solution of problems from the original class easy. The resulting system consists of many language layers, each providing narrower, semantically richer mechanisms for describing the solution to a problem.

For instance, instead of building a single schematic entry system in SCHEMA, we have chosen to construct a language that makes it easy to construct graphics editing systems. This language includes spatial management data structures, icons and connectivity structures. New control structures exist for tracking the mouse, moving objects and synchronizing screen updates. This language is then used not only to build the schematic entry system, but also is the basis for the layout and waveform entry systems also.

The new data structures are often built using the Lisp Machine's "flavor system" and the new operations using "flavor methods" [17]. The multiple inheritance and sophisticated method combination mechanisms allow for greater sharing than many other approaches. New control structures are built using macros and functional arguments. A more detailed paper on this approach is in preparation.

3.2 Optimal Sizing of Transistors

Power consumption and signal delay are crucial to the design of high-performance VLSI circuits. Mark Matson and Lance Glasser have developed CAD tools for the modeling and optimization of digital MOS designs [14]. The tools determine the transistor sizes that minimize circuit power consumption subject to constraint on signal path delays. If a signal path constraint is unachievable then the tool returns the fastest possible design. Computational efficiency is obtained through macromodeling techniques and a specialized optimization algorithm. The macromodels are based on device equations and encapsulate logic gate behavior in a set of simple yet accurate formulas. The macromodels are valid for complex MOS gates with the limited use of pass transistors. Typical accuracies are in the 5–10% range. The optimization algorithm exploits properties of the digital MOS domain to convert the primal optimization problem into a dual form which is much easier to solve. It handles the case of multiple constraint paths and transistor size constraints. Not all constraints need be active. The result of this research is a pair of CAD tools that can optimize a circuit in roughly the amount of time needed to perform a transistor level simulation of the circuit.

3.3 Reliability Simulation

RELIC [2, 3] is a reliability simulator developed to analyze stress and wear in MOS VLSI chips. Unlike its predecessors, RELIC is not designed around any particular failure phenomena or model but around the idea of reliability simulation in general. RELIC is the first reliability simulator to encompass several different failure mechanisms. RELIC is designed to be used in the circuit design phase to identify devices which are under excessive stress and to determine the worst-case reliability of the circuit. It is also helpful in comparing the median time to fail (MTTF) of different circuits and in determining whether or not a part can be effectively screened by accelerated testing.

RELIC uses a simple methodology for analyzing the stress caused by many different failure phenomena. A device which is stressed over time accumulates wear. The probability of device failure at time t depends on how much wear the device has collected by time t and the critical value of wear for that failure phenomenon or for that circuit. This critical value of wear might be the amount of trapped charge a gate oxide can take before undergoing TDDB; it can also be the threshold shift of a transistor under hot electron stress which causes the circuit to malfunction.

RELIC provides a number of features to aid the user in performing reliability analyses. A circuit device may be analyzed for any number of failure mechanisms, and multiple tests may be run with variations of model parameters. The circuit designer has the option of analyzing the entire circuit for reliability hazards, or concentrating on a few critical devices. RELIC employs a circuit simulator so that the voltages and currents used in stress calculations are worst-case operating waveforms and not just the maximum voltages and currents. This feature will become increasingly important as devices scale to the submicron regime.

The present implementation of RELIC includes three failure phenomena: metal migration, hot electron trapping, and time dependent dielectric breakdown. The metal migration model predicts and includes the effects of thermal transients in the wire. RELIC has been used to analyze several circuits. The most illustrative example is a bootstrapped superbuffer. The simulator successfully determined that one of the transistors in the circuit was being stressed at a rapid rate by both hot electrons and time dependent dielectric breakdown. RELIC also verified that a redesigned version of the superbuffer had a lower wear rate and hence larger MTTF.

3.4 Waveform Bounding

The purpose of this project is to develop theoretical foundations and practical algorithms for a new approach to fast timing analysis and simulation of monolithic digital VLSI circuits. The method is based on easily computed bounds for transient node voltages and signal propagation delays. It is intended as an alternative to the two methods that are currently standard practice: "exact" numerical solution and approximate delay formulas. It is an attractive alternative in many cases because exact numerical solution requires prohibitive amounts of computer time for VLSI circuits, and approximate delay formulas can result in large uncontrolled errors for some practical circuits.

The starting point for this work is the original paper by Rubinstein *et. al.* [21]. That paper provided computationally convenient upper and lower bounds for the step response of linear RC tree networks, which are useful as network models for the branching interconnect lines in MOS IC's. We have extended the usefulness of these results by generalizing them to include interconnect networks with closed loops [26], such as the gate electrodes for large driver transistors. And we have proved theorems on the dynamics of RC networks that reduce the region of uncertainty in the original bounds. Tighter bounds have been achieved for all interconnect networks by exploiting slew rate limits on the node voltages, and further improvement was obtained for tree networks by using the spatial convexity of interconnect voltage during transients in a novel way [28,22]. Both these bound improvements have been successfully incorporated in a commercial CAD system for MOS standard cell design [23].

A project currently underway aims to extend the original work, intended for MOS applications, to include appropriate models for bipolar circuits [18]. The base-emitter junction in, e.g., ECL logic, provides a d.c. path to ground, not present in MOS, that invalidates the original bound formulas. The entire effort has been unified by the discovery [27] that the mathematical foundation of waveform bounding resides in the properties of linear dynamical systems governed by a special class of matrices, called "M-matrices."

A more broadly focussed and ambitious project is underway to develop a method for iterative bound relaxation, which will allow the user to flexibly trade off tightness of bounds for computer time as required at different stages in the design [31].

3.5 Network Extraction

A network extractor is a program which accepts an IC layout as input, and produces from it a machine-readable circuit representation, with sufficient information to perform detailed circuit simulation. The network extractor must be told about the fabrication technology and the meaning of the various layers; typically this is done through a "technology file" which is also treated as input. The output is typically in a form directly usable by a circuit simulator such as SPICE, but can also be used for layout verification, circuit schematic drawing, and electrical rule checking.

The network extractor EXCL has been developed by MIT by Steven P. McCormick and Jonathan Allen [15, 16]. The program has general extraction algorithms capable of accurate computations of interconnect resistance, inter-nodal capacitance, ground capacitance, and transistor size. However, where possible the general algorithms are replaced with simpler, faster techniques.

First, non-manhattan geometry is converted to staircase orthogonal geometry, since EXCL does not deal with arbitrary angles. Then comes a geometric decomposition phase, during which intersecting rectangles are grouped together, using technology-dependent intersection rules that are user-definable. Next is the extraction phase. Each transistor is transformed into its lumped equivalent. Each interconnect is also given a lumped representation. Separate algorithms for extracting capacitances and resistances are used. Again the extraction phase is user-controllable through technology-dependent extraction descriptions. Finally, the result is formatted for the desired use (logic simulation, timing simulation, circuit simulation, graphical presentation, etc.).

To extract interconnect resistance, EXCL uses three techniques. The most general technique works for arbitrary shapes, and solves Laplace's equation in two dimensions. The numerical techniques used are Gaussian elimination and successive over-relaxation. The second technique is valid for long straight wires with right-angle corners. This uses the standard formulas involving length and width, and standard corner corrections. The third technique uses a table lookup to handle commonly encountered geometry, such as tees, crosses, or vias.

To extract capacitance in the most general case is quite difficult. EXCL finds ground capacitance through calculations of areas and perimeters in a standard way. Coupling (internode) capacitance is calculated when EXCL judges that it will be significant. Three special techniques are used. For overlapping areas, the parallel-plate formula is used, with fringe correction. Parallel runs in the same or different layer are handled with a capacitance proportional to length, the constant of proportionality being a precomputed function of layer, separation, and technology. The third technique uses a lookup library to handle frequently occurring shapes. Besides these three special techniques, EXCL uses a Green's theorem approach for arbitrary geometry, if this is necessary.

3.6 Regular Structure Generation

A regular structure generator is a program which creates a VLSI layout of circuits which are repetitions of smaller circuits. Examples are n-bit adders which might be implemented as n full adders placed adjacent and wired together. Other regular structures include systolic arrays, multipliers, PLAs, and register files.

A regular structure generator has been developed at MIT by Cyrus S. Bamji and Jonathan Allen [BamjiRef]. This program does much more than merely place leaf cells adjacent to each other, since in practice regular structures always have special edge or end conditions, and for optimum performance the leaf cells should be personalized according to the size of the array. For example, in an n-bit adder, the LSB can be a half adder rather than a full adder because there is no carry in.

The RSG uses previously defined cells to hierarchically build larger cells. It accepts a library of leaf cells, a parameter file, and a design file, and produces a circuit layout. The design file is a parameterized, procedural description of the architecture. The parameter file contains the parameters for the particular design desired. Hierarchy is used to enable the RSG to deal with macrocells, which are built up from subcells using legal interfaces. RSG can effectively deal with reflections and rotations of layouts. The legal ways of placing, orienting, and connecting cells are represented in an interface table.

The RSG operates as follows. First, a sample layout is read. From this, the RSG determines the leaf cells and some of the legal interfaces. Then new cells are created by building a connectivity graph for the new cell, and the converting this into a layout. The result is a new cell. If this new cell is to be itself used in a larger cell, new interfaces must be created. This process is repeated as necessary to hierarchically build up the final layout.

The RSG is written in the language CLU and consists of approximately 6000 lines of source code. It has been used to create a variety of layouts.

The previous results have been proven feasible by implementation in the form of CAD programs, either within MIT or outside. The results mentioned in the remaining sections have not yet been incorporated into working code.

3.7 Retiming

The goal of VLSI design automation is to speed the design of a system without sacrificing the quality of the implementation. A common means of achieving this goal is through the use of optimization tools that improve the quality of a quickly designed circuit. A group of researchers led by Charles Leiserson have developed a class of optimization techniques for clocked circuits called *retiming*, that relocate registers so as to reduce combinational rippling [9, 10, 11]. Unlike pipelining, these techniques do not increase circuit latency.

These techniques are most appropriate for circuits with a relatively large number of clocked stages such as signal processing systems. This sort of problem can be represented as a weighted graph where the vertices are the computational elements, and the weights on the edges indicate the number of registers between computational elements. A delay associated with each computational element is included as a vertex weight. The minimum clock period is the maximum sum of vertex weights between any pair of registers. The usual *ad hoc* pipelining techniques insert additional registers between computational elements to decrease the clock period. The retiming techniques try to find an optimal placement of the registers by phrasing this problem as a particular integer programming problem that can be solved quickly.

The general ideas behind retiming provide a broad framework in which clocked circuit optimization problems besides clock period minimization can be considered, and it allows one to bring the powerful combinatorial optimization techniques to bear on these problems. Retiming seems to be a valuable technique that could be incorporated into both circuit and compilers and interactive design tools.

3.8 Wafer-Scale Wiring

VLSI technologists are fast developing wafer-scale integration. Rather than partitioning a silicon wafer into chips as is usually done, the idea behind wafer-scale integration is to assemble an entire system (or network of chips) on a wafer, thus avoiding the costs and performance loss associated with individual packaging of chips. A major problem with assembling a large system of microprocessors on a single wafer, however, is that some of the processors are likely to be defective. Thus a practical procedure for integrating wafer-scale systems must have the ability to configure networks around faults.

Recently, a group headed by Tom Leighton has developed improved and provably efficient algorithms for constructing two-dimensional systolic arrays on a wafer [6, 7]. Systolic arrays are a desirable architecture for VLSI because all communication is between the nearest neighbors.

In a wafer-scale system, however, all the nearest neighbors of a processor may be dead, and thus the prime advantage of adopting a systolic array architecture may be lost if a long wire connects electrically adjacent processors. In general, the longest interconnection between processors will be a bottleneck in the system. Of the many possible ways in which the live cells on a wafer can be connected to form a systolic array, therefore, the one that minimizes the length of the longest wire is most desirable.

The new algorithm for integrating two-dimensional arrays is based on a matching process. In particular, we find a matching between the functioning processors and points in an imaginary grid with evenly spaced rows and columns for which the length of the longest distance between matched points is minimized. Assuming independent cell failures, we prove that the longest matching length is $O(\log^{3/4} N)$ with very high probability. Initial experimental evidence confirms that the matching distances are quite small, and that the wire lengths in the resulting $N \times N$ array are within a factor of two of $\log^{3/4} N$.

The new algorithm is dramatically superior to row/column elimination approaches to wafer-scale integration (which fail for large N and/or high probability of cell failure), and moderately better than previously discovered divide and conquer algorithms. The algorithm requires roughly $O(N^2)$ time for N processors in the worst case, but appears to run much faster on average.

3.9 Compaction

An automatic compaction procedure is an effective tool for cutting production costs of a VLSI circuit at low cost to the designer, because the yield of fabricated chips is strongly dependent on the total circuit area. In order to perform any sort of compaction, the components of the layout must be differentiated into *modules*, which are fixed in size and shape, and *wires*, which are flexible. Common procedures for generating design rule constraints [4, 5, 12] assume that wires are simply rectangular regions of variable length, and otherwise identical to modules. A vertical wire, for example, would be assigned an x-coordinate during horizontal compaction, and could only be moved rigidly from side to side. But one would often like a previously straight wire bent around an obstacle during compaction, if the are of the circuit could thereby be reduced.

This problem is not easily overcome. Many systems [4, 25] attempt to solve it by allowing the designer to specify *jog* point at which wires my bend. Compaction then becomes an interactive procedure in which the designer repeated examines the compacted layout, adds more potential jog points and retries the compaction. Miller Maley, a student of Charles Leiserson has developed a new polynomial-time compaction algorithm that automatic introduces jog points in an optimal fashion [13]. Automatic jog insertion is achieved by treating wires not as solid objects, but only as indicators of the topology of the layout. Constraints between modules no longer express design rules directly; instead, they will ensure that there exist paths for the wires, having the given topology that satisfy the rules. The new constraints, called *routability conditions*, can be formulated as simple linear inequalities, and are easily solved. When the optimal placements have been established, the new wire paths are determined by a single-layer router [8]. This router does not generate "unnecessary U's," and therefore minimizes wire lengths, given that the layout topology is fixed.

Earlier work by Ron Rivest and his students introduced a theoretical approach to placement and routing. This work was embodied in a demonstration system called PI [19, 20], and is now being transferred to industry for practical implementation.

4. Conclusions

As with the larger MIT VLSI effort, we have stressed interdisciplinary approaches to VLSI CAD. This can be seen from the work described here: SCHEMA, which applies novel software engineering techniques to building a CAD frame; retiming, where integer programming ideas are applied to pipelined architectures; compaction, where we are using new combinatorial optimization techniques for jog insertion. The work on macromodels and reliabilty analysis is an example of tools that span the usual design boundaries. We are guided by the principle that the most challenging and needed work is that which crosses or even obliterates conventional design boundries.

We have summarized our most important recents results in VLSI CAD. Additional information on specific efforts may be found in the references.

5. Acknowledgments

This work was supported in part by the Air Force Office of Scientific Research, under contracts F29620–81–C–0054, OSR–82–0326 and F49620–84–C–0004; in part by DARPA under contract N00014–80–C–0004; in part by NSF grant ECS83–10941; in part by fellowships from the RCA Corporation, the office of Naval Research, National Science Foundation, IBM Corporation, Bantrell Foundation and Analog Devices; and in part by the Harris Corporation.

6. References

1. C. S. Bamji, *A Design by Example Regular Structure Generator,* M.S. thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February, 1985.
2. T. S. Hohol, *RELIC: A Reliability Simulator for Integrated Circuits,* M.S. thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, (1986).
3. T. S. Hohol and L. A. Glasser, "RELIC: A Reliability Simulator for Integrated Circuits," *IEEE International Conference on Computer-Aided Design,* Santa Clara, CA, November, 1986.
4. M. Y. Hsueh, *Symbolic Layout and Compaction Of Integrated Circuits,* Ph. D. thesis, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA, (1979).
5. G. Kedem and H. Watanabe, *Optimization Techniques for IC Layout and Compaction,* Technical Report 117, Computer Science Department, University of Rochester, September, 1982 .
6. F. T. Leighton and C. E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEE Trans. on Computers,* C–34(5), (1984), 448–461.
7. F. T. Leighton and P. Shor, "Tight Bounds on the Complexity of Minimax Grid Matching with Applications to the Average Case Analysis of Algorithms," *Proc. ACM Symposium on the Theory of Computing,* (1986).
8. C. E. Leiserson and F. M. Maley, "Algorithms for Routing and Testing Routability of Planar VLSI Layouts," *17th Annual ACM Symposium on Theory of Computing,* May, 1985.
9. C. E. Leiserson, F. M. Rose and J. B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *Third Caltech Conference on Very Large Scale Integration,* date of publication unknown, 87–116.
10. C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," *Journal of VLSI and Computer Systems,* **1**(1), Spring, 1983, 41–67.
11. C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Systems," date of publication unknown.
12. T. Lengauer, "Efficient Algorithms for Constraint Generation for Integrated Circuit Layout Compaction," *Proceedings of the 9th Workshop on Graph Theoretic Concepts in Computer Science,* January, 1984.
13. F. M. Maley, "Compaction with Automatic Jog Introduction," *1985 Chapel Hill Conference on Very Large Scale Integration,* Computer Science Press, Rockville, MD, (1985), 261–283.
14. M. D. Matson and L. A. Glasser, "Macromodeling and Optimization of Digital MOS VLSI Circuit," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.,* in preparation.
15. S. P. McCormick, *Automated Circuit Extraction from Mask Descriptions of MOS Networks,* M.S. thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February, 1984.
16. S. P. McCormick, "EXCL, A Circuit Extractor for IC Designs," *Proceedings of 21st Design Automation Conference,* Albuquerque, NM, June, 1984, 616–623.
17. D. A. Moon, R. M. Stallman, D. L. Weinreb, *LISP Machine Manual, Fifth Edition,* MIT Artificial Intelligence Lab., Cambridge, MA, January, 1983.
18. P. O'Brien and J.L. Wyatt, Jr., "Signal Delay in ECL Interconnect," *Proc. 1986 IEEE Int. Symp. on Circuits and Systems,* San Jose, CA, May, 1986, 755–758.
19. R. L. Rivest, "The PI (Placement and Interconnect) System," *Proc. 19th Design Automation Conference,* Las Vegas, June, 1982, 475–481.
20. R. L. Rivest and C. M. Fiduccia, "A Greedy Channel Router," *Proc. 19th Design Automation Conference,* Las Vegas, June, 1982, 418–424.
21. J. Rubinstein, P. Penfield, Jr., and M.A. Horowitz, "Signal Delay in RC Tree Networks," *IEEE Trans. Computer-Aided Design,* **CAD-2**(3), July, 1983, 202–211.
22. D. Standley and J.L. Wyatt, Jr., *Improved Signal Delay Bounds for RC Tree Networks,* VLSI Memo No. 86–317, Microsystems Research Center, Room 39-321, MIT, Cambridge, MA 02139, May, 1986 .
23. S. Teig, R. Smith, and J. Seaton, "Timing Driven Layout of Cell-Based IC's," *VLSI Systems Design,* May, 1986, 63–73.
24. J. P. Wade and C. G. Sodini, "Dynamic Cross-Coupled Bitline Content Addressable Memory Cell for High Density Arrays," *Proceedings of IEDM,* (1985).
25. J. D. Williams, "STICKS—A Graphical Compiler for High Level LSI Design," *National Computer Conference,* (1978), 289–295.
26. J. L. Wyatt, Jr., "Signal Delay in RC Mesh Networks," *IEEE Trans. Circuits and Systems,* **CAS-32**(5), May, 1985, 507–510.
27. J.L. Wyatt, Jr., C. Zukowski, and P. Penfield, Jr., "Step Response Bounds for Systems Described by *M*-Matrices, with Application to Timing Analysis of Digital MOS Circuits," *Proc. 24th IEEE Conf. on Decision and Control,* Ft. Lauderdale, FL, December, 1985, 1552–1557.
28. Q. Yu, J.L. Wyatt, Jr., C. Zukowski, H.N. Tan, and P. O'Brien, "Improved Bounds on Signal Delay in Linear RC Models for MOS Interconnect," *Proc. IEEE 1985 Int. Symp. on Circuits and Systems,* Kyoto, Japan, June, 1985, 903–906.
29. R. Zippel, *The Database Accelerator: Architecture,* Smart Memories Project, Report 1, (1985) .
30. G.C. Clark, R. Zippel, "Schema: An Architecture for Knowledge Based CAD," *Proceedings of ICCAD'85,* (1985), 50–52.
31. C.A. Zukowski, *The Bounding Approach to VLSI Circuit Simulation,* Kluwer Academic Publishers, July, 1986.

Highlights of CMU Research on CAD, CAM and CAT of VLSI Circuits

John Paul Shen

SRC-CMU Research Center for Computer-Aided Design
Department of Electrical and Computer Engineering
Carnegie Mellon University
Schenley Park, Pittsburgh, PA 15213

The primary goal of the Semiconductor Research Corporation-Carnegie Mellon University Research Center for Computer-Aided Design is to carry out a comprehensive research program that will result in the development of an integrated methodology and associated computer aids for the design, manufacture, and test of VLSI circuits and systems. Currently the research center has ten faculty members associated with it. There are also about forty graduate students currently pursuing research in the center as well as a number of visiting researchers. This paper outlines the ongoing projects in the center and highlights a number of these projects.

1. OVERVIEW

Three disciplines are involved in the realization of a VLSI circuit: design, manufacture, and test. Each of these disciplines involves a number of activities and requires the use of various computer aids. Because Computer-Aided Design (CAD), Computer-Aided Manufacture (CAM), and Computer-Aided Test (CAT) traditionally have been viewed as separate disciplines, the aids that have been developed for them are by and large unrelated and noninteractive. In our view, it is essential that an integrated approach be used for the design and implementation of computer aids for these three disciplines.

Taking into account the fabrication process, four major steps are involved in the realization of a VLSI circuit chip, namely process design, circuit design, nanufacture and test. Traditionally CAD tools are used to speed up the circuit design step and CAT tools are used during the test step to guarantee circuit performance. The SRC-CMU research center for CAD has a broad-based program aimed at developing an integrated approach to CAD, CAM and CAT. Our first step towards this integration involves the concepts of manufacturing-based CAD and manufacturing-based CAT; see Figure 1. Manufacturing-based CAD supports both process design and circuit design with the aim of achieving short design time, fast turn-around and high manufacturing yield. Manufacturing-based CAT utilizes technology and process information to effect accurate screening of defective chips during fabrication test.

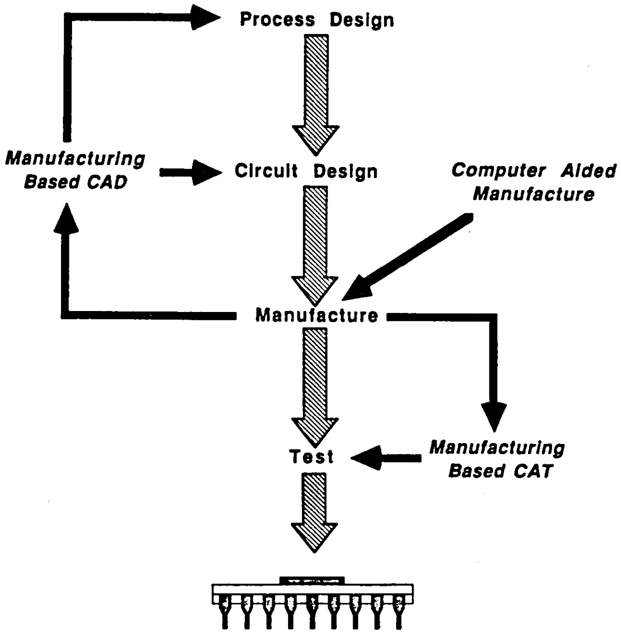


Figure 1. Integrated CAD/CAM/CAT - First Step.

Currently, the center has approximately forty ongoing projects, which are loosely grouped into the following areas:

- **Design**
  - Synthesis
  - Verification
  - Simulation
- **Manufacturing**
- **Testing**
- **Design Environment**

A representative, though not comprehensive, set of projects is described in this paper. Some projects are described in more detail to highlight recent results and future directions. This paper represents an abridged and updated version of an earlier paper [1]. Only a sample of references appears in this paper; a comprehensive list of references can be found in [1].

2. DESIGN

The goal of our CAD research is to develop a design automation system that involves a design methodology and a supporting set of computer aids that can take a behavioral description of the desired VLSI circuit, a description of the target technology, and a set of constraints, and then produce a set of masks that could be used to fabricate the circuit. The design produced should be testable and comparable in cost, speed, and reliability with manually produced designs. Availability of such a design automation system would significantly reduce the time required to design a VLSI system as well as produce designs which are correct the first time and manufacturable with acceptable yield. Figure 2 illustrates this overall objective.

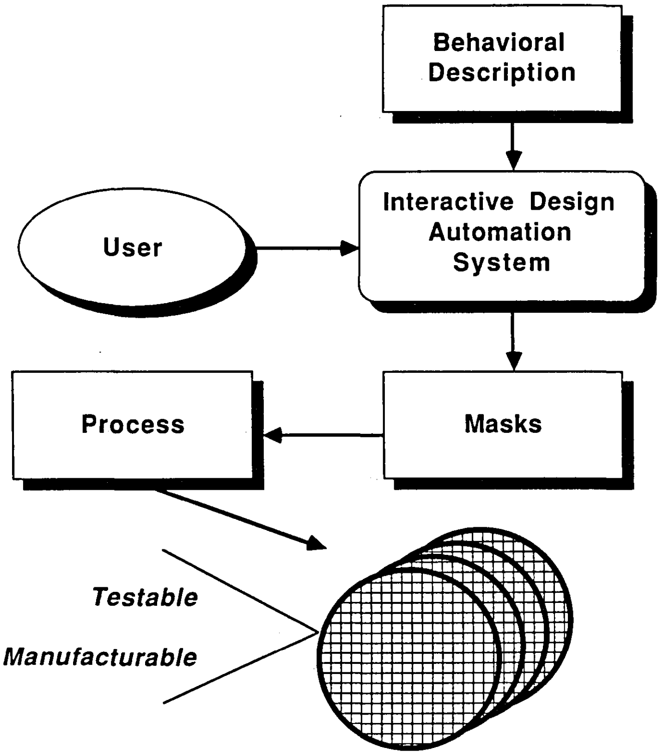


Figure 2. CAD Research Objective.

Our research activities have been aimed at developing new techniques to aid in a wide range of CAD tasks, and can be categorized around several main thrusts which include:

- An investigation into the use of Knowledge-Based Expert Systems (KBES) as an aid in synthesis and analysis.
- Development of top-down synthesis methods at the system and physical design levels which can make use of bottom-up information.
- Development of simulation tools to span all levels of the hierarchy.
- Development of a consistent multilevel representation to provide a means for our programs to communicate and as a basis for multilevel analysis aids such as multilevel simulation.
- Development of special purpose hardware for CAD tasks.

These research thrusts are being applied across the hierarchy of VLSI design. We view the design hierarchy to be composed of eight levels:

1. Task level
2. Architecture-definition level
3. Behavioral level
4. Functional-structure level
5. Logic-structure level
6. Transistor or switch level
7. Mask layout level
8. Process level

These levels are described in detail in [1]. Design can be viewed as involving synthesis, verification and simulation. Ongoing CAD research projects in these three areas are described.

2.1. SYNTHESIS

A synthesis tool takes a description of a desired system at one level in the design hierarchy and generates a description at the next lower level in the hierarchy. A number of synthesis tools have been developed or are under development now.

2.1.1. System Synthesis

Design Representation and the System Architect's Workbench

R.A. Walker, D.E. Thomas

The objective of this research is to develop a model of design representation, and then use that model to provide a designer's workbench that will allow manipulation of designs at high levels of abstraction. The model of design representation being developed is characterized by three separate but coordinated domains of description - the behavioral domain, the structural domain, and the physical domain - each characterized by a particular design decision. Using the SCS language, developed at CMU, we can coordinate these domains to represent the entire design. We are also developing a system architect's workbench that uses this model of design representation, and that will allow a designer to transform and analyze the design, particularly in the behavioral and structural domains. This workbench will extend our current system to higher levels of abstraction, and will include such transformations as structural partitioning, process creation, and support for instruction-level pipelining.

Interface Synthesis

J.A. Nestor, D.E. Thomas

This research investigates the description and synthesis of digital systems with complicated interface requirements. Previous work in high level synthesis has successfully developed tools that can automatically generate designs from an abstract specification. However, these tools have neglected the issues involved in designing interfaces. A method of describing systems with interfaces has been developed. A new system of tools building on previous research uses these extended descriptions to automatically generate designs with interfaces. The system of tools is nearly complete. Currently, we are evaluating its effectiveness with several examples of real interface specifications.

Compilation of Interprocessor Communication for Multiple Processor Systems

R.P. Bianchini Jr., J.P. Shen

Recent research on parallel systems has clearly shown that the most difficult problem faced by the system designer is interprocessor connection and communication. A methodology for the automated design and implementation of interprocessor communication for special-purpose multiple processor systems has been developed. It is shown that for many special-purpose and mission-oriented systems the interprocessor communication is deterministic and can be specified at system inception. This specification, identified as the target architecture, can then be automatically mapped or compiled onto a physical multiple processor system, identified as the host architecture, using a network traffic scheduler. Algorithms for such a scheduler have been implemented. It is shown that the order of complexity of network scheduler components is polynomial rather than exponential as in classical solutions.

Towards Knowledge-Based Synthesis of Analog Circuits

R. Harjani, R.A. Rutenbar

Although analog circuits do not exhibit the same rigid hierarchical structure as digital circuits, hierarchy maybe exploited by analog designers. Design still proceeds at different levels of abstraction, but the coupling between levels and between objects within a level is much stronger, and the interactions that must be planned for and designed (i.e., not logic values, but electrical, parasitic, electromagnetic and thermal considerations) are richer and more varied. Analog designers rely strongly on experience with partial circuit topologies that meet classes of constraints, and knowledge-intensive strategies for choosing the parameters that define each module or primitive device. We are investigating design representations for a knowledge-based analog synthesis tool in the restricted domain of CMOS analog-to-digital converters. The goal is to develop a representation that will not limit design to a set of fixed modules, but will help design fully custom analog circuits.

2.1.2. Logical Synthesis

Synthesis by Delayed Binding of Decisions

J. Rajan, D.E. Thomas

A program called SUGAR is currently being developed to investigate the automatic synthesis of single chip microprogammed microprocessors from behavioral descriptions. Subtasks in complex problem solving activities like synthesis often interact. As a result, decisions made prematurely can lead to poor designs. A least-commitment style of binding decisions has been developed that allows better design choices to be made by postponing important decisions until adequate information is available to make them. This design style partitions the design task into a sequence of temporally ordered subtasks that cooperate during design. The subtasks consist of flow analysis, control flow transformations, control flow partitioning, data path model selection, and hardware allocation. Most subtasks have been implemented and the project is nearing completion.

A Knowledge Source Based Synthesis System : A Feasibility Study

D.L. Springer, D.E. Thomas

This project investigates the organization of programs to automatically synthesize VLSI systems from the behavioral level to the register-transfer level. In particular we are exploring knowledge source based systems and an organization structuring language. Knowledge source based systems are composed of a set of independent knowledge sources which communicate via a software blackboard. Present knowledge source based systems include Hearsay-II which is a speech recognition program and ULYSSES (discussed below) which is a design environment for the logical and physical design of VLSI systems. Present synthesis systems for the upper levels of the design hierarchy do not allow the flexibility to: change knowledge, integrate different implementation styles, or deal with conflicting data, which a knowledge source based system would provide. This flexibility is highly desirable since this domain of the synthesis task is still largely in the research stage.

Instruction Processor Pipeline, Design and Analysis

R.J. Cloutier, D.E. Thomas

The design of the stages and the control structures of a pipeline, for an instruction processor, are weakly supported by existing tools. The manual methods used today are similar to those that were originally used to describe the IBM 360/91. This lack of tools is hindering the development of advanced pipelined processors. The major problem is the diversity of control structures and the inability to deal with them in computer aided tools. Take for example, the partitioning of a processor into stages. Stage boundaries are proposed by considering timing relations and resource conflicts. A simulator is then written for that particular partitioning and the proposed control structure. Potential instruction streams are then run through the new pipeline model. The simulator is used to search for unexpected delays caused by specific combinations of instructions and external signals (such as interrupts) flushing the pipeline. Instead of simulation, analysis can be performed to find the combinations of instruction sequences and signals that will cause problems. Combining this analysis with synthesis tools avoids the problem of building a unique simulator for each proposed pipeline configuration.

2.1.3. Physical Synthesis

Logic Synthesis System

M. Trick, S.W. Director

The purpose of this research is to develop a system that takes a structural level specifications of a system and generates the necessary layout. It is our intent that this synthesis system will allow a structural synthesis tool or the designer as much freedom as possible in specifying the structure of the circuit. It will be able to handle a variety of interconnection patterns, clocking schemes, and functional blocks. This freedom puts an additional burden on the synthesis system because it must analyze the connectivity and the shapes of the modules in the circuit and choose appropriate physical implementation for the modules. Many chips can be implemented very efficiently as a single row of modules of the data path connected to a logic array which implements the control of the data path as was done in Bristle Blocks and MacPitts. The layout synthesis system should recognize such designs and implement them in that style, but if a design is specified that contains modules or interconnect not amenable to layout in a single row of modules, the system will be able to perform the layout of the IC using an appropriate style.

A Multiple Expert System Approach for Placement

M. Hirsch, D.P. Siewiorek

It is recognized that the activities performed during physical design fall into three main categories: partitioning, placement and routing. These three categories of activities are deeply interrelated in that if one considers any without considering the others, then the results will most likely be unacceptable. Existing tools approach the problem from a single point of view. For instance, min-cut will view physical design from a top-down point of view, whereas some graph-theoretic techniques which cluster components view the problem from the bottom up. It is believed that human designers attack the problem from a multiplicity of viewpoints. That is, the viewpoint of the problem dynamically changes between bottom up, top down and middle out as the solution to a physical design problem is refined. It is also believed that while human designers use metrics such as congestion estimates, number of wire crossings between partitions and balance of area between partitions, these metrics are only guidelines. Human designers also look for recognizable design situations such as busses, locations of pads on the periphery of the chip and familiar interconnections of components and handle these situations in a manner that their experience has shown to be most effective. A new approach will be investigated which combines a variety of metrics and viewpoints and will be implemented with a multiple expert system.

Floorplanning Using Simulated Annealing on a Multiprocessor

R. Jayaraman, R.A. Rutenbar

The floorplanning task for VLSI chip layout determines the relative placement of circuit macro modules on a silicon surface. The basic objective is to optimize critical layout parameters, for example, interconnection length, total chip area, and bus placement. This project studies simulated annealing algorithms as applied to floorplanning. Annealing algorithms applied to physical design problems have yielded excellent solutions, but they are computationally very intensive. In this research we are studying the implementation of an annealing-based floorplanner on a multiprocessor. Our goals include floorplanning for modules with a range of alternative shapes, modules with varying degrees of constraint on their final position in the floorplan, and explicit consideration of busses. After completing the serial version of the proposed floorplanner, this research will focus on identifying parallelism and multiprocessor partitioning strategies appropriate to speed up the execution times.

Multiprocessor-Based Placement by Simulated Annealing

S.A. Kravitz, R.A. Rutenbar

This project is developing multiprocessor-based standard cell placement tools based on simulated annealing algorithms. Placement of standard cell ICs by simulated annealing produces results of high quality, but at a very high cost. The goal of this research is to reduce the high cost of the iterative improvement phase of annealing-based placement by mapping the algorithm onto a shared-memory multiprocessor. The key issue in parallel tool implementation is the partitioning of an algorithm across communicating processors. We have developed a taxonomy of possible multiprocessor decompositions of simulated annealing algorithms. This provides a systematic framework for extracting parallelism in annealing-based layout algorithms. In particular, we have shown that the choice of multiprocessor annealing strategy is strongly influenced by the annealing temperature parameter. We introduce the idea of *adaptive strategies* which change dynamically as annealing proceeds through different temperature regimes to best exploit parallelism. Three implementations of multiprocessor standard cell placement have been completed and their performance evaluated. An adaptive strategy which switches from one parallel decomposition to another at the optimal temperature yields speedup significantly better than any single strategy approach. Practical speedups for benchmark problems have been achieved for a four processor machine.

2.2. VERIFICATION

Linking Design Representations

R.L. Blackburn, D.E. Thomas

This research is concerned with the development of a means for linking design representations at different levels of abstraction. The linkage should provide the means of identifying corresponding values and operators in two or more different representations. Linkage will be accomplished by adding additional information to design representations to describe how they relate to each other. This work has applications in interactive debugging and formal verification of synthesized data paths and controllers.

A Data-Structure Processor for VLSI Geometry Checking

E.C. Carlson, R.A. Rutenbar

A critical step in the verification of VLSI chip designs is the geometric analysis of IC masks. This geometric analysis performs the functions of design rule checking, connectivity analysis, transistor identification and parasitic extraction from the integrated circuit mask geometry. Scanline algorithms are at the base of many geometric analysis functions, and appear to be the best available solution both in speed and memory requirements to the needs of these verification tasks. Bigger machines to run these compute-intensive programs are an obvious solution, but are not necessarily the most cost effective. The design of special-purpose hardware is attractive since much of VLSI mask verification is based on similar scanline primitives. We have proposed an architecture that implements two fundamental scanline operations: boolean operations between mask layers, and region numbering within a mask layer. The proposed architecture is a pipeline that implements directly the basic operators needed to manipulate a scanline data structure. Using edges as its primitive operands, the architecture is capable of handling arbitrary polygon geometry. Preliminary simulations of the proposed hardware suggest that it is significantly faster than a VAX 11/780.

A Localizing Circuit Extractor

M. Chew, A.J. Strojwas

Process disturbances during the fabrication of a digital integrated circuit might affect the equivalent electrical circuit of the resulting chip. Unfortunately, the circuit extraction of the complete layout of a digital circuit layout is too computation-intensive a task to perform if one wants to see the effects of a given set of process disturbances. The goal of this project is to develop an extractor which, given the disturbance region and the type of the disturbance, will extract from the altered layout only the electrical components which might have undergone a change. The results of this tool will then be used to update a previously extracted circuit of the undisturbed layout.

2.3. SIMULATION

Simulators developed at CMU span most of the levels in the design hierarchy. One trend in the design of CMU's simulators is the use of abstractions for controlling detail. Abstraction can define a new level which approximates the details of a lower level but with simulation cost of a higher level. An example is WASIM, a waveform simulator that allows device and circuit properties at the cell level.

Another trend is the use of hierarchy. Hierarchy can be used within a single design level by taking advantage of the natural hierarchy of the system under design. Hierarchy can also be used to mix two or more design levels. SAMSON, a mixed-level simulator, bridges the circuit and logic levels with more than two orders of magnitude performance increase over a traditional circuit simulator.

A third trend is the increased use of statistical techniques. FABRICS uses statistics to model process variations. The Hierarchical Statistical Simulator summarizes detailed behavior with a probability distribution function. A fourth trend is the use of symbolic and compile-driven techniques. An example of this trend includes the switch level simulator COSMOS which is currently being developed at CMU. A number of simulation projects are described below in roughly the top-down order.

2.3.1. Functional Level Simulation

Functional Specification Simulation of Computers

S. Bose, D.P. Siewiorek

This project is concerned with the functional description of computers. All constructs of a computer system can be based on such functions, which can be written and incorporated into a general purpose language. Lisp has been used for this purpose. A pseudo language called the Functional Specification Language (FSL) was designed to incorporate the features of functional languages as well as special constructs for Hardware Description Languages (HDL). FSL is actually Lisp with an added set of functions. These functions are arranged in two layers. The upper layer functions build on the lower layer and capture the unique features of HDLs for describing target computer instruction sets. The development of a tool is the subject of the ongoing research.

2.3.2. Switch-Level Simulation

COSMOS: A New Switch-Level Simulator

R.E. Bryant

The COSMOS (COmpiled Simulator for MOS) project aims to produce a high quality switch-level simulator with at least an order of magnitude better performance than MOSSIM II. The program also includes additional capabilities such as concurrent fault simulation. Furthermore, it can easily be adapted for execution on any special purpose simulation accelerator that supports Boolean evaluation. Unlike programs that operate directly on the transistor level description during simulation, COSMOS transforms the transistor network into a Boolean description during a preprocessing step. This Boolean description, produced by a symbolic analyzer, captures all aspects of switch-level networks including bidirectional transistors, stored charge, different signal strengths, and indeterminate logic values. Unlike previous symbolic analyzers for MOS circuits, the COSMOS preprocessor produces very concise formulas, generally linear in the circuit size.

Gaussian Elimination of Boolean Equations for Symbolic Simulation of MOS Transistor Networks

T.J. Sheffler, R.E. Bryant

This project is responsible for solving the network equations symbolically to arrive at Boolean equations which represent the behaviour of the network. A method closely related to Gaussian elimination seems promising. Instead of solving the equations in the domain of real numbers, the elimination and backsolving steps will be performed in the domain of Booleans. Much has been written on the relation between Gaussian elimination and solving systems of Boolean equations, however, there has been no clear discussion of this topic with respect to switching circuits. The systems of equations to be solved in this application are sparse in nature, thus sparse matrix techniques will be explored.

Manipulation and Reduction of Boolean Expressions Represented as Directed Acyclic Graphs

K.S. Brace, R.E. Bryant

This project deals with the maintenance and simplification of structures which store representations of Boolean formulas. A formula is represented as a directed acyclic graph (dag). A dag resembles a parse tree except that a subgraph may be shared by more than one branch, yielding a more compact representation. When a new formula is defined simplifications must be made in a standardized way so that equivalent formulas may be recognized as such. The shared subgraphs of the dag structure complicate this. The primary concern of this project deals with the trade-off between the computation involved in simplification and the time and space saved by having an optimally simplified dag.

2.3.3. Timing/Circuit Simulation

SPECS2 : A Timing Simulator

C. Visweswariah, R.A. Rohrer

This project is aimed at the development of a timing simulator called SPECS2. The timing simulator is based on a tree/link analysis of the circuit. Table look up is used for model evaluation. An event-driven approach is employed for simulation. An event is the movement of a device from one region of its characteristics to the next. An event is processed, generating a new event for the time when the device is expected to move out of the present region of its characteristics. This event is scheduled for the future. Time moves ahead to the next scheduled event and the process continues. A new modeling approach based on the conservation of charge and energy is used to build the table models. The simulator does not rely on complex models for good numerical conditioning or to guarantee reliability.

Statistical Timing Analysis

J. Benkoski, A.J. Strojwas

The purpose of this project is to develop a tool that will provide the designer with a fast but accurate statistical estimate of the delays in VLSI circuits. At first the circuit is partitioned into logically meaningful blocks. Each block will then undergo several E-logic (Electrical Logical) simulations in order to fully characterize its delay as a function of the process parameters. The result of this analysis will be the input to output delays for this block and will be decomposed into three sub-delays: the input delay, the intrinsic delay, and the output delay. While the first is a function of the input slopes and the last depends on the loading conditions, the intrinsic delay is independent of the environment in which the block is placed. By using this technique, we can afford to analyze only unique blocks. The last part of our estimation process involves either an event driven simulation of the circuit at the block level or a longest path evaluation.

A Macromodelling Approach for VLSI Simulation

S.R. Nassif, S.W. Director

We have developed a methodology for the simulation of cell-based MOS designs. We attempt to exploit the regularity of such designs by building macromodels of the individual cells which are capable of predicting circuit-level electrical performance. In particular, we derive analytical approximations for waveforms that are generated by typical device-level output stage topologies, and fit these models to waveforms generated by circuit simulation. Macromodels are generated for these cells, such that the output of these macromodels are waveforms, and the inputs are designable parameters such as layout, as well as circuit parameters such as fan-out. An event-driven, waveform-based simulator, called WASIM has been developed which employs these macromodelled cells. The basic events in this simulator are waveform transitions. WASIM is capable of achieving near logic-simulator speeds, while accurately predicting waveforms, delays, and hazards [2].

Efficient Modeling of Small-Geometry MOSFETs

M.J. Saccamango, A.J. Strojwas

This project is concerned with developing more accurate models of small-geometry MOSFET's for circuit simulation purposes. We are currently deriving methods for determining device parameters (threshold voltage, body factor and channel-length modulation parameter) with efficient physically-based models. These methods will be a part of the device simulation portion of the statistical process/device simulator FABRICS II. This phase of the project is focused on the special modeling problems of modern MOSFET devices, i.e. those with small-geometries, nonuniformly-doped channels and/or lightly-doped source/drain regions. The next project phase will involve the use of the physically-based models to extend the FABRICS II device simulation to produce I-V characteristics. From these characteristics we hope to extract device models for circuit simulation which better reflect the bias dependencies of the device parameters. The availability of these I-V curves will also allow the extraction of device parameters to meet the modeling requirements of many circuit simulators presently in use.

Tree/link Based Mixed-mode Circuit Simulator

X. Huang, R.A. Rohrer

This project aims at transcending the borders between switch level simulation and circuit level simulation of VLSI. A mixed mode capability will be accomplished by modeling the circuit devices on variable levels of accuracy. Spatial and temporal latency can be exploited by replacing dormant devices with the simplest models, which very often may be ideal switches. To accommodate these different models, tree/link repartitioning will be employed to avoid ill-conditioning and to effect fast convergence of iterative solution.

Table Look-Up Device Modeling for a Charge and Energy Conserving VLSI Circuit Timing Simulator

C. DellaTorre, R.A. Rohrer

Conventional simulators violate basic laws of physics, such as the conservation of charge and energy. A new method of device modeling has been proposed that is aimed at charge and energy conservation in circuit simulation. This project involves the development of this modeling effort so that it could be incorporated into a simulator, the premise being that this simulator would, therefore, be more reliable and efficient than existing simulators. This modeling effort will incorporate table look-up methods of device modeling consistent with the above constraints. Thus, values entered into these tables will be derived in a manner that ensures that charge and energy are conserved. These tables will be built in the most memory-efficient manner possible.

2.3.4. Yield Prediction and Maximization

Yield Simulation for Integrated Circuits

H. Walker, S.W. Director

We have developed a catastrophic fault yield simulator VLASIC (VLSI LAyout Simulation for Integrated Circuits). VLASIC is a Monte Carlo simulator that uses defect models and statistics to place random catastrophic point defects on a chip layout and determine what circuit faults, if any, have occurred. The defect models are described in tables, and so are readily extended to new processes or defect types. The defect statistical model is based on actual fabrication line data, and has not appeared before in the literature. The output of VLASIC is a population of chip samples with the same distribution of circuit faults as observed in the fabrication line. This circuit fault information can be used to predict yield, optimize design rules, generate test vectors, evaluate redundancy, etc. We have also developed a simple redundancy analysis system as an example application.

A Realistic Yield Simulator for IC Failures

I. Chen, A.J. Strojwas

This research is aimed at developing a CAD tool to realistically estimate the yield losses due to IC structural failures. This tool extends the statistical design rule developer to an accurate and efficient yield simulator. The methodology adopted by the simulator accounts for most of the mechanisms which cause IC yield degradation due to structural failures. For computational efficiency, analytical method rather than Monte Carlo method is used. There are two main parts in this simulator: formulation of yield statistics in hierarchical levels and derivation of the Probability Of Failure (POF) for layout patterns. In the first part, we have successfully derived yield statistics which consider the effects of defect size sensitivity, globally nonuniform distribution (wafer level), and local clustering effects. In the second part, a theoretical method to find the POF for most of the simple patterns has been developed. Thus, given a set of layout design rules, this simulator will be able to predict yield in a hierarchical manner on the device level, chip level, or even wafer level.

Statistical Integrated Circuit Design: Parametric Yield Maximization

K.K. Low, S.W. Director

The goal of this project is to develop an efficient way to optimize the production yield of VLSI circuits using statistical circuit design techniques. Specifically, we will address the parametric yield maximization problem using fabrication process control parameters (such as implantation dose and energy) and device geometries as designable parameters. Our approach differs from most previous work in this area in that we have chosen the space of process inputs instead of device parameters as part of our design space. This distinction is particularly relevant in integrated circuits since the device parameters are correlated. Moreover, previous methods require separate optimizations to be performed at the process and circuit levels which may result in suboptimal solutions. Two major obstacles can be identified in our effort. Firstly, accurate yield estimation is very computationally expensive due to the cost of simulations. Therefore, techniques must be found to reduce the cost without compromising accuracy. Secondly, the dimensionality of the problem must be brought down to a manageable level. This may be achieved by exploiting certain properties of integrated circuit design such as regularity and hierarchy.

3. MANUFACTURING

Minimizing cost-per-chip is the main objective in VLSI design, manufacturing and testing. Our research program in the area of manufacturing has this goal as its principal objective. We have recognized the fact that in VLSI circuit design, statistical characterization of the fabrication process is a necessity in order to describe the random fluctuations in processing conditions, material parameters and point defects that cause drops in production yield. This process level information can be abstracted into the higher levels, i.e. circuit, logic or functional levels, where it can be used to produce realistic IC designs or test programs.

Towards this end the statistical process/device simulator FABRICS II was developed to provide a common denominator for the integrated CAD/CAM [3] system with the capabilities shown in Figure 3. FABRICS II employs efficient (numerical or analytical) models of the fabrication steps and semiconductor devices manufactured in a number of technologies (NMOS, CMOS and bipolar). FABRICS simulation system accepts IC layout and process description, and generates model parameters for the active devices and parasitics. These parameters are compatible with the models implemented in such a popular circuit simulator, as SPICE. The CPU time consumed by FABRICS for the process/device simulation is usually a small fraction of the time consumed in the circuit simulation stage. The accuracy of simulation is accomplished by tuning FABRICS to a particular fabrication process by a statistical optimization system called PROMETHEUS [3]. As a result of tuning, device parameters obtained from FABRICS are in good agreement with the parameters measured in the real fab lines. The current status of software in the CAM system is shown in Figure 4.

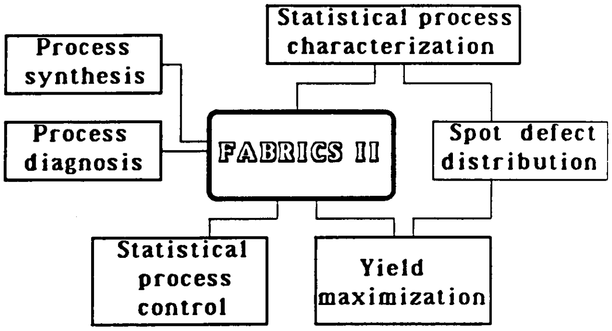


Figure 3. CAD/CAM System Capabilities.

To facilitate synthesis of new fabrication processes, an interactive process interpreter/compiler PI/C [3] has been recently added to the simulation system. It employs a graphical process editor PED [3] and allows the user to specify the process by defining the lithographic masks and process conditions. The user can perform incremental simulations and view the simulation results which are displayed in the form of impurity profiles and device cross-sections. Extensive error checking is performed by the system to help in the synthesis.

FABRICS II combined with a circuit simulator (SPICE or SAMSON) can be employed for a number of tasks ranging from the design verification/optimization to process diagnosis, to statistical process control. The design optimization tasks include:

- Nominal design in terms of process parameters (e.g. minimization of power-delay product of a dynamic RAM in terms of gate oxidation parameters).
- Worst-case design in terms of statistically independent process disturbances.
- Parametric yield maximization in the space of layout and process parameters.

A complete integrated system for process diagnosis PROD [4] has been developed and it uses FABRICS for efficient fault simulation. PROD identifies faults in the process that caused a drop in yield for a particular production run. The approach used in PROD is based on the analysis of distributions of IC performances and employs statistical pattern recognition techniques.

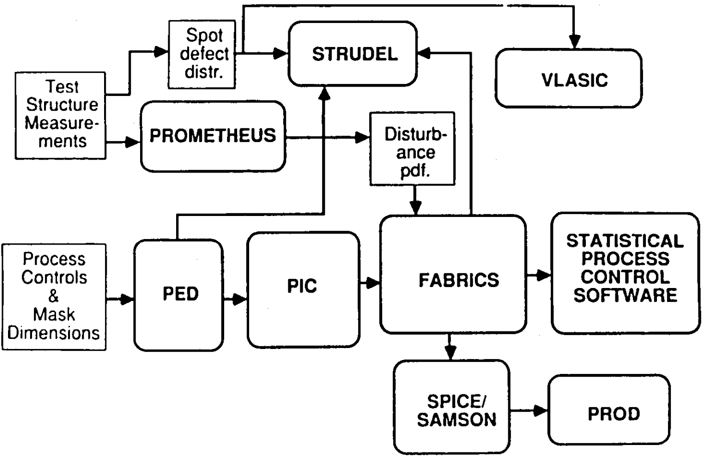

Spot defect distr.
STRUDEL
VLASIC
Test Structure Measure-ments
PROMETHEUS
Disturb-ance pdf.
Process Controls & Mask Dimensions
PED
PIC
FABRICS
STATISTICAL PROCESS CONTROL SOFTWARE
SPICE/ SAMSON
PROD

Figure 4. CAD/CAM Software.

To obtain realistic estimates of final production yield, the probability distribution of spot defects (e.g. shorts or breaks due to local lithographic errors) has to be taken into account together with the parametric variations. A yield simulator, called VLASIC, has been developed and can be used for prediction of the decrease in yield due to spot defects. For circuit design purposes, all the random factors affecting IC layout have to be translated into *layout design rules*. A statistical design rule developer STRUDEL [5] has been implemented to derive rules which maximize total yield. Two ongoing projects in this area are described below.

Statistical Quality Control for VLSI Fabrication Processes

P.K. Mozumder, A.J. Strojwas

Random fluctuations, which are inherent in any IC fabrication process, cause the production yield to drop significantly below 100%. The faults which cause the yield drop can be classified as catastrophic and parametric. This project deals with monitoring and controlling the IC fabrication process via statistical quality control. A cost optimization approach to the statistical quality control of IC fabrication processes has been taken. The project aims at creating a system which will, after every step in the fabrication sequence, make decisions as to whether the lot as a whole or in part is to be rejected from further processing. The system, given the constraints, is expected to develop selection criteria for the in-line measurements, which are then used by the system to make quality control decisions. The concept of process observability via in-line measurements, appropriate models necessary for process control, and algorithms for quality control based on yield maximization objectives have been developed.

Adaptive Control of the IC Fabrication Process

C.R. Shyamsundar, A.J. Strojwas

This research deals with adaptive control of VLSI fabrication processes. Based on the in-line measurements that are made during the fabrication of integrated circuits, decisions will be made regarding the values of the future process parameters. Models relating the process parameters to in-lines and circuit performances, will be used for this purpose. These models were built using MULREG (MUlti-Layer REGression program), which has been specially developed for this purpose. An acceptability region in the space of process parameters is also used to arrive at the decisions. The aim will be to maximize the total profit from the fabrication line. With this in mind, a minimum acceptable yield will be determined at each stage. This will not be a constant but will depend on a number of factors including the current stage in the fabrication process. If at a particular stage in the IC fabrication process, the predicted yield is greater than this acceptable yield then the process parameters for the future stages will not be modified, otherwise one or more of these process parameters will be modified so as to meet the yield criterion.

4. TESTING

We have started a research effort to develop a new methodology and associated software tools for the testing of MOS VLSI circuits. Most of the existing fault modeling and test generation techniques are based on the traditional scenario of putting many TTL SSI/MSI packages on a printed circuit board. It is believed that the efficient and effective testing of MOS VLSI circuits requires the development of new approaches and tools. We believe the following three problems associated with traditional testing must be addressed.

1. **Technology independent fault model.** Regardless of the implementation technology, the single line stuck at fault model is the most widely used fault model. We believe that, for VLSI testing, the fault model should be based on the physical defects and should take into account the technology, layout, and process characteristics.
2. **Unranked fault list.** Traditionally, the relative importance of different faults is not considered. We believe that not all faults are equally likely or equally important. It may be more cost effective to rank the fault list and appropriately order the test set so as to detect the most likely to occur faults first.
3. **Single level approach.** Most existing testing methodologies and tools assume the logic gate level model. Although MSI level macros are often used in logic simulators, most systematic test generation algorithms require the gate level representation. We believe, just as in design, a new testing methodology should involve a multiple level or hierarchical approach in order to ease the complexity burden.

As part of our initial step towards the integration of CAD/CAM/CAT, the concept of a manufacturing-based methodology for fabrication testing is being developed. This methodology addresses the above three problems in the following two ways:

- A customized fault list is generated for each circuit being tested. Faults in the list take into account the technology, layout, and fabrication characteristics, and can be grouped into different types depending on the faulty behaviors. Each fault list can be ranked according to relative likelihood of occurrence of the faults.
- Starting with likely defects at the physical level, faults are extracted to the transistor, logic or even functional levels. Test pattern generation is performed separately for different fault types, at possibly different levels. Test pattern composition is then performed to produce the final test set.

The proposed manufacturing-based testing scenario is illustrated in Figure 5. Two ongoing projects in this area are presented below.

Inductive Fault Analysis

F.J. Ferguson, J.P. Shen

Our first project in this area addresses the problem of the inappropriateness of the traditional fault model. Our underlying assumption is that circuit level faults are caused by physical defects which are inherent in the IC fabrication process. A technique called Inductive Fault Analysis (IFA) has been developed which allows the transistor or logic level faulty behaviors of a given MOS integrated circuit to be extracted from the layout level using process level information concerning spot defects [6].

IFA is an example of crossing multiple levels of the design hierarchy and involving elements of CAD, CAM and CAT. It is a systematic fault extraction procedure that spans the process, layout, transistor, and logic levels. A higher level fault model is developed based on conclusions

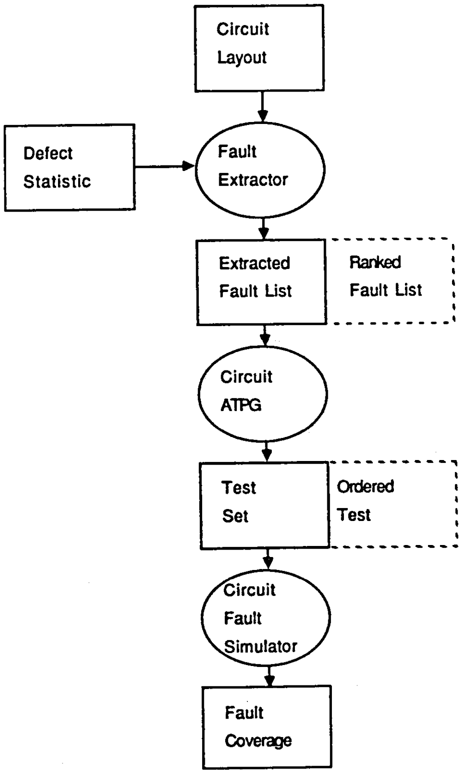


Figure 5. Manufacturing-Based Testing Scenario.

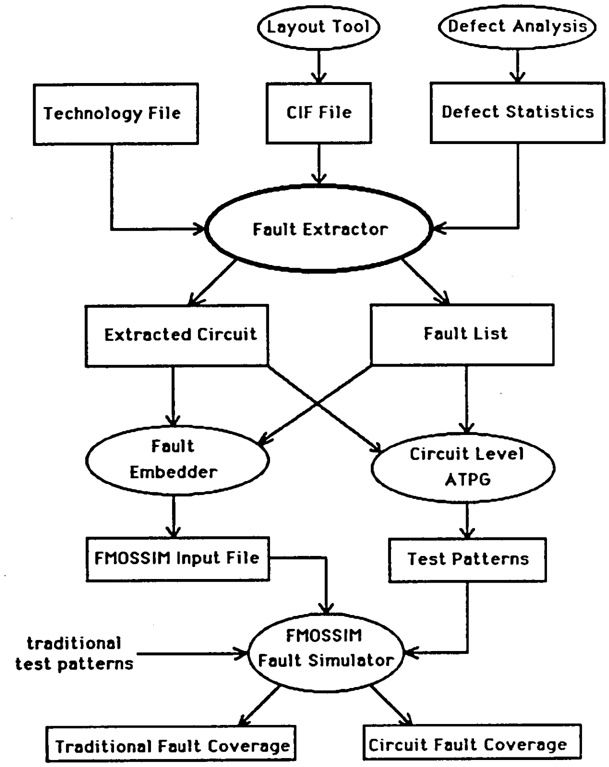


Figure 6. Inductive Fault Analysis Implementation Environment.

drawn from examining particular defects at the lower level, hence the word "inductive".

The four major steps of the IFA procedure are: (1) generation and placement, on the circuit layout, of physical defects using statistical data obtained from the fabrication process; (2) extraction of primitive (geometric) faults caused by these defects; (3) abstraction of these primitive faults to the transistor, logic or even the functional level; and (4) classification of fault types and ranking of faults based on the likelihood of occurrence. Hence, given the layout of an integrated circuit, a customized and very accurate fault model and an associated ranked fault list can be automatically generated which take into account the technology, layout, and process characteristics. A CMOS fault extractor has been implemented. Currently a number of example circuits are being analyzed. The IFA implementation environment is as shown in Figure 6.

Initial application of the IFA procedure to an example circuit led to a number of very interesting observations [6]. Only 64% of the extracted faults can be modeled by the traditional single and multiple line stuck-at fault model. Not all faults are equally likely, and unusual faults, such as the creation of new transistors, are possible. Only 3% of the faults involve transistors stuck at open, whereas 30% of the faults are bridging faults. The IFA procedure can be used to (1) characterize potential faults in a circuit; (2) evaluate the true effectiveness of traditional test sets; (3) provide a basis for more effective test generation; and (4) uncover likely faults that are very difficult to test.

Switch-Level Test Generation

S.H. Robinson, J.P. Shen

This project focuses on automatic generation of test patterns to screen out defective CMOS integrated circuits during fabrication testing. The algorithm developed uses a switch-level model in a search, similar to the D-Algorithm, to find tests for a specified fault list. Backtracking is allowed at each step in the search where multiple alternatives exist, so that if a test exists, it will be found by the algorithm. At the switch-level, transistors are viewed as bilateral switches, and a composite-valued algebra is used to represent the node values within the good and faulty circuits. The switch-level representation of CMOS circuits provides enough detail to model the faults and the bidirectional nature of CMOS circuits while retaining enough abstraction to be computationally efficient [7].

We have implemented this test generation methodology and embedded it within a test generation and test evaluation environment. Using this system, switch-level test generation and fault simulation for several small example circutis has been performed. The fault coverage results are encouraging and suggest that test generation is feasible at the switch level and is a promising area of research.

5. DESIGN ENVIRONMENT

In order to achieve an interactive design automation system that is made up of a set of disparate tools, we must develop a computing environment that has

- the ability to allow various programs that have incompatible input and output languages to communicate with each other;
- the ability to manipulate a variety of hardware description languages and to be able to resolve incompatibilities that may result;
- the ability to organize the flow of information between programs to realize a specified design task;
- the ability to implement automatically a diverse set of design tasks and methodologies;
- the ability to allow the designer to arbitrarily interrupt a design task (such as a specified program execution sequence) and restart or redirect the sequence of operations;
- the ability to explain its sequence of design activities to the designer, and to explain the reasons for particular design decisions;
- the capability of easily introducing new computer aids or tools or substituting newer tools for older ones; and
- the ability to keep track of all design attempts for a particular VLSI chip, and to make some comparison of these attempts, based on quantitative measures.

5.1. EXPERT-SYSTEM BASED ENVIRONMENT

We refer to a computing environment which has the above capabilities as a VLSI design environment. Such an environment, called ULYSSES [8], has been developed and employs AI concepts to achieve the flexibility which the above attributes require.

ULYSSES -- An Expert-System Based VLSI Design Environment

M.L. Bushnell, S.W. Director

The ULYSSES project has created a design environment appropriate for VLSI design. Conceptually, the environment may be viewed as a knowledge-based expert system for choosing which CAD program to execute next. The emphasis here is on controlling existing CAD tools, rather than on writing new ones. Furthermore, the environment is designed so that it will be very easy to substitute new, and hopefully better, CAD tools for existing tools. The environment employs a blackboard architecture originally employed in the Hearsay-II speech understanding system. This architecture is particularly suited for the VLSI problem domain, where many independent CAD tools must be controlled in order to solve various design problems. The *blackboard* is a global data base used for communicating information among various VLSI CAD tools, which are called *knowledge sources.* A language was designed to conveniently describe VLSI design activities, which are called *scripts.* An archival file backup system and a design space are part of ULYSSES. The design space supports comparisons among competing design points, and both the environment and the designer can switch design activities to a different design point when the current one no longer appears viable.

Knowledge-Based Chip Planning

A.M. Dewey, S.W. Director

The objective of this research is to develop computer-aided support for the task of chip planning by utilizing the capabilities of the ULYSSES environment. The chip planning task will assist in the incremental development of more accurate, complete, and consistent chip-level specifications. Chip planning activities can include defining initial functional specifications, studying feasibility issues, and determining realistic performance goals. The purpose of the chip planning activities is to produce a credible chip-level specification that can be used to more efficiently conduct the subsequent detailed IC design and implementation phases. Emphasizing chip planning is in contrast to current design practices where the chip-level specification either simply does not exist or is an informal description which serves only as a guideline for the implementers. Using the chip-level specification as a more formal part of the integrated circuit design process can lead to a more efficient design methodology because major design errors at the specification level are, in general, easier to detect and correct. Thus, chip planning can help to avoid costly redesigns when, halfway into an expensive design cycle, it is discovered that the initial specifications were incorrect, incomplete, or unrealistic.

5.2. USER INTERFACE

With the advent of a design environment such as ULYSSES, we may view the role of the IC designer as that of an intelligent advisor communicating with an "intelligent" design environment. In such a situation, it would become very difficult for the designer to express his intentions in a traditional manner. In order to allow the designer the greatest degree of flexibility in using the design automation system, we feel that a natural language interface is required.

The task of building a natural language interface for CAD is not, unfortunately, simply one of applying an existing AI technology to a new domain. Though the study of previous natural language interfaces is a rich source of ideas, it also underlines the imperfections of these interfaces, and the need for a fresh approach. A key component of a natural language interface is the parser, which analyses a sentence or sentence fragment and produces some kind of structured representation for it. Previous natural language interfaces, in general, have addressed this issue of parsing in rather piecemeal ways. Their coverage of constructions in English (the natural language most interfaces have dealt with) has been motivated and dictated by the domain of application of these interfaces, and not by any systematic study of the structure of English. For the goals of some of these interfaces, these limitations have not been detractions, but they have restricted the utility of the interfaces to other domains. Since most natural language interfaces have been "toy" ones, either restricted to simplified domains, or, by self-admission, never intended for "real" use, their application to substantial problems has always been problematic.

In contrast to the domains for which most natural language interfaces have been intended, the CAD domain is substantial and complex. Hence, we have developed a new approach to natural language understanding. Based on this approach, we have implemented an initial version of a natural language interface called CLEOPATRA [9] that deals with circuit simulation post-processing.

While CLEOPATRA is currently limited to the sub-domain of circuit simulation post-processing, future extensions will gradually address other design tasks. Some of these tasks will not require too much additional programming effort. These include post-processing for functional and behavioral simulators, and data-base query. Our ultimate goal, however, is a natural language interface for our integrated CAD/CAM/CAT environment. A follow-on project to CLEOPATRA is currently being pursued as described below.

An Improved Natural Language Interface for CAD

T.F. Cobourn, S.W. Director

The purpose of this project is to functionally extend CLEOPATRA. Currently, CLEOPATRA is limited to operation in the domain of circuit simulation post-processing. We intend to extend it to handle circuit simulation preprocessing as well. Also, a more user-friendly environment will be developed by integrating the existing system with interactive graphics. The new interface will be tested on designers in industry.

6. CONCLUSION

The SRC-CMU research center for CAD has been in existence for four years under the directorship of S.W. Director, and has a broad-based program aimed at developing an integrated approach to CAD, CAM and CAT of VLSI circuits and systems. Special emphasis has been placed on the investigation of appropriate AI techniques, and the development of manufacturing-based CAD and manufacturing-based CAT. A number of tools have already resulted from these activities; see Table 1. We anticipate the release of a number of additional tools in the near future as well as an environment in which these tools will communicate with each other.

Level:	Synthesis:	Simulation:		
Task	*Micon*			
Behavioral	*DAA* *FACET* *EMUCS*	*VT*	User Interface: *DELILAH* *CLEOPATRA*	Misc: *GKS* *DIF* *FRAMSMITH*
Functional	*MOBY*	*CORAL*		
Logic		*SAMSON* *PRIMO* *WASIM* *CINAMON*	Environment: *ULYSSES*	
Circuit				
Layout	*MASON* *TALIB* *WEAVER*	*FRED* *VLASIC*	CAT: *IFA*	CAM: *PROD* *STRUDEL*
Process	*PIC* *PED*	*FABRICS* *PROMETHEUS*	Optimization: *PROMISE*	

Table 1. Computer Aids Developed at CMU.

Approximately half of the $2.5 million operating budget for the center comes from the Semiconductor Research Corporation. The remaining funds, and technical guidance, come from the National Science Foundation, the Army Research Office, AT&T Bell Labs, Digital Equipment Corporation, General Electric, General Motors, Gould, GTE, Harris Semiconductor, Hewlett-Packard, Hughes Aircraft, IBM, Intel, ITT, MCC, Northern Telecom, Philips, Raytheon, and United Technologies.

ACKNOWLEDGEMENT

This paper is obviously the result of efforts by a large number of individuals. Two are especially appreciated. Gary York, assistant director of the center, compiled most of the project descriptions. Steve Director, director of the center, was responsible for the final editing.

REFERENCES

1. S.W. Director et al., "Integrated CAD, CAM, and CAT of VLSI Circuits and Systems: The CMU Perspective," *IEEE Design & Test of Computers*, June 1985.
2. S.R. Nassif and S.W. Director, "WASIM: A Waveform Based Simulator for VLSICs," *Proc. Int. Conf. on CAD*, ICCAD, November 1985.
3. A.J. Strojwas, "CMU-CAM System," *IEEE Design & Test of Computers*, February 1986.
4. P. Odryna and A.J. Strojwas, "PROD: A VLSI Fault Diagnosis System," *IEEE Design & Test of Computers*, December 1985.
5. R. Razdan and A.J. Strojwas, "A Statistical Design Rule Developer," *IEEE Trans. on CAD*, October 1986.
6. J.P. Shen et al., "Inductive Fault Analysis of MOS Integrated Circuits," *IEEE Design & Test of Computers*, December 1985.
7. S.H. Robinson and J.P. Shen, "Towards a Switch-Level Test Pattern Generation Program," *Proc. Int. Conf. on CAD*, ICCAD, November 1985.
8. M.L. Bushnell and S.W. Director, "VLSI CAD Tool Integration Using the ULYSSES Environment," *Proc. Design Automation Conf.*, DAC, July 1986.
9. T. Samad and S.W. Director, "Natural Language Interaction for Computer-Aided Design -- A First Step," *IEEE Design & Test of Computers*, August 1985.

RESEARCH IN RELIABLE VLSI ARCHITECTURES AT THE UNIVERSITY OF ILLINOIS

Jacob A. Abraham

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, IL 61801

ABSTRACT

A major research program in the design of reliable VLSI architectures is underway at the University of Illinois. The program involves twelve faculty and over thirty graduate students, and addresses issues of testing and diagnosis, concurrent error detection, and fault tolerance. This paper provides an overview of the research directions and describes selected research topics.

INTRODUCTION

Advances in Very Large Scale Integration (VLSI) technology are making possible the manufacture and use of very complex chips and systems. Unfortunately, it is becoming increasingly difficult to ensure that the products are free from manufacturing defects and that they will operate reliably in the field.

The University of Illinois has been active in the field of reliable systems for over 25 years. A major research program is now underway in the design of reliable VLSI architectures with funding from the Semiconductor Research Corporation (SRC) and its member companies. The goal is to find innovative solutions to the problems of quality in integrated circuits. The research program is very broad, addressing areas of testing and diagnosis for both manufacturing defects and field failures, concurrent (on-line) error detection to ensure reliability of computations, and fault tolerance and reconfiguration for bypassing defective or failed modules in a system.

TESTING AND DIAGNOSIS

The complexity of VLSI is making it increasingly difficult to derive high-quality test patterns which will expose defects and failures in chips and systems. Even the problem of grading the fault coverage of test patterns for today's complex chips, such as microprocessors, is beyond the capability of existing tools. The contradictory requirements of addressing realistic failures at the circuit level and, at the same time, the ability to handle chips with hundreds of thousands of transistors, make it seem that it will be almost impossible to solve the testing problem.

Our approach to this problem is two-pronged: the first step is to develop techniques and tools which can derive accurate tests or grade test patterns for realistic failures in relatively small modules and cells. The second step is the development of techniques and tools which exploit the hierarchy present in complex systems to derive tests for them, using the results of the accurate characterization of the modules in the first step. We believe that this will enable us to tackle the testing problems of the next generation of VLSI.

We have developed techniques for generating tests for realistic failures (such as shorts and opens) in MOS circuits [1]. Techniques for reducing the number of faults which need to be considered have also been derived [2]. These techniques produce a set of candidate tests for a module. These tests can be checked for their effectiveness under practical circuit operation (including effects of charge sharing, etc.) using a fault simulator.

We have developed an MOS fault simulator, FAUST, which produces output waveforms for circuits under realistic physical failures [3]. This simulator can thus be used to check whether physical failures are, indeed, detected by tests generated by any of the proposed algorithms and to obtain fault coverage for realistic failures. This tool was used to identify some problems with testing CMOS circuits.

We have determined, using our tool, that tests for shorts or opens in CMOS circuits may be invalid if they were derived using only logic-level considerations [4]. Our approach was to attempt to derive tests using various proposed techniques and to use our fault simulator to determine whether the fault under consideration was, indeed, detected. Some examples of problems with tests are given below:

(1) A logic-level test may not detect a failure because of its electrical behavior.

(2) A fault deemed undetectable because of assumptions regarding resistances may actually be detectable.

(3) A test may be invalidated under variable delays in the circuit.

(4) A test sequence may not detect a failure because of charge sharing in the circuit.

(6) Faults in latch circuits may be detectable only with a long test time.

(7) Faults in sequential circuits may be detectable only by carefully controlling the input timing.

Acknowledgment: The research program in Reliable VLSI Architectures is funded by the Semiconductor Research Corporation (SRC) under contract number 84-06-049 and by grants from SRC member companies.

It was clear from the study that tests for the submodules must be carefully designed. In particular, detailed parameters such as transistor sizes and node capacitances need to be used, since tests for some faults are critically dependent upon them. This was shown to be true even for simple failures such as shorts and opens.

Current fault simulation techniques, such as parallel fault simulation, are not powerful enough for today's very large integrated circuit designs because simulation algorithms show second-order effects. More powerful fault simulation techniques are needed to prevent a crisis in integrated circuit testing. Our research attempts to improve simulator performance through the use of new algorithms and data structures in order to overcome the current limiting factors in fault simulation. We have developed a new fault simulation technique which uses fault lists for primitives; these can be accurately produced by a transistor-level simulator such as FAUST.

Our approach to improving simulator performance has focused on reducing both the space and computation requirements of fault simulation. We substantially reduce the space required by representing the circuit as the combination of a hierarchy and tree, and simulate directly from these structures without flattening the circuit. The hierarchy is used to store invariant information such as module name, number and order of inputs and outputs, and composition in terms of other modules (macromodules). The tree structure is used to store information which is unique to each instance of a module, such as state and undetected faults. As each fault is detected, it can be removed from consideration, which reduces the total number of faults under consideration and speeds up the simulation. While improving performance, this also allows the simulator to report the current fault coverage after every stimulus, if desired. The fault coverage can either be computed for each stimulus or cumulatively for a sequence of stimuli.

These ideas resulted in the development of CHIEFS, a Concurrent, HIerarchical, and Extensible Fault Simulator [5]. A hierarchical partitioning scheme was incorporated to improve the performance; measurements show that it can dramatically speed up fault simulation (6000% on a 24 x 24-bit multiplier). An analytical model which predicts speedup due to concurrent hierarchical fault simulation has been derived; the predictions closely follow experimental results. Figure 1 shows the major components of CHIEFS. The circuit description and fault library are maintained separately. The wrapper can be used to compress a set of interconnected primitives into a single, more complex primitive. This can drastically decrease fault simulation time by reducing the number of primitives. Work is now underway on the test generator which will also exploit the hierarchy in the circuit.

We are also addressing the problem of deriving short but thorough tests for regular structures (useful for VLSI) such as iterative logic arrays and tree structures [6], as well as the design of circuits for easy testability. The problem of hardware support for fault simulation is being researched [7]. Methods for designing circuit structures for higher reliability, for example, to prevent single-event upsets [8], are also being studied.

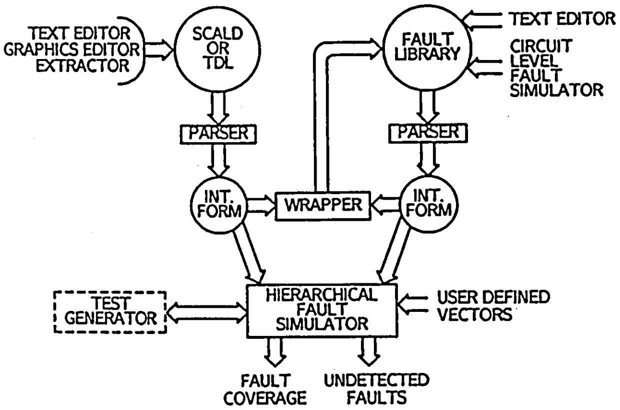


Figure 1. Major components of CHIEFS.

CONCURRENT ERROR DETECTION

Reports from the field and experimental studies indicate that the predominant failure mode in systems is due to transient rather than permanent failures. In these cases, systems should be designed with the capability for detection of errors in the computation concurrently with normal operation, since off-line testing will not necessarily detect transient faults.

A method of Concurrent Error Detection (CED) through exploitation of highly structured VLSI hardware and software module design has been developed [9]. The complexity of current hardware and software systems has encouraged the design of systems that exhibit a high degree of structural regularity. Highly structured design provides an opportunity for comprehensive CED through encoding algorithms that are designed to take into account the structural regularity of the hardware or software module. This method of encoding inputs and outputs of a module such that the module structure is included in the encoding is referred to as *structure encoding*.

Specific forms of highly structured logic arrays have been shown to be strongly fault secure and strongly code disjoint if they were programmed to implement a novel form of input/output encoding developed in this research. This encoding approach allows the use of one checker to detect errors caused by faults on inputs, outputs, or within the array itself. The approach can be modified to reduce the cost of the code-word checker by employing fault-avoidance techniques in the array design. Faults which cause unidirectional errors in the outputs rather than single-bit address or data errors can be avoided by appropriate mask-level layout rules. Such a technique has been employed for PLAs and ROMs in the design of a low-cost approach to CED in these arrays. Figure 2 shows an application of this technique.

Two classes of multistage interconnection networks have also been analyzed. Encoding schemes have been proposed for delta networks and centralized control networks that provide for comprehensive error detection. An example application of the proposed CED techniques to a mask-level design of an nMOS microprogram control unit

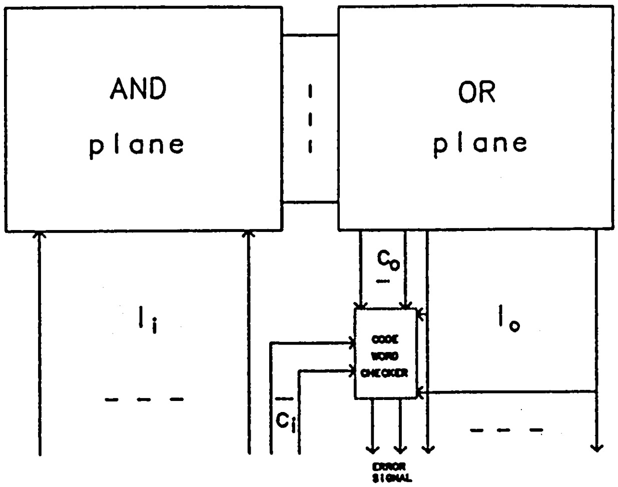


Figure 2. CED Technique for a Programmable Logic Array (PLA)

has been completed and evaluated. Implementations exhibiting a high degree of structural regularity were shown to result in cost-effective CED. These ideas have been extended to the use of structural encoding techniques to detect errors in the structural integrity of linked-data structures. Two techniques of distributed and appended checks, as well as signature access path checking, have been developed for concurrent detection of pointer errors.

We have also developed a novel technique for CED which uses time redundancy, called Recomputing with Shifted Operands (RESO) [10]. Figure 3 shows the application to a general Iterative Logic Array (ILA). The array computes the result twice, once with the normal operands and once with shifted operands. It can be shown that, with the appropriate amount of shifts, any fault in the array will affect different parts of the result in the two tries, causing the error to be detected by the checker.

Ongoing research involves the design of CED capability in complex systems using both space and time redundancy, as appropriate.

FAULT-TOLERANT SYSTEMS

Classical approaches to fault tolerance use large amounts of redundancy to correct or mask errors; an example is triplication with voting. We have developed an exciting new concept which we call *algorithm-based fault tolerance* for obtaining reliable results when computations are performed using multiple computation units (such as array processors). In this scheme, algorithms have their output encoded in a system-level error-detecting or error-correcting code. The algorithms rearrange the computations so that any failure in part of the system will only affect a portion of the result, enabling the correct result to be extracted from the encoding. This technique has been used to design low-cost fault-tolerant systems for matrix operations [11] and signal processing [12].

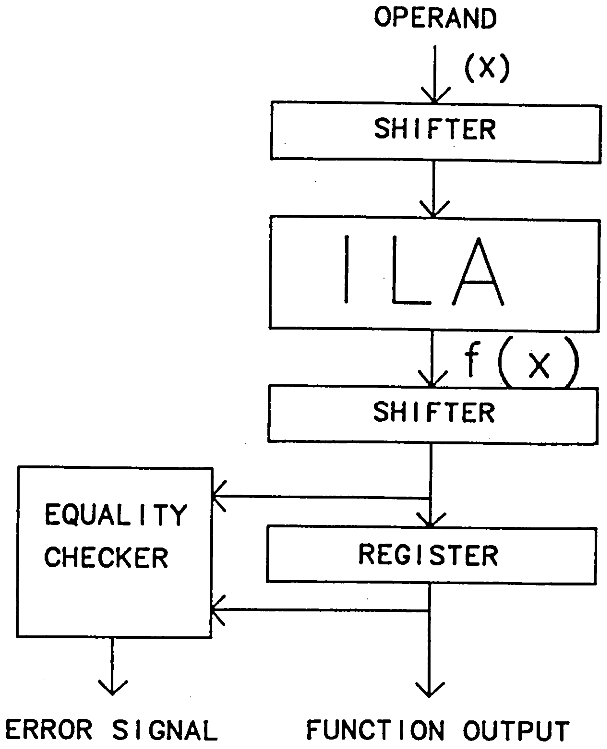


Figure 3. Recomputing with Shifted Operands (RESO).

Figure 4 shows a one-dimensional processor array for matrix-vector multiplication, $AX = B$. $A_{i,j}$ is encoded with a column checksum, where $a_{5j} = \sum_{i=1}^{4} a_{i,j}$. The matrix element $a_{i,j}$ is stored in the local memory of the ith processor at the jth time step, and x_j is broadcast to each processor at the jth time step. Each processor multiplies n pairs of $a_{i,j}$ and x_j and accumulates the products in a register. Each processor thus calculates one element of the result vector and the faulty processor affects only one element. Any error can, therefore, be detected using the fact that there is a checksum on the result. The error can be corrected and the faulty processor identified by appropriately repeating the computation on the set of processors.

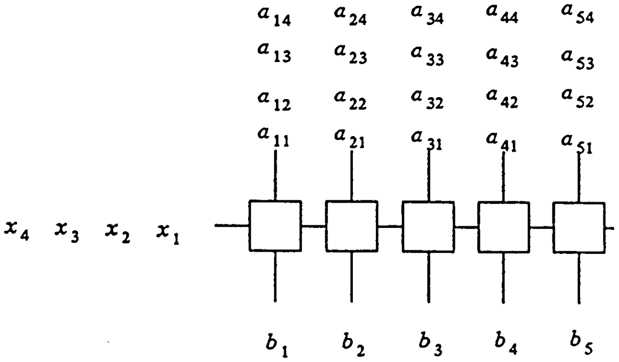

a_{14} a_{24} a_{34} a_{44} a_{54}
a_{13} a_{23} a_{33} a_{43} a_{53}
a_{12} a_{22} a_{32} a_{42} a_{52}
a_{11} a_{21} a_{31} a_{41} a_{51}
x_4 x_3 x_2 x_1
b_1 b_2 b_3 b_4 b_5

Figure 4. Checksum matrix-vector multiplication performed in a linear array.

Figure 5 shows the application of algorithm-based fault tolerance to an FFT processor [13]. The encoding uses the principles of superposition and circular shift, and it can be shown that errors due to any failure of one butterfly module will be detected by the comparator. The error can be corrected by repeating the erroneous computation with a different encoding of the data. This faulty module can also be identified at this time. If spare modules and switches are incorporated in the design, the system can be reconfigured around the faulty module.

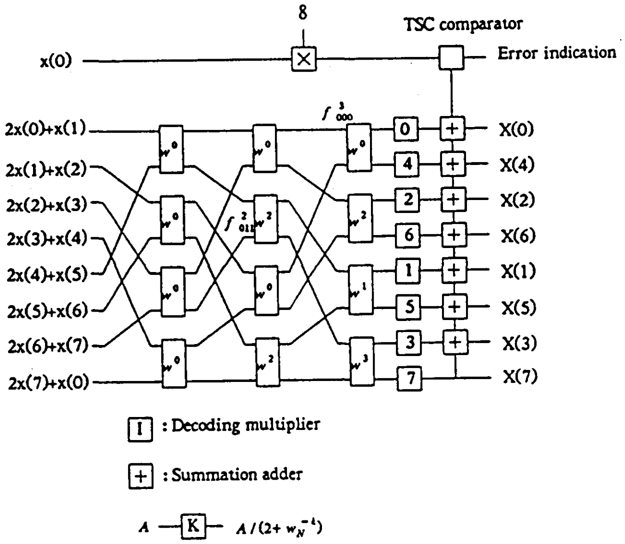


Figure 5. Algorithm-based fault tolerance in an FFT processor.

The hardware and time overheads for algorithm-based fault tolerance are very small compared with conventional techniques. This scheme is being extended to achieve fault tolerance in other signal processing systems, including those for singular value decomposition and calculation of eigenvalues. Attempts are underway to apply these ideas to more general-purpose systems.

Other work in fault tolerance involves the design of reconfigurable systems [14] and techniques for yield enhancement [15]. We feel that this research effort will result in the identification of novel, low-cost fault tolerance techniques for the next generation of VLSI.

REFERENCES

[1] H.-C. Shih and J. A. Abraham, "Transistor-Level Test Generation for Physical Failures in CMOS Circuits," *Proceedings, 23rd Design Automation Conference*, pp. 243-249, July 1986.

[2] H.-C. Shih and J. A. Abraham, "Fault Collapsing Techniques for MOS VLSI Circuits," *Proceedings, 16th International Symposium on Fault-Tolerant Computing*, pp. 370-375, July 1986.

[3] H.-C. Shih, J. T. Rahmeh, and J. A. Abraham, "FAUST: An MOS Fault Simulator with Timing Information," *IEEE Transactions on Computer-Aided Design*, 1986 (to appear).

[4] J. A. Abraham and H.-C. Shih, "Testing of MOS VLSI Circuits," *Proceedings, International Symposium on Circuits and Systems (invited paper)*, pp. 1297-1300, June 1985.

[5] W. A. Rogers and J. A. Abraham, "CHIEFS: A Concurrent, Hierarchical and Extensible Fault Simulator," *Proceedings, International Test Conference*, pp. 710-716, November 1985.

[6] W.-T. Cheng and J. H. Patel, "Testing in Two-Dimensional Iterative Logic Arrays," *Proceedings, 16th International Symposium on Fault-Tolerant Computing*, pp. 76-81, July 1986.

[7] A. J. Stein, D. G. Saab, and I. N. Hajj, "A Special-Purpose Architecture for Concurrent Fault Simulation," *Proceedings, International Conference on Computer Design*, October 1986 (to appear).

[8] S. M. Kang and D. Chu, "CMOS Circuit Design for Prevention of Single-Event Upset," *Proceedings, International Conference on Computer Design*, October 1986 (to appear).

[9] W. K. Fuchs and J. A. Abraham, "A Unified Approach to Concurrent Error Detection in Highly Structured Logic Arrays," *IEEE Journal of Solid-State Circuits*, 1986 (to appear).

[10] J. H. Patel and L.-Y. Fung, "Concurrent Error Detection in ALUs by Recomputing With Shifted Operands," *IEEE Transactions on Computers*, vol. C-31, pp. 589-595, July 1982.

[11] K. H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, pp. 518-528, June 1984.

[12] J.-Y. Jou and J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of the IEEE, Special Issue on Fault Tolerance in VLSI*, vol. 74, pp. 732-741, May 1986.

[13] J.-Y. Jou and J. A. Abraham, "Fault Tolerant FFT Networks," *Proceedings, 15th International Symposium on Fault-Tolerant Computing*, pp. 338-343, June 1985.

[14] P. Banerjee, S.-Y. Kuo, and W. K. Fuchs, "Reconfigurable Cube-Connected Cycles Architectures," *Proceedings, 16th International Symposium on Fault-Tolerant Computing*, pp. 286-291, July 1986.

[15] S.-Y. Kuo and W. K. Fuchs, "Efficient Spare Allocation in Reconfigurable Arrays," *Proceedings, 23rd Design Automation Conference*, pp. 385-390, July 1986.

HIGHLIGHTS OF VLSI RESEARCH AT BERKELEY

Carlo H. Sêquin, A. Richard Newton,
and Alberto L. Sangiovanni-Vincentelli

Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

ABSTRACT

The broad spectrum of expertise in the Department of EECS at Berkeley provides a stimulating and fertile environment for productive research in all aspects of VLSI. We report on a recent large-scale project that focussed on the development of a suite of synthesis tools in the context of the design of a chip set for a multiprocessor workstation.

INTRODUCTION

For more than two decades the EECS department at U.C. Berkeley has had strong activities in the area of integrated circuits and CAD tools for IC design and more recently also in VLSI architectures. Particularly exciting results are obtained when all of these areas start to interact. In the past, the RISC[1,2] and the SOAR[3] processors developments were strong forcing functions for our CAD environment, leading to such crucial tools as fast circuit extractors, design rule checkers, timing verifiers, PLA optimizers, and interactive layout editors.[4] Often the development of these tools occurred in response to a crucial need of a design job already in progress. This orientation towards getting real problems solved naturally shapes the nature of our tools. On the other hand, the available tools determine the scope of the designs we can hope to complete successfully.

This paper focuses on the most recent and very exciting interaction of the aforementioned VLSI research areas. In recent years the attention given to *synthesis* tools has been growing steadily. During the academic year 1985/86 this activity culminated in a coordinated effort referred to as the 'Synthesis Project'. Organized around a few graduate courses, the "official" goal of the project was to automate the design of digital integrated circuits by enhancing and integrating the various existing synthesis tools into a cohesive framework, and to create first prototypes for the missing links. The final objective is to create tools that perform as well or even better than human designers in each phase of the design process, from behavioral specification, through logic synthesis, to module generation, automated place and route, and final layout generation for a particular fabrication process.

PROJECT ORGANIZATION

In our department there is a tradition of strongly integrating research and instruction at the graduate level. Some part of a long-term research project with a well defined sub goal is made the subject of a special project-oriented graduate course. The structure of formal lecture and meeting times, combined with the hard deadlines for grades at the end of the term, often produce substantial results in a relatively short time. In the past, a lot of exploratory work has been carried out in that manner, and the best approaches were subsequently refined and implemented as MS or PhD work.

The Synthesis Project mentioned above was organized around several courses. It formally involved over forty graduate students, seven industrial visitors, and three regular faculty members. In addition many other students and faculties contributed in a substantial manner as guest lecturers or through technical discussions. The project had two distinct phases.

A preparatory phase from August through December 1985 used a regularly scheduled graduate course (EECS 244) to prepare students for the actual tool development and chip design. All aspects of IC synthesis from floorplanning, placement and routing, to automated logic design, and procedural layout were reviewed. Existing approaches to automating the design process in each area were presented, contrasted with other approaches, and classified. Data management techniques were introduced, and the students had to familiarize themselves with the *OCT* data manager.[5] All students in the course had to develope working tools and interface them to this data management system.

The second phase of the Synthesis Project took place from January through May 1986. Again, the project was

CH2345-7/86/0000/0894$01.00 © 1986 IEEE

embedded in our instructional schedule. Formally, a design-oriented graduate course (CS 292H) and a CAD tool-oriented course (EECS 290H) were used as the organizational framework, and all participants had to enroll in both courses. In the *design class*, each student was a member of one of three chip design teams. In this capacity everybody was responsible for the design of a specific module on one chip and shared responsibility for the overall function of the chip. In the other course, the *CAD class*, each student was a member of one of several CAD tool groups (e.g. logic synthesis, place and route, module generation) and was involved in developing a CAD tool suite that would be usable for all chips. This matrix organization exposed each student to the system aspects of the design process — developing a model and the detailed information flow for a given design style as well as the specific, algorithmic aspects of a particular class of tools. The students uniformly agreed that this organization was very useful in helping them gain an overview of the design process and in clarifying the function each tool must perform.

As in the past, we found it important to tie the development of new tools to a real design project; this helps to formulate realistic and practical specifications for each tool's function, interface, and performance targets. The obvious choice was to select the SPUR project[6] that had been in progress for about a year. This project concerns the development of a multiprocessor workstation for 'Symbolic Processing Using RISCs'. It presented an honest challenge in terms of complexity, as the chip set to be developed consists of three chips with over 150,000 transistors each: the central RISC processor chip (CPU), a cache controller chip (CCU), and an IEEE-standard floating-point chip (FPU). The main goal in teaming up with this project was that we had access to the architects and original implementors for consultation and that we hoped to obtain large parts of chip descriptions in a machine-readable form.

OUR SYNTHESIS SYSTEM

One of the basic requirements of our CAD environment is that it be flexible and extensible. It must be useful for the design of a wide class of circuits, from operational amplifiers to microcomputers, and the system should be able to respond do changing needs as our understanding of the VLSI design process matures. Our general approach is thus a modular set of tools, all interfaced to a common database. In this section we give a brief overview over the database and the various groups of tools addressed in the Synthesis Project.

Our automated synthesis process currently starts at the register-transfer level and converts a behavioral description into state machines, Boolean logic, and registers. The final layout is created by iteration between two groups of tools. Global placement and routing tools are used to outline the floorplan of the chip and produce desirable aspect ratios for the various modules of which the floorplan is composed. A variety of module generation tools then create the individual functional blocks, first at the symbolic level and then as properly spaced dense layouts. These finished macro modules are then reinserted into the floorplan and properly interconnected. Some tools such as a general spacing program may play a role in both phases, first compacting transistors into a dense module, then compacting symbolically routed channels to minimal physical width.

Framework and Infrastructure

To simplify the integration of our tools, we chose to use a single object-oriented data management system, *OCT*, the development of which had started some time ago.[5] OCT has as its basic unit the *cell* which can have many *views* — physical, logical, symbolic, geometrical. A cell is a portion of a chip that a designer wishes to abstract and can vary in size from a simple transistor to the entire floorplan of a CPU. The system is hierarchical, i.e., cells can contain instances of other cells. Moreover, cells can have different abstract representations depending on the intended application, and these are represented in OCT by *facets*, the accessible units that can be edited. OCT provides powerful constructs for complex data structures but manages this complexity unseen by the user.

A graphical CAD shell, *VEM*, was developed that permits the user to inspect and alter the contents of the various cells in the data base in a natural manner. OCT also provides project management support in the form of change-lists, time stamps, and search paths. All evolving synthesis tools were provided with interfaces to the OCT data manager.

Logic Synthesis

The high-level entry point to the design system is a register-transfer-level description of the chip. After evaluating a number of alternatives, we decided to use the ISP-based Behavioral Description Language (BDS) for this purpose. BDS is one of the languages used in the *DECSIM* mixed-level simulation system developed by Digital Equipment Corporation. Digital agreed to provide access to DECSIM during the Synthesis Project which gave us the possibility to use the mixed-level features of the program (behavior, register, logic gate, and switch-level simulation modes) to refine our designs. (During the course we did not get far enough to use all these options.)

In the logic synthesis tool suite,[7] the DEC BDS description is mapped by a language translator into BDSYN, a subset of BDS developed to represent logic partitioned into combinational blocks and latches. From there, another translator maps the BDSYN description into BLIF, the Berkeley Logic Intermediate Format, by expanding high-level constructs into Boolean equations. The BDS to BDSYN translation determines from a BDS description a combinational logic network equivalent to the BDS description.

MIS, a multilevel interactive logic synthesis program, then restructures the equations to minimize area and to attempt to satisfy timing constraints. MIS first implements global optimization steps that involve the factoring of Boolean equations and multiple-level minimization. Local optimization is then performed to transform locally each function into a set of implementable gates. Finally, MIS includes a timing-optimization phase that includes delay approximation based on technology data and critical-path analysis.

Module Generation, Topology Optimization

Once the logic equations have been optimized and the floorplanning tools have provided first targets for optimal aspect ratios and pin positions for the logic modules, the module generators are responsible for optimal packing of the logic into regular or irregular array-based structures.[8] These tools must also consider slack times for critical paths. If any of the objective functions (area, aspect ratio, timing) cannot be met, the floorplanning and/or logic synthesis steps can be repeated.

TOPOGEN generates a standard-cell-like layout at the symbolic level from a description of a Boolean function in the form of nested AND, OR, INVERT expressions.[9] This form is converted in a straight-forward manner into a group of series / parallel blocks of FETs in a complex static CMOS gate. The transistor placement occurs in pairs on two strips of diffusion of N- and P-type, respectively. The sequence of transistor pairs is optimized to minimize the length of the diffusion strips by maximizing the sharing of source / drain areas between adjacent transistors,[10] and to minimize the width of the wiring required between them. The output from TOPOGEN can be inspected and modified with EDISTIX, a graphic editor using a symbolic description on a virtual grid.[9] The symbolic layout can then be sent to one of the compactors mentioned below.

A more sophisticated module generator is the combination of GENIE and MKARRAY.[8] GENIE is a fairly general software package using simulated annealing to optimize the topology of a wide range of array design styles, including PLAs, SLAs, Gate Matrix, and Weinberger arrays. It handles nonuniform transistor dimensions, allows a variety of pin-position constraints, approximates desired aspect ratios by controlling the degree of column folding, and performs delay optimization. Its output is sent to the array composition tool, MKARRAY, which takes specifications of arrays of cells in a topological format. It then places the cells and aligns and interconnects all the terminals.

Layouts of existing cells, or cells where there is a special, optimized implementation that cannot be captured properly at the behavioral level (e.g. multi-ported register cells, barrel shifter), can be introduced into the system either via some adhoc *disassembly* tools, or directly through one of the editors VEM or EDISTIX.

Layout Generation

The modules at the symbolic level have to be spaced or compacted to a dense layout obeying a particular set of design rules.[11] SPARCS is a new constraint-based IC compaction tool that provides an efficient graph-based solution to the spacing problem. It can deal with upper bounds, user constraints, even symmetry requirements. It detects of over-constrained elements, and permits adjustable positioning of noncritical path elements

Another symbolic compactor under development, ZORRO, works in two dimensions and is derived from the concept of *zone refining* used in the purification of crystal ingots. The crystal is slowly fed through a zone of intense heat that locally melts the ingot. Since the generic crystal atoms recrystallize faster than other atoms, most impurities are swept out of the ingot with the molten zone. Similarly ZORRO passes an *open zone* across a precompacted layout. Circuit elements are taken from one side of this zone and are then reassembled at the other side in a denser layout. Early results obtained with ZORRO have shown reduction in layout area of up to 30% after the application of a one-dimensional compactor.[12]

Floorplanning, Place, and Route

These tools are first used to plan the chip layout and to produce optimum size and aspect ratios targets for all the modules. After the physical layouts of all the modules have been generated, the tools are used again to assemble the complete chip. Various tools have been developed to perform module placement, channel definition and ordering, global routing, and finally detailed routing.[13] These tools can handle routing on multiple layers as well as over-the-cell wiring.

TIMBERWOLF-MC [14] performs the placement function using the principle of simulated annealing. This program handles cells of arbitrary rectilinear shape; it accommodates fixed or variable shapes with optional bounds on aspect ratio, and accepts fixed, constrained, or freely variable pin locations.

CHAMELEON[15] is a new multi-level channel router that allows the specification of layer-dependent pitch and wire widths. It has as its primary objective the minimization of channel area and as its secondary objective the minimization of the number of vias and the length of each net. On two-layer problems it performs as well or better than traditional channel routers.

MIGHTY[16] is a 'rip-up and reroute' two-layer detailed switch-box router that can handle any rectagon-shaped routing region with obstructions and pins positioned on the boundary as well as inside the routing region. It outperforms all the known switch-box routers and even performs well as a channel router on problems with a simple rectangular routing region.

Analysis, Verification, and Testing

Throughout the design process, tools for analyzing and verifying performance and function are essential. The Synthesis Project did not make a major push for new tools in this area and used mostly tools that were already available.

With respect to testability we tried at the beginning of the Spring course to establish some clear policies about a design style that would guarantee testable chips.[17] The plan was to follow a scan-in / scan-out strategy.[18] However, some of the predesigned cells from the SPUR group, which we wanted to reuse because of their compactness, did not adhere to this philosophy. Other design teams did not like the extra space taken up by the more complicated scan latches and refused to use them. No consensus was reached on this point during the duration of the course. This is clearly an area that needs more attention, better tools, and much more work.

HARDWARE ENVIRONMENT

When over forty CAD developers and chip designers are working jointly on a major tool and circuit development project, a tightly coupled, highly interactive computing environment is essential. To support this course, we used an extensive collection of ethernet-based workstations and mainframe computers, running both the UNIX (Ultrix) and VMS operating systems. Our backbone machine was a VAX 8650 CPU with approximately 500 Mbytes of disc storage available to the course. This machine was our central design database server, the repository for CAD tools, both new and old, and the main electronic 'meeting place' for the participants in the Synthesis Project. In addition, seven color VAXStationII/GPX workstations and twelve monochrome VAXStationII machines were dedicated to this course. Two of these machines, used as DECSIM servers, ran the micro-VMS operating system and were tied to the other workstations via DECNET. The network capabilities allowed us to run VEM on any particular machine, while the OCT data management software could reside on any machine on the network.

RESULTS

Fifteen weeks is not enough time to build a complete synthesis system (not even at Berkeley). Thus we could not "press the button" on the last day of class and watch the layouts for the three SPUR chips pop out of the computer.

After the fifteen-week course period, all three chip designs had been converted from their original descriptions in 'N.2' or SLANG formats to BDS and inserted into our data management system. In the last weeks of the course, these descriptions were then used to exercise the pipeline of tools that had been created in parallel. Major parts of these designs have run through various tool groups and produced results of widely varying quality. Improvements were quite visible as the tools were debugged and improved.

The major benefit of this course is a very good understanding of the bottlenecks and missing links in our system and concrete plans to overcome these deficiencies. Over all, the Synthesis Project of Spring 1986 must have been a positive experience; the students polled at the end of the term voted strongly in favor of continuing the Synthesis Project in the Fall term.

CONCLUSIONS

The EECS Department in Berkeley offers a very exciting environment for research in VLSI. Circuit wizards, experts on IC processing, CAD tool builders, computer architects, and theoreticians pointing out the fundamental limits, within which the solutions must lie, are all within the same department and are strongly interacting in a cooperative manner. The quality of the chips and tools being constructed at Berkeley would not be as high as it is without this cooperative interdisciplinary interaction.

ACKNOWLEDGEMENTS

We would like to thank our colleagues who helped us, directly or indirectly, to create a powerful CAD environment and thank all the users for their patience with tools that are still far from perfect.

The development of our CAD tools is strongly supported by the Semiconductor Research Corporation, by Digital Equipment Corporation, and by Tektronix.

References

1. D.A. Patterson and C.H. Séquin, "A VLSI RISC," *Computer*, vol. 15, no. 9, pp. 8-21, Sept. 1982.
2. M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI,* MIT Press, 1984.

3. D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Symp. on Computer Architecture*, Ann Arbor, MI, June 1984.

4. W.S. Scott , R.N. Mayo, G. Hamachi, and J.K. Ousterhout, "1986 VLSI Tools," CS Division Report No. UCB/CSD 86/272, University of California, Berkeley, CA, 1985.

5. D. Harrison, P. Moore, A.R. Newton, A.L. Sangiovanni-Vincentelli, and C.H. Séquin, "Data Management in the Berkeley Design Environment," *submitted to ICCAD-86*, Santa Clara, CA, Nov. 1986.

6. M.D. Hill, S.J. Eggers, J.R. Larus, G.S. Taylor, G. Adams, B.K. Bose, G.A. Gibson, P.M. Hansen, J. Keller, S.I. Kong, C.G. Lee, D. Lee, J.M. Pendleton, S.A. Ritchie, D.A. Wood, B.G. Zorn, P.N. Hilfinger, D.A. Hodges, R.H. Katz, J.K. Ousterhout, and D.A. Patterson, "SPUR: A VLSI Multiprocessor Workstation," CS Division Report No. UCB/CSD 86/273, University of California, Berkeley, CA, 1986.

7. R. Brayton, A. Cagnola, E. Detjens, K. Eberhard, S. Krishna , P. McGeer, L.F. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, R. Yung, T. Villa, A.R. Newton, A.L. Sangiovanni-Vincentelli, and C.H. Séquin, "Multiple-Level Logic Optimization System," *submitted to ICCAD-86*, Santa Clara, CA, Nov.1986.

8. G. Adams, S. Devadas, K. Eberhard, C. Kring, F. Obermeier, P.S. Tzeng, A.R. Newton, A.L. Sangiovanni-Vincentelli, and C.H. Séquin, "Module Generation Systems," *submitted to ICCAD-86*, Santa Clara, CA, Nov. 1986.

9. C.H. Séquin, "Design and Layout Generation at the Symbolic Level," in *Proceedings of the Summer School on VLSI Tools and Applications*, ed. W. Fichtner and M. Morf, Kluwer Acadmic Publishers, 1986.

10. T. Uehara and W.M. VanCleemput, "Optimal Layout of CMOS Functional Arrays," *Trans. Comp.*, vol. C-30, no. 5, 1981.

11. J.L. Burns, T. Laidig, B. Lin, H. Shin, P.S. Tzeng, A.R. Newton, A.L. Sangiovanni-Vincentelli, and C.H. Séquin, "Symbolic Design Using the Berkeley Design Environment," *submitted to ICCAD-86*, Santa Clara, CA, Nov. 1986.

12. H. Shin and C.H. Séquin, "Two-Dimensional Compaction by Zone Refining," *Proc. Design Autom. Conf., Paper 7.3*, Las Vegas, July 1986.

13. J. Burns, A. Casotto, G. Cheng, W. Dai, M. Igusa, M. Kubota, U. Lauther, F. Marron, F. Romeo, C. Sechen, H. Shin, G. Srinath, H. Yaghutiel, A.R. Newton, A.L. Sangiovanni-Vincentelli, and C.H. Séquin, "MOSAICO: An Integrated Macrocell Layout System," *submitted to ICCAD-86*, Santa Clara, CA, Nov. 1986.

14. C. Sechen and A. Sangiovanni-Vincentelli, "TIMBERWOLF 3.2: A New Standard Cell Placement and Global Routing Package," *Proc. Design Autom. Conf., Paper 26.1*, Las Vegas, July 1986.

15. A. Sangiovanni-Vincentelli, D. Braun, J. Burns, S. Devadas, H.K. Ma, K. Mayaram, and F. Romeo, "CHAMELEON: A New Multi-Layer Channel Router," *Proc. Design Autom. Conf., Paper 28.4*, Las Vegas, July 1986.

16. H. Shin and A. Sangiovanni-Vincentelli, "MIGHTY: A 'Rip-up and Reroute' Detailed Router," *submitted to ICCAD-86*, Santa Clara, CA, Nov. 1986.

17. R. Brayton, E. Detjens, S. Krishna , B. Lin, H.K. Ma, F. Obermeier, T. Quarles, L.F. Pei, R. Spickelmier, J. Tam, A. Wang, D. Webber, R.S. Wei, N. Weiner, A.R. Newton, A.L. Sangiovanni-Vincentelli, and C.H. Séquin, "Simulation, Timing Analysis, Verification and Testing in the Berkeley Synthesis Project," *submitted to ICCAD-86*, Santa Clara, CA, Nov. 1986.

18. E.B. Eichelberger and T.W. Williams, "A Logic Design Structure for LSI Testability," *Jour. Design Automation and Fault-Tolerant Computing*, vol. 2, pp. 165-178, May 1978.

DEFT - A DESIGN-FOR-TESTABILITY EXPERT SYSTEM

M. Arif Samad J. A. B. Fortes *
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

ABSTRACT

DEFT is a knowledge-based program which uses Design for Testability (DFT) knowledge to modify circuits into more easily testable circuits. This paper describes the DEFT system and gives an example of its operation. The paper begins with a review of previous research in automated design for testability and then presents an overview of the operation and organization of the DEFT system. This includes a discussion of the Prolog-based knowledge representation scheme used to describe DFT knowledge. The demon mechanism is described which allows DEFT to make use of Lisp functions to handle low-level implementation details. This mechanism allows DEFT to benefit from a high-level representation without sacrificing either efficiency or ease of implementation. An example is presented of an 8 X 8 multiplier to which DEFT adds a scan path. The implementation of the DEFT system is currently in progress. The last section of the paper reviews the status of the implementation effort.

1. INTRODUCTION

DEFT is a knowledge-based design for testability system which accepts as input the description of a circuit that has been designed without any testability features and modifies it into an easily testable circuit. DEFT is designed to work in conjunction with the IBM MVISA Design Automation System recently installed at Purdue University [25]. The MVISA system allows for the design of integrated circuits using standard cells ranging from and/or gates to arithmetic logic units. The transformation of a logical design using these standard cells into a physical design, i.e. a chip layout, is done largely automatically. DEFT takes as input a circuit description created using the MVISA system, adds hardware and makes other modifications to the circuit and then returns a description of the modified circuit to MVISA. In this way, design for testability is incorporated into the design cycle with little effort on the part of the circuit designer. DEFT is able to provide the user with an explanation of its behavior and of related design for testability concepts. The knowledge that enables a program to exhibit expert problem-solving behavior is often not sufficient for the purpose of explanation. Generally, it is necessary to include extra "support" knowledge in the knowledge-base expressly for the purpose of explanation. The DEFT knowledge-base contains a representation of general testability concepts and principles, in addition to specific design for testability schemes. This enables DEFT to provide the user with meaningful explanations of its actions during the course of problem-solving. The DEFT explanation facility is described in greater detail in [27].

This paper begins with a discussion of the motivation for using the knowledge-based systems approach for DEFT. This is followed by a review of previous research in automated design for testability. The architecture of the DEFT system and the knowledge representation scheme used are then examined. An example is then presented of a circuit to which DEFT adds a scan path. The paper concludes with a review of the implementation status of the system.

2. KNOWLEDGE-BASED SYSTEMS IN DESIGN FOR TESTABILITY

2.1. DESIGN FOR TESTABILITY

Design for Testability refers to a collection of techniques that can be used to make integrated circuits easier to test [29,9,4,28]. There are a number of different classes of DFT techniques which focus on different kinds of testability problems, e.g., memory test, PLA test etc. Many DFT techniques are concerned with controlling the memory elements of sequential circuits so that test pattern generation programs for combinational circuits such as those based on the D-Algorithm [26] can be used. Examples of such schemes include LSSD [8], Scan Path [10] and Random Access Scan [2].

There is increasing interest in designing circuits with additional test hardware so that they are able to test themselves. This capability is called Built-In Self-Test (BIST). Many of the ideas developed in connection with the DFT schemes mentioned earlier are relevant to BIST. For example, many BIST schemes make use of the scan path idea [20]. In general, BIST circuitry consists of hardware for test pattern generation and response analysis [23,24]. The most common configurations use linear feedback shift registers to generate test patterns and capture the circuit response, e.g., BILBO [21,22].

2.2. MOTIVATION FOR USING A KNOWLEDGE-BASED SYSTEM

Part of the motivation for designing DEFT as a knowledge-based system, i.e. a system with a distinct knowledge-base constructed using a formal knowledge representation language, came from the nature of design for testability knowledge. The DFT domain is characterized by a large variety of problems, with several possible solutions to each problem. Moreover as technology develops, new testability problems arise. Since testability knowledge is dynamic, it is important that the knowledge

* This work was supported in part by a grant from the AT&T Foundation

CH2345-7/86/0000/0899$01.00 © 1986 IEEE

in an automated DFT system be observable so that the system can be modified easily.

A second motivating factor in using the knowledge-based approach was a desire to incorporate an explanation capability in DEFT. In order for a system to produce meaningful explanations, it must include not just the knowledge used to implement a DFT scheme but also some of the principles and motivations underlying these schemes. Since these motivations are often abstract and complex in nature, it was felt that the flexibility and power of AI-based knowledge representation schemes could be put to good use.

A somewhat philosophical reason for using formal knowledge representation was the desire to codify DFT knowledge in a form that was meaningful to humans but was still rigorous enough for use by a computer system. The advantage of using a relatively high-level language to express knowledge is that the knowledge-base developed can conceivably be used by different programs. For example, the rules used by DEFT describing different DFT schemes could be used by a different program, say a silicon compiler, that needed DFT knowledge.

Recently a number of knowledge-based automated design for testability systems have been presented in the literature. The next section briefly reviews some of this work.

2.3. PREVIOUS RESEARCH IN KNOWLEDGE-BASED DFT

The pioneering effort in the field of knowledge-based DFT systems was the work done by Paul Horstmann as part of his doctoral research at Syracuse University [14,15,16,17,18]. Horstmann used Prolog to implement a system with testability rules that were able to detect and remove violations of the LSSD DFT methodology.

Breuer and Zhu describe the PLA-ESS system that helps choose a DFT methodology for a PLA [5]. The PLA-ESS system uses an evaluation matrix containing the tradeoffs for various DFT schemes for PLA's to choose an appropriate scheme. In case no scheme can be found that satisfies all the criteria specified by the user, the system uses a strategy called Reason Analysis Directed Backtracking to see if some of the criteria specified by the user can be relaxed in order to fit the capabilities of the existing DFT schemes known to the system.

The Testable Design Expert System (TDES) was developed by Abadir and Breuer to work with the Advanced Design Automation System (ADAM) under development at the University of Southern California [1]. The input to TDES consists of a register-transfer-level description of a circuit as well as design goals and constraints. The TDES knowledge-base is organized into Testable Design Methodologies (TDM's) that are implemented as frames.

Fung, Hirschhorn and Kulkarni describe an automatic design for testability (ADFT) system that works in conjunction with the Silc silicon compiler being developed at GTE Laboratories [11,12]. The Silc ADFT system consists of testability rules which are built into Silc's parsing and logic synthesis software, the Testability Evaluator which identifies hard to test parts of the design being synthesized and the Testpert which proposes changes to the design.

As the preceding description shows, significant progress has been made in the area of knowledge-based design for testability. The following section presents the operation and organization of the DEFT system.

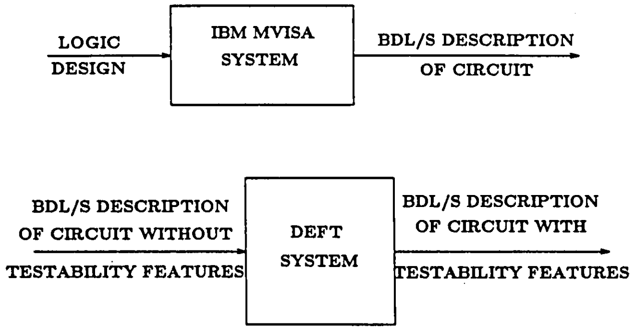


Fig. 1. The IBM MVISA System and DEFT

3. THE DEFT SYSTEM - OPERATION AND ORGANIZATION

3.1. SYSTEM OPERATION

DEFT attempts to incorporate testability considerations into the integrated circuit design cycle without excessively burdening the chip designer. The prototype works in conjunction with the IBM MVISA design automation system recently installed at Purdue (Fig.1). The MVISA system allows the integrated circuit designer to implement his design using standard logic elements such as arithmetic logic units, registers, etc. The designer is responsible mainly for the logical design of the circuit: the mapping of the logical design to a physical design (mask image) is performed with minimal designer intervention. The MVISA system uses IBM's Basic Design Language for Structure (BDL/S) to describe the circuit being designed. The inputs to DEFT consists of a BDL/S description of the circuit which is to be modified to improve testability and a statement of the testability objectives such as fault coverage, hardware overhead, etc. DEFT uses these objectives to choose a DFT scheme and then modifies the circuit to improve testability. These modifications generally consist of the addition of hardware to facilitate test. The added hardware is purely for testability purposes and the circuit function remains unchanged. After DEFT has decided what modifications to the circuit are appropriate, it uses its knowledge of how to implement the chosen DFT scheme to modify the circuit. A new BDL/S description of the circuit is then produced which can be fed back to MVISA. In this way, design for testability is incorporated into the design cycle with minimal intervention from the designer.

After DEFT has finished modifying the circuit, the user can request the system for an explanation of the reasoning steps underlying the suggested changes. DEFT uses a mixture of text and graphics to explain its behavior. The system user can not only query DEFT about the specific case that was being worked upon but also examine general testability information related to the case. The DEFT explanation facility makes use of dialogue and computer graphics to explain various testability concepts. The explanation for a concept generally consists of sequences of images, some animated, that attempt to illustrate the concept. The graphics are presented on an AT&T 5620 bit-mapped display. The DEFT explanation facility is discussed at greater length in [27].

The next section analyzes the characteristics of knowledge in design for testability and other engineering domains. This analysis is used as a foundation for the development of an abstract problem-solving architecture which attempts to provide a framework for the expression of knowledge in DFT.

3.2. THE NATURE OF KNOWLEDGE IN DFT

Most problem-solving systems for DFT (and other engineering domains) have the following kinds of knowledge associated with them:

1. **Heuristic knowledge**
2. **Procedural knowledge (non-heuristic)**
3. **Structural knowledge**

Heuristic knowledge often takes the form of advice on what to do given a particular set of circumstances. Heuristic knowledge can be further categorized into **strategic, meta-level knowledge** (which guides the overall problem-solving process) and **microlevel heuristics** (which are useful in dealing with particular steps during the course of solving a problem). In the DFT domain, the knowledge that is concerned with choosing a testability scheme falls under the category of strategic knowledge. An example of microlevel heuristics would be advice about the order in which hardware modifications should be made to a circuit during the course of implementing a DFT scheme.

The non-heuristic, **procedural knowledge** consists of plans or sequences of actions that must be taken to achieve problem-solving goals. Like the heuristic knowledge described above, the procedural knowledge also appears at different levels of abstraction. A plan can be described at a high-level as a conjunction of abstract goals; each of these goals can then be refined into subgoals at lower levels of abstraction, e.g., a scheme for adding a scan path to a circuit (the goal: make complete scan path) can be described at an abstract level as a conjunction of three goals.

```
make complete scan path :-
   replace all non-scannable memory elements
      by scannable memory elements,
   add hardware for scan path and
      scan clocks i/o,
   connect scannable memory elements to
      form scan path.
```

The third category of knowledge is **structural knowledge** about the domain which could include a **taxonomy** and **causal models** of behavior of objects in the domain. In the design for testability domain, the most important objects are circuit elements. Knowledge about how these circuit elements can be classified and how these classes relate to each other would fall in this category. For example, various latches and flip-flops might be classified under the generic term memory elements. Other structural knowledge might include fault-models and how they relate to each other, and how different DFT objectives are related. Circuit simulators and mathematical models describing the impact of increased hardware overhead on yield would fall under the category of causal models.

While it is possible to categorize domain knowledge in DFT in this way, it is not as simple to describe the interaction of the various kinds of knowledge during problem-solving. An ideal problem-solving architecture must allow for completely flexible invocation of the various categories of knowledge as they are needed. In the next section, an abstract problem-solving architecture is presented which allows the different kinds of knowledge discussed above to be represented and used in problem-solving. The organization of DEFT is patterned after this architecture.

3.3. AN ABSTRACT PROBLEM-SOLVING ARCHITECTURE (Fig.2)

The heuristic knowledge in a knowledge-based system is generally best represented using a rule-based scheme with a **forward-chaining** control scheme. Such an architecture is a good model for the data-directed way in which heuristic knowledge is used during the problem-solving process. For each cycle of the production system's operation, the antecedent of each rule in the system is matched against the global-database to see if the rule is applicable. In this way all the rules that are relevant to the situation at any particular time are applied[13].

In contrast, the non-heuristic, procedural knowledge in the system is most appropriately represented using a rule-based scheme with a **backward-chaining** control scheme. In a backward chaining system, a plan or procedure to solve a problem can be represented using an **AND/OR goal tree** [3] which is traversed by the system from the root, which represents the goal to be solved, downwards towards the leaves, which represent various low-level steps needed to achieve the goal.

The plan to solve the goal at the root of the AND/OR tree is represented in abstract terms on the levels of the tree close to the root and becomes more detailed towards the leaves of the tree. This ability to represent procedural knowledge at different levels of abstraction is one of the advantages of the goal tree representation and has important implications in terms of the efficiency of problem-solving, the quality of explanations produced by a system and the ease with which new knowledge can be added to the system.

The structural knowledge in the problem-solving system can be represented using a **frame-based representation** supporting **inheritance of properties[6]**. A frame network can be used to describe both the relationships between objects in the domain and a causal model for behavior in the domain. Other specialized modeling mechanisms such as circuit simulators, mathematical equations, etc, may also be used where appropriate.

Many existing AI tools support one or the other of the knowledge-representation and inference schemes discussed above. An ideal architecture would consist of an integrated package that allowed for the flexible interaction of the various types of knowledge in the system. This system would have the capability to use specialized representations for domain objects where appropriate. An example of such an architecture is shown in Fig.2.

The system has the following components:

1. A **global data-base** - this data-base contains a description of the current problem.
2. The **domain knowledge-base** - this consists of the heuristic, procedural, and structural knowledge described earlier.
3. The **inference mechanisms** - these use the global data-base and the domain knowledge-base to do the actual problem-solving. The architecture shown here

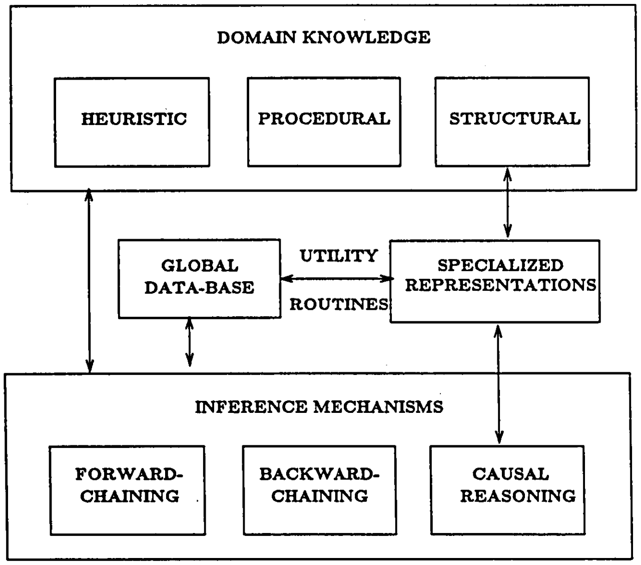


Fig. 2. An Architecture for a Knowledge-Based Problem-Solver

incorporates both forward and backward chaining inference.

4. **Specialized representations** - these consist of efficient representations for domain objects. The term "specialized" is used to distinguish these representations from the knowledge representation scheme being used to represent information in the global data-base. The specialized representations cannot be directly used by the inference mechanisms. For example, DEFT uses a graph to represent the circuit. This graph is represented using a Lisp data-structure rather than Prolog clauses in the global database, allowing for far more efficient operations on the circuit graph.
5. **Utility routines** - these consist of interface routines between the global data-base and the specialized representations. These routines are implemented as demons that watch over the global data-base. When the problem-solving process requires a piece of information that is not present in the global data-base, the appropriate utility routine is triggered and extracts the required information from the specialized representation and then adds it to the knowledge-base. Similarly, the specialized representations are updated by the utility routines to reflect changes in the global data-base. In DEFT, there are a number of demons that use the circuit graph described above to extract any information about the circuit that may be needed during the problem-solving process.

The DEFT system attempts to incorporate many of the ideas presented in the discussion above. The next section presents the DEFT knowledge representation and the demon mechanism.

3.4. DEFT KNOWLEDGE REPRESENTATION AND THE DEMON MECHANISM

DEFT uses a Prolog-based knowledge representation scheme[7]. The DEFT Prolog interpreter is written in Franz Lisp and incorporates a special **demon mechanism** which allows it to interface with Lisp. This makes it possible to use the flexibility of Lisp to enhance program efficiency where necessary. A set of Prolog rules implicitly define an AND/OR graph which is searched in a depth-first manner. DEFT uses Prolog to define AND/OR trees corresponding to different DFT schemes. For example, Fig. 3 shows a set of rules for the DFT modification, **make_complete_scan_path.** These are the rules that are used by DEFT to add a scan path to the 8 X 8 multiplier in the example in section 4. A fragment of the AND/OR tree implicit in these rules is shown in Fig.4.

The rules shown in Fig.3 represent a high-level description of how to make a complete scan path. However, so far we have said nothing about how DEFT actually uses these rules to modify the data-structure representing the circuit under consideration. In the problem-solving model discussed earlier, it was proposed that specialized representations of domain objects be used whenever necessary in order to make an efficient system. In keeping with this philosophy, DEFT represents the circuit under consideration as a graph implemented using a Lisp data-structure. Since DEFT works with actual circuit descriptions from the IBM MVISA system, there is a large amount of information that must be stored and manipulated during the course of implementing a DFT scheme. It is easier and generally more efficient to use specialized data-structures and access routines for this purpose than to represent the entire circuit as clauses in the Prolog database.

Each demon is a Lisp routine that is used by the interpreter as described below. When attempting to solve a goal, the DEFT Prolog interpreter first attempts to use the rules contained in the Prolog database; this is how conventional Prolog operates. However, if the DEFT Prolog interpreter is unable to satisfy a goal using the rules that are contained in the database, it looks instead for a demon corresponding to the goal under consideration. These demons work by adding new clauses to the Prolog database which are then used by the interpreter to satisfy the goal. The demon mechanism can be thought of as a virtual extension to the Prolog database, i.e., demons can be used to access information that is not stored explicitly in rule form in the database whenever it is needed.

The demon mechanism used in DEFT is very similar in intent and implementation to the language FProlog which attempts to combine functional and logic programming [19]. FProlog allows arbitrary Lisp functions to be accessed from Prolog. These functions can return information to the FProlog world by instantiating variables in the FProlog rule for which the function is called. In contrast to this, the DEFT demon mechanism works by adding clauses to the Prolog database as discussed above.

In the rules shown in Fig.3, many of the predicates used in the formulas have demons associated with them, e.g., the predicate "latch" used in the second rule in the set has a demon associated with it (Fig.5). The first time that the predicate "latch" is referenced in a rule the associated demon is triggered. This demon accesses the Lisp data-structure representing the circuit to get a list of latches and then adds a set of facts of the form shown below to the Prolog data-base, one for each latch in the circuit (the names AA100AB and AA100DE are arbitrary).

```
latch(AA100AB).
latch(AA100DE).
      •
      •
```

```
make_complete_scan_path :-
   replace_non_scannable_latches_by_scannable_latches,
   connect_latches_to_scan_clocks,
   connect_scannable_latches_to_form_scan_paths.

replace_non_scannable_latches_by_scannable_latches :-
   latch(Latch),
   not(scannable_latch(Latch)),
   replace_by_scannable_latch(Latch),
   fail.

replace_non_scannable_latches_by_scannable_latches.

replace_by_scannable_latch(Latch) :-
   !,
   replace(Latch,scannable_latch).

connect_latches_to_scan_clocks :-
   connect_latches_to_A_clock.

connect_latches_to_scan_clocks.

connect_latches_to_A_clock :-
   add_receivers_for_scan_clocks,!,
   scannable_latch(Latch),
   connect_latch_to_A_clock(Latch).

connect_latches_to_A_clock.

connect_latch_to_A_clock(Latch) :-
   a_clock_input(Latch,A_clock_input),
   a_clock(A_clock),
   connect(A_clock,A_clock_input),
   !,
   fail.

add_receivers_for_scan_clocks :-
   choose_io_pin_for_A_clock(A_clock_pin),
   add_receiver_for_pin(A_clock_pin,A_clock_rcvr),
   output_of_receiver(A_clock_rcvr,A_clock),
   asserta(a_clock(A_clock)).

choose_io_pin_for_A_clock(First) :-
   free_io_pins([[First|_]|Rest]),
   delete_pin_from_free_io_list(First).


connect_scannable_latches_to_form_scan_paths :-
   choose_chip_scan_in_pin(ChipScanInPin),
   choose_chip_scan_out_pin(ChipScanOutPin),
   form_scan_path.

choose_chip_scan_in_pin(ScanInPin) :-
   free_io_pins(FreeIoPins),
   latch_closest_to_any_corner_of_chip(Latch),!,
   location_of_block(Latch,Location),!,
   free_io_pin_closest_to_location(FreeIoPins,Location,ScanInPin),!,
   delete_pin_from_free_io_list(ScanInPin),
   asserta(first_element_of_scan_path(Latch)),
   asserta(chip_scan_in_pin(ScanInPin)).

delete_pin_from_free_io_list(Pin) :-
   free_io_pins(FreeIoPins),
   delete_using_key(Pin,FreeIoPins,NewFreeIoPins),
   retract(free_io_pins(FreeIoPins)),
   asserta(free_io_pins(NewFreeIoPins)).

choose_chip_scan_out_pin(ScanOutPin) :-
   free_io_pins(FreeIoPins),
   chip_scan_in_pin(ScanInPin),
   location_of_io_pin(ScanInPin,ScanInLocation),!,
   corner_closest_to_location(ScanInLocation,Corner),!,
   corner_opposite_to_corner(Corner,OppositeCorner),!,
   free_io_pin_closest_to_location(FreeIoPins,OppositeCorner,ScanOutPin),!,
   delete_pin_from_free_io_list(ScanOutPin),
   asserta(chip_scan_out_pin(ScanOutPin)).

form_scan_path :-
   chip_scan_in_pin(ChipScanIn),
   add_receiver_for_pin(ChipScanIn,ReceiverId),!,
   output_of_receiver(ReceiverId,ReceiverOutput),!,
   first_element_of_scan_path(FirstLatch),!,
   scan_in_pin_of_latch(FirstLatch,FirstLatchScanIn),!,
   connect(ReceiverOutput,FirstLatchScanIn),!,
   delete_from_unconnected_latch_list(FirstLatch,Rest),
   connect_latches_to_form_scan_path(FirstLatch,Rest).

connect_latches_to_form_scan_path(FirstLatch,[]) :-
   scan_out_pin_of_latch(FirstLatch,LatchScanOut),!,
   chip_scan_out_pin(ChipScanOut),!,
   add_off_chip_driver_for_pin(ChipScanOut,OCDId),!,
   input_of_off_chip_driver(OCDId,OCDInput),!,
   connect(LatchScanOut,OCDInput).

connect_latches_to_form_scan_path(FirstLatch,RemainingLatches) :-
   location_of_block(FirstLatch,FirstLoc),!,
   unconnected_latch_closest_to_location(RemainingLatches,
                                      FirstLoc,NextLatch),!,
   scan_out_pin_of_latch(FirstLatch,FirstScanOut),!,
   scan_in_pin_of_latch(NextLatch,NextScanIn),!,
   connect(FirstScanOut,NextScanIn),!,
   delete_from_unconnected_latch_list(NextLatch,Rest),
   connect_latches_to_form_scan_path(NextLatch,Rest).


delete_from_unconnected_latch_list(Latch,RemainingLatches) :-
   unconnected_latches(UnconnectedLatches),
   delete(Latch,UnconnectedLatches,RemainingLatches),
   retract(unconnected_latches(UnconnectedLatches)),
   asserta(unconnected_latches(RemainingLatches)).
```

Fig. 3. DEFT Rules to Add a Scan-Path to a Circuit

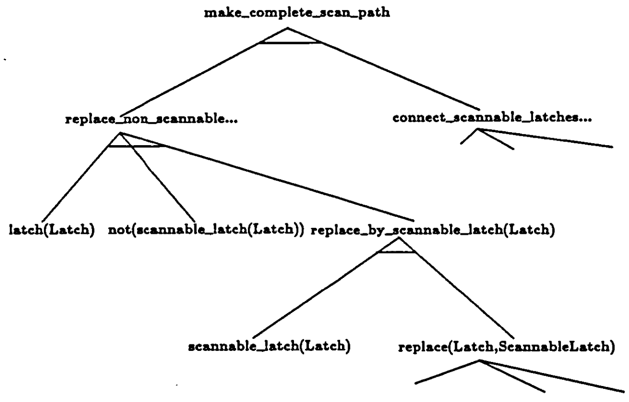

make_complete_scan_path
replace_non_scannable...
connect_scannable_latches...
latch(Latch)
not(scannable_latch(Latch))
replace_by_scannable_latch(Latch)
scannable_latch(Latch)
replace(Latch,ScannableLatch)

Fig. 4. A Fragment of the AND/OR tree for make_complete_scan_path

```
(defun latch(sent itree)
 (prog(latches)
     (putprop 'latch t 'inactive)              ; deactivate demon
     (setq latches (GetLatches))
     (mapc
       (function
        (lambda(latch)
              (AddRule '((latch ,latch) nil))  ))
       latches)
     (return t)))
```

Fig. 5. A demon for the Prolog clause latch

An interesting and important effect of the demon mechanism is that the it allows the rules used by DEFT to describe DFT schemes to be written in an abstract, technology independent way. For example, the second rule in Fig.3 uses the abstract term "latch". The demon mechanism knows which circuit elements actually used by the IBM MVISA system correspond to the abstract latch. Since the rules themselves make no reference to MVISA circuit elements, they could presumably be used with a different design automation system with the demons appropriately modified.

In addition to endowing the rules with a degree of technology independence, the demon mechanism allows for a description of DFT schemes that is not cluttered by details of how the circuit data-structure is actually manipulated. All that detail is hidden behind the demons. One way of thinking about the demons is that they are equivalent to the routines that would probably make up DEFT had it been developed as a conventional program rather than as a knowledge-based system.

4. AN EXAMPLE - ADDING A SCAN PATH TO AN 8 BY 8 MULTIPLIER

Fig. 6 shows a plot of the chip layout of an 8 X 8 multiplier after it has been modified by DEFT. The modifications to the multiplier include the addition of a receiver for the scan in, a receiver for the A scan clock, a tri-state off chip driver for the scan out, the connection of the A clock to the A clock inputs of the SRL's and the interconnection of the registers to form a scan path. The added hardware is encircled in Fig. 6; the heavy black line represents the scan path wiring. Fig. 7 and 8 show close-ups of the added receivers and off chip driver. Fig. 9 shows a sheet of logic from the MVISA circuit description of the 8 X 8 multiplier before the modification of the circuit by DEFT. Note that the scan in pin and the A clock

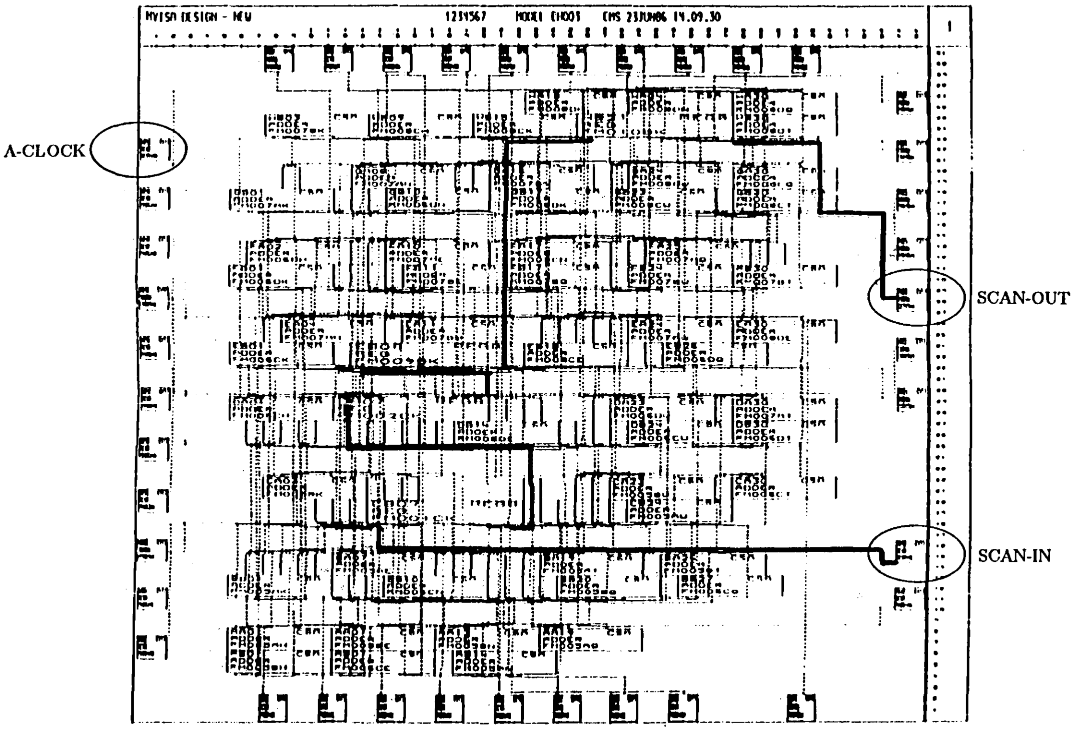


Fig. 6. 8×8 Multiplier with Added Hardware and Wiring

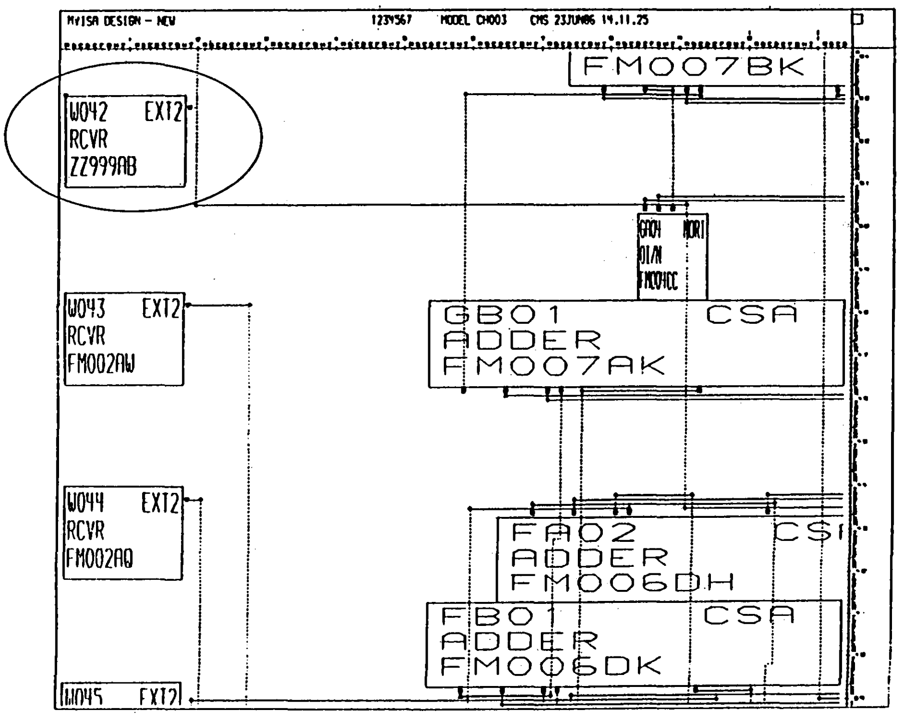


Fig. 7. Receiver Added for A-CLOCK

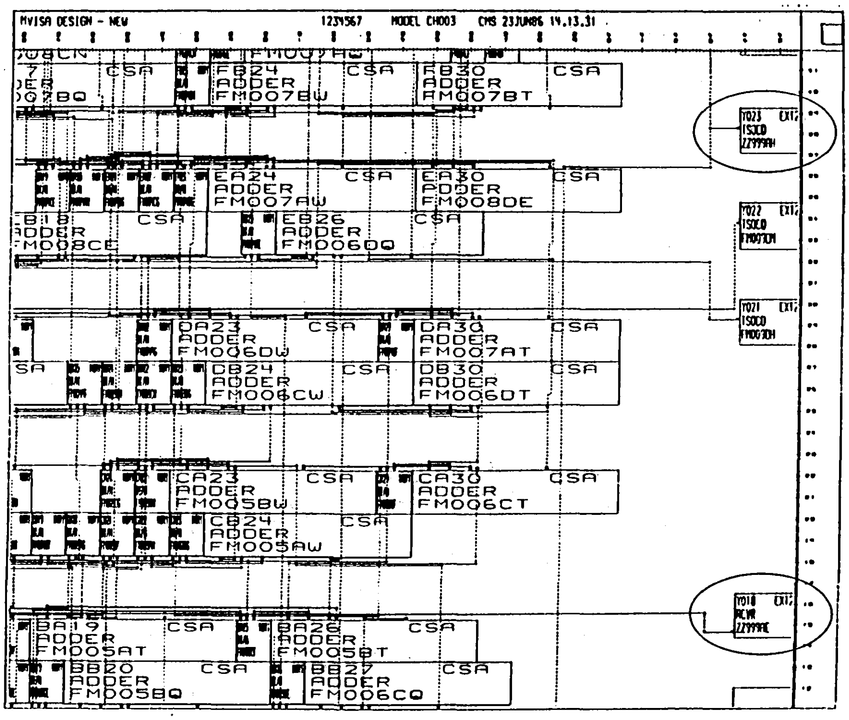


Fig. 8. Receiver Added for Chip SCAN-IN and Off Chip Driver Added for Chip SCAN-OUT

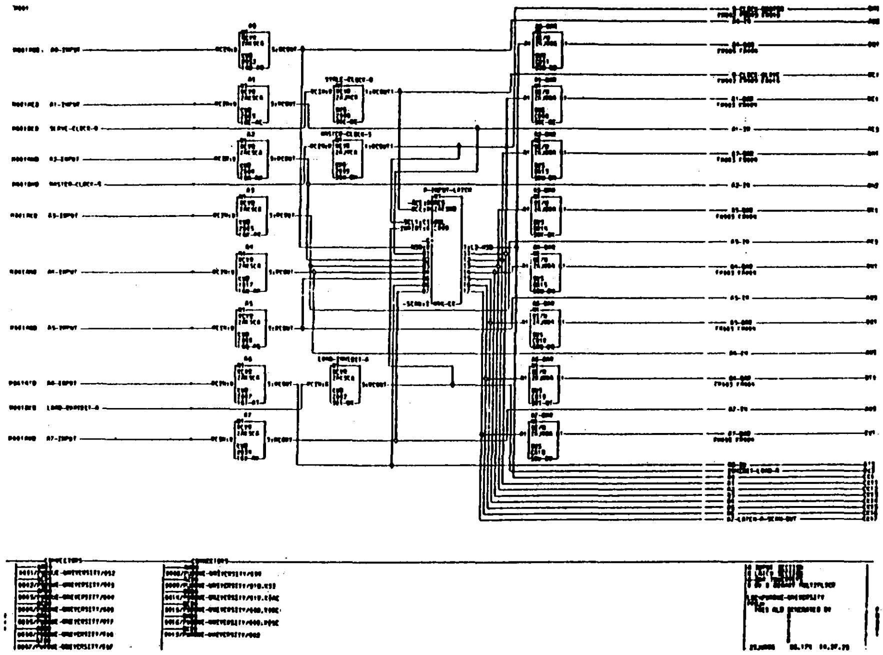


Fig. 9. Sheet of Logic with A-CLOCK and SCAN-IN Not Connected

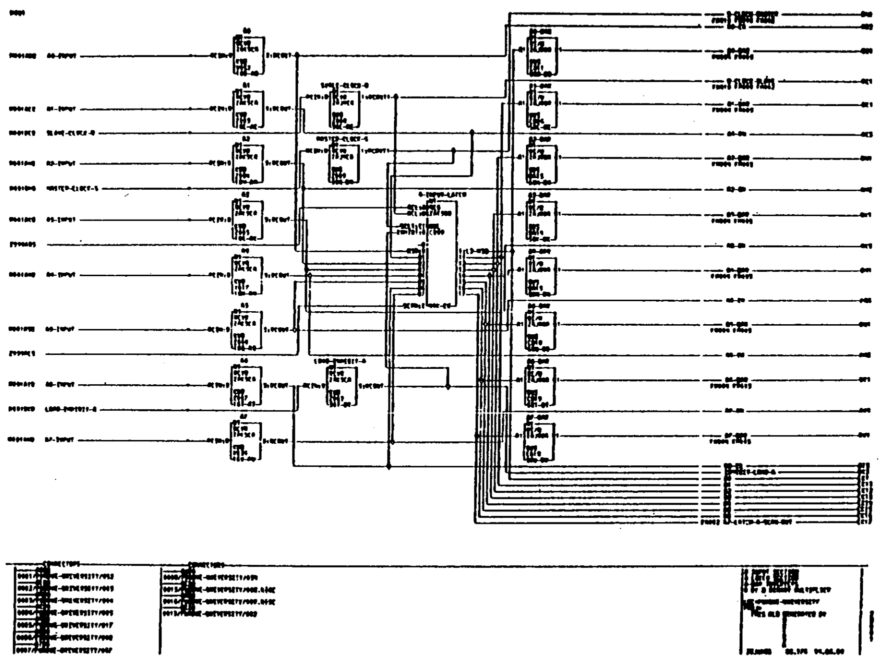


Fig. 10. Sheet of Logic with A-CLOCK and SCAN-IN Connected

pin of the register are not connected yet. Fig. 10 shows the same sheet of logic after the scan path has been wired.

The following steps were taken to produce the circuit modifications shown here:

1. A BDL/S description of the 8 X 8 multiplier without the scan path was produced by the MVISA system. The actual placement of cells making up of the 8 X 8 multiplier on the master image had already been performed by the automatic placement routines on MVISA.
2. The BDLS description of the circuit was used to produce an equivalent Lisp description by the MVISA/DEFT interface routines. These routines also do some additional processing to extract information anticipated to be useful to DEFT.
3. This description was then given to DEFT and the rules shown in Fig. 3 and the associated demons were run to produce a modified circuit.
4. A BDL/S description of the modified circuit was then produced and returned to the MVISA system.
5. The automatic placement and wiring routines were then rerun to produce the final chip layout shown in Fig. 6.

While the example demonstrated here is relatively simple, it serves to demonstrate the feasibility of our approach. Work is currently in progress on the rules and demons necessary to add a BILBO-like built-in self-test scheme to a circuit.

5. IMPLEMENTATION STATUS AND CONCLUSIONS

At the time of this writing the following parts of the DEFT system have been implemented:

1. The interface between DEFT and the MVISA System - MVISA uses IBM's Basic Design Language for Structure (BDL/S) to describe circuits. The DEFT/MVISA interface has two parts. The first part written using C and the Unix compiler writing tools Lex and Yacc, reads in a BDL/S description from MVISA and produces a Lisp description of the circuit. In addition to producing a BDL/S equivalent Lisp description of the circuit, the BDL/S to Lisp interface software also does some processing on the circuit to extract additional information that is anticipated to be useful to DEFT regardless of the circuit under consideration. Examples of such processing include the classification of all circuit elements by type and the generation of a Lisp array representing the layout. The second part of the MVISA/DEFT interface generates a BDL/S description of the circuit from the data-structures used by DEFT. This part of the interface is written in Lisp.
2. The modified Prolog interpreter incorporating the demon mechanism has been implemented using Franz Lisp. A set of Prolog rules and associated demons that implement a scheme to add a scan path to a circuit have been developed.
3. Part of the explanation mechanism has been implemented. Work is currently underway on the graphics routines to be used for this purpose.

The various parts of DEFT described above consist of approximately 2500 lines of C and 3800 lines of Lisp.

DEFT is a knowledge-based system that automatically modifies circuits to make them testable. The use of the knowledge-based approach was motivated by the desire to have a system that was easily modifiable and could provide the user with explanations. DEFT uses a Prolog-based knowledge representation scheme. The DEFT Prolog interpreter is written in Lisp and uses the demon mechanism to interface with Lisp data-structures and functions. The demon mechanism allows DEFT to benefit from a declarative knowledge representation scheme without paying a price in terms of efficiency. Initial experiences with DEFT have shown that this hybrid approach combining rule-based and algorithmic programming has indeed resulted in a modular and easily modifiable system. The Prolog rules describing a DFT scheme are written at a high level and are uncluttered by details regarding how the data-structures representing the circuit are actually manipulated. The high-level representation of DFT knowledge also has benefits related to debugging knowledge during system development: by tracing the rules that are being used to modify a circuit, the behavior of the system can be observed at a level that is meaningful to a person familiar with testability but not necessarily with the details of the data-structures and low level functions needed to implement a working design automation system.

The DEFT prototype described in this paper represents our initial efforts towards the goal of implementing a knowledge-based design for testability system. Work is currently underway to expand the capabilities of the system. Rules are being developed to check for violations of LSSD design rules and to modify circuits to remove these violations. Work is also underway on rules for a built-in self-test scheme based on BILBO. Directions for future research include development of the capability to accept circuit descriptions in VHDL. This will expand the range of circuits that DEFT can handle.

ACKNOWLEDGEMENTS

The authors are indebted to Bill Carlson and Kirk Smith for letting us use and modify their Prolog interpreter.

REFERENCES

[1] Abadir, M. and M.A. Breuer,"A Knowledge-Based System for Designing Testable VLSI Chips." IEEE Design and Test of Computers, Vol.2, Number 4, August 1985,pp.56-68

[2] Ando,H. "Testing VLSI with Random Access Scan." Proceedings of COMPCON Spring '80, 1980,pp.50-52. In "Selected Reprints in Logic Design for Testability." C.C. Timoc ed, IEEE Computer Society, 1984

[3] Barr A. and E.A. Feigenbaum,"The Handbook of Artificial Intelligence." Vol.1, HeurisTech Press, Stanford, California William Kaufmann,Inc., Los Altos, California 1981, pp.38-40

[4] Bennetts, R.G., "Design of Testable Logic Circuits." Addison-Wesley 1984

[5], Breuer, M.A. and X. Zhu, "A Knowledge-Based System for Selecting a Test Methodology for a PLA." Proceedings of the 22nd Design Automation Conference 1985,pp.259-265

[6] Brachman, R.J. and H.J. Levesque,"Readings In Knowledge Representation." Morgan Kaufmann Publishers Inc, Los Altos, California 1985

[7] Clocksin,W.F. and,C.S. Mellish,"Programming in Prolog," Springer-Verlag,Berlin Heidelberg New York 1981

[8] Eichelberger, E.B. and T.W. Williams,"A Logic Design Structure for LSI Testability." Proceedings of the 14th Design Automation Conference, 1977,pp.462-468; also in [35].

[9] Fujiwara,H. "Logic Testing and Design for Testability." MIT Press, 1985

[10] Funatsu,S. et al. "Designing Digital Circuits with Easily Testable Consideration." Proceedings of the International Test Conference 1978,pp.98-102; also in [35].

[11] Fung,H.S., S. Hirschhorn and R. Kulkarni, "Design for Testability in a Silicon Compilation Environment." Proceedings of the 22nd Design Automation Conference 1985,pp.190-196

[12] Fung,H.S.,S. Hirschhorn,"An Automatic DFT System for the Silc Silicon Compiler." IEEE Design and Test of Computers, Vol.3, No.1., Feb. 1986,pp.45-57

[13] Hayes-Roth,F.,D.A. Waterman and D.B. Lenat.,"Principles of Pattern-directed Inference Systems," in D.A. Waterman and F. Hayes-Roth, eds.,"Pattern-directed Inference Systems," 1978 Academic Press, New York,pp.577-601

[14] Horstmann,P.W., "Automation of the Design for Testability Using Logic Programming." Doctoral Thesis, Syracuse University, 1983.

[15] Horstmann,P.W., "Design for Testability Using Logic Programming" Proceedings of the 1983 International Test Conference,Phila.,PA.,pp.706-713

[16] Horstmann,P.W., "Expert Systems and Logic Programming for CAD." VLSI Design Magazine, November 1983,pp.37-46

[17] Horstmann,P.W. and E. Stabler,"Computer Aided Design Using Logic Programming." Proceedings of the 21st Design Automation Conference, 1984,pp.144-151

[18] Horstmann,P.W.,"A Knowledge-Based System Using Design for Testability Rules," Proceedings of the 14th International Conference on Fault-Tolerant Computing, Orlando, Florida, June 1984, pp.278-284

[19] Hutchinson,S.A. and A.C. Kak, "FProlog: A language to Integrate Logic and Functional Programming For Automated Assembly." Proceedings of the IEEE International Conference on Robotics and Automation,1986, pp.904-908

[20] Komonytsky,D."LSI Self_Test Using Level-Sensitive Scan Design and Signature Analysis." Proceedings of the 1982 International Test Conference, pp.441-424

[21] Konemann,B., J. Mucha and G.Zwiehoff,"Built-In Logic Block Observation Techniques." Proceedings of the 1979 International Test Conference,pp.37-41

[22] Konemann,B., J. Mucha, and G. Zwiehoff,"Built-In Self-Test For Complex Digital Circuits." IEEE Journal of Solid-State Circuits, Vol.SC-15, No.3, June 1980

[23] McCluskey, E.J.,"Built-In Self-Test Techniques." IEEE Design and Test of Computers, Vol.2, No.2, April 1985

[24] McCluskey, E.J.,"Built-In Self-Test Structures." IEEE Design and Test of Computers, Vol.2, No.2, April 1985

[25] IBM Federal Systems Division,"Overview, Logic Entry, Design for Testability. Master Image Designer's Guide." , Manassas, Virginia, 1985

[26] Roth,J.P. "Diagnosis of Automata Failures: A Calculus and a Method." IBM Journal of Research and Development No.10, Oct. 1966,pp.278-281

[27] Samad,M.A. and J.A.B. Fortes,"Explanation Capabilities in DEFT - A Design-For-Testability Expert System." Proceedings of the International Test Conference, 1986 [to be published]

[28] Timoc,C.C.,"Selected reprints on Logic Design for Testability." IEEE Computer Society 1984

[29] Williams,T.W., K.P. Parker,"Design for Testability - A Survey." Proceedings of the IEEE,Vol.71,No.1,January 1983,pp.98-112

EXPERIENCES IN PROLOG-BASED DFT RULE CHECKING

Gianpiero CABODI Paolo CAMURATI Paolo PRINETTO

Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi 24, I-10129 Torino Italy

Abstract

Since testing of VLSI is becoming increasingly important, many methodologies have been introduced to enhance Design For Testability (DFT). Manual verification is becoming more and more difficult as the size of designs increases and automation is thus essential. A method and a tool to check whether a given hardware design verifies or not a set of high level DFT rules (LSSD and BILBO) is presented.

The work reported in this paper is based on Prolog. This logic programming language is used in many ways: as a basis for a predicative representation of hardware, as a means to encode the knowledge of DFT techniques, as an interpreter to support the implementation of a frame- and rule-based expert system kernel. Eventually conclusions are drawn on its effectiveness.

1 INTRODUCTION

The increasing complexity in VLSI challenges CAD tools designers, since more and more computer aid is necessary to take full advantage of technological improvements.

A field in which CAD is essential is testing, not only because of the need for Automatic Test Equipments (ATE) and Automatic Test Pattern Generators (ATPG), but also because cost-effective testing requires VLSI circuits to be designed to guarantee testability.

Many approaches to Design for Testability have been proposed in recent years [1] and are widely accepted in both the industrial and academic world. CAD tools designers are now moving in two major directions: they try to include such DFT techniques in their logic synthesis systems or they produce automatic compliance verification programs. As far as the first entry is concerned, a lot of work is going on, as reported in [2], [3], [4], [5], and [6]. This approach is being confronted by the same problems of synthesis and silicon compilation. As far as verification is concerned, its goals seem to be more immediate. Algorithmic methods have already been introduced [7], but their main limit resides in the scarce flexibility of the system with respect to the knowledge it encodes. Since knowledge is rapidly varying, it seems useful to gather a large knowledge base and exploit it resorting to knowledge engineering techniques [4]. Most DFT techniques are already expressed in a rule form [1], therefore rules seem to be very suitable as a knowledge representation formalism. Prolog [8] provides an easy way to express rules and clauses representing knowledge both about DFT methods and about hardware, and, moreover, puts at the user's disposal a built-in inference engine. This separation of inference and control strategies from the knowledge base allows the CAD tool designer to concentrate on the latter, leaving utilization problems apart, at least in the preliminary phase.

This paper presents a prototypical DFT rule verifier using Level Sensitive Scan Design (LSSD) [9] and BILBO [10].

In the sequel, we shall cover the following topics: predicative representation of hardware, DFT rules in Prolog form, the use of the Prolog reasoning mechanism to verify designs, and a frame- and rule-based expert system kernel implemented in Prolog. The project's current status and future plans are described. Conclusions are eventually drawn justifying the validity of this choice, discussing its limits and possible improvements.

2 HARDWARE REPRESENTATION FORMALISMS

It is possible to describe hardware, i.e., the knowledge we have about the circuit in several ways, among which Hardware Description Languages (HDLs) [11] and clauses are the most widely used.

HDLs have been introduced to describe models to be used in simulation and test pattern generation. They describe hardware structurally and/or behaviourally, possibly resorting to different abstraction levels.

Many efforts are currently directed to provide efficient and effective executable

CH2345-7/86/0000/0909$01.00 © 1986 IEEE

specification formalisms. Such executable specifications can be used for simulation as well as verification purposes. Most formalisms are related to first-order predicate calculus and/or to subsets of logic [12], [13], [14].

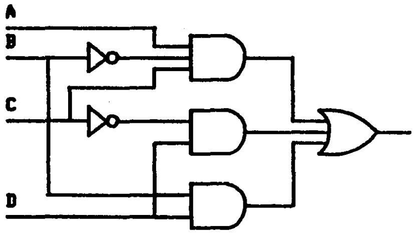
A very popular form of knowledge representation uses rules and Horn clauses [14], [15], [16] since they are Prolog-like and Prolog provides a simple, yet effective inference mechanism. Some formalisms used to represent hardware in rule form are briefly illustrated in the sequel. A general classification is adopted to distinguish them according to their main characteristics into Functional, Extensional, and Definitional methods.

2.1 The Functional Method

The functional method [14] can easily represent combinational acyclic circuits in which a single output signal is function of several input signals.

The function symbol together with its arguments identifies the output of the block itself.

The connection relationship between blocks is purely functional: the syntactic form of a given term determines its connections. An example is shown in Fig. 1.



or(and(a,not(b),c),and(not(c),d),and(b,d))

Fig. 1

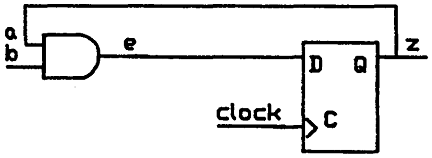
Input signals are named by a constant symbol. Blocks are named by the corresponding function symbols. Functions have an arbitrary number of inputs, but only one output. This formalism allows processing the circuit by recursive descents, transforming a given term in another one.

There are two main disadvantages to this technique. First, only acyclic circuits can be described, and this precludes the specification of realistic sequential ones. Second, a separate expression must be used to represent each output of a circuit.

2.2 The Extensional Method

The extensional method [15], [16] represents blocks and connections as clauses in which constants are used to indicate the connections between modules.

In the example of Fig. 2 the "module" predicate has three arguments, describing its function, its input list, and its output list. Connections between modules are specified by the "connect" predicate. This predicate has two arguments, each indicating the module type and the connected port.



```
module(and,[a,b],[e])
module(latch,[D,C],[Q])
connect(latch(Q),z)
connect(and(a),z)
connect(and(e),latch(D))
connect(clock,latch(C))
```

Fig. 2

The circuit is thus described as a set of module and connection predicates.

It is usual to consider the module and connection relations as templates, and their arguments as variables standing for any instance of the template.

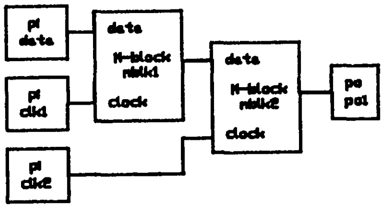
This method can accommodate arbitrary types of circuits such as multiple output cyclic circuits. However, the main disadvantage stems from the fact that the modules are not represented by a single term, rather they are represented extensionally, with no syntactic relationship among them.

Variations of this formalism are used in a number of systems. We shall now focus the attention on a particular one introduced in [15], [17].

The design description contains two general types of statements. The first kind of clauses describes the nodes' functions and their interconnections. Each clause contains a node name, a node function, a list of the come_from nodes (and pin names), and a list of go_to nodes (and pin names).

The second type of statements consists of clauses stating the special behaviour of the node itself. This is particularly useful when modelling clock inputs.

A simple description with this method is shown in Fig. 3 [17].



```
data(p1,(),(connect(mblk1,data,_))).
clk1(pi,(),(connect(mblk1,clock,_))).
clk2(pi,(),(connect(mblk2,clock,_))).
clk1(clock).
clk2(clock).
mblk1(m-block,(connect(data,data,_),
               connect(clk1,clock,_)),
     (connect(mblk2,data,_))).
mblk2(m-block,(connect(mblk1,data,_),
               connect(clk2,clock,_)),
     (connect(po1,_,_))).
po1(po,(connect(mblk2,_,_)),()).
```

Fig. 3

Since every block has one output only, the output's name is the same as the block's one. It is possible to modify this formalism to include inputs and outputs of the block in its template.

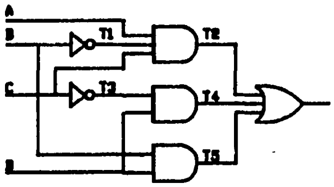
2.3 The Definitional Method

In the definitional method [12], [18] blocks having n ports are represented as n_ary predicate symbols.

Modules are described by Horn clauses whose head is the module to be defined, and whose body is a composition of either already defined or primitive modules.

The ":-" operator is interpreted as "is defined by". The order of modules in the body of the clause is not important.

A simple description with this method is shown in Fig. 4.



```
comb(A,B,C,D,E):-not(B,T1),and(A,T1,C,T2),
                 not(C,T3), and(T3,D,T4),
                 and(B,D,T5),or(T2,T4,T5,E).
```

Fig. 4

Primitive modules are a priori defined by rules describing their behaviours.

This method has numerous advantages:

- the circuits may be directly represented by Prolog clauses;
- the module's name is explicitly part of the specification and this allows easy modular decomposition;
- internal connections are named by variables which do not appear in the head of the clause.

2.4 Our Approach

The approach we adopted combines some features of the definitional and extensional methods. This approach is especially tailored to the representation of Finite State Machines, both in Moore and in Mealy form.

The circuits we consider consist in combinational networks and registers as shown in Fig. 5.

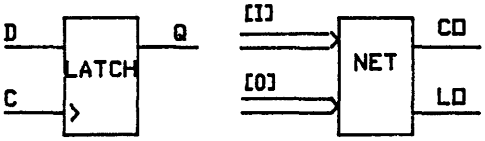


Fig. 5

We are not concerned with the internal structure of the combinational networks, but only with their connection to registers. This consideration leads to the introduction of two simple predicates to describe the circuit in a definitional way: REGISTER()

and NET(). "REGISTER" describes a block having a pulse input (C), a level input (D) and a level output (Q).

The register is described by following predicate:

```
register(D,C,Q).
```

"NET" describes logic networks with n level inputs (n>=0), m clock inputs (m>=0) and either a level output or a clock output.

The network is described by following predicate:

```
net ( [I], [C], LO, CO )
```

where:

```
[I] is the level input list
[C] is the pulse input list
LO  is the level output
CO  is the pulse output.
```

Lists are used to represent networks with multiple inputs. Since each network has only one output, either CO or LO is missing.

It is necessary to distinguish the input clocks of the circuit, hereinafter called Primary Input Clocks (PICs) from other inputs. An ad hoc predicate PIC(x) has therefore been introduced to define them.

3 DFT RULES IN PROLOG FORM

The Prolog clauses which represent LSSD and BILBO methodologies may be found in [19]. In this paragraph we present a flash on a particular rule, just to give the taste of how it is represented. All rules in Prolog form return true if and only if the corresponding DFT rule is violated. It would have been possible to rewrite all rules in order to have true returned only when the corresponding DFT rules were satisfied. Such alternative approach is much more complex and the readability of the system's output is reduced.

We shall consider the following LSSD rule:

> "a latch X may gate a clock Ci to produce a gated clock Cig which drives another latch Y if, and only if, clock Cig does not clock latch X, where Cig is any clock produced from Ci."

This rule may be violated if and only if the following situation arises:

> "there are two latches and one combinational network connected in such a way that the input clock of the second latch is fed by the output of the combinational network one of whose inputs is the output of the first latch".

This rule is effectively violated if the input clock of the first latch is equal to any of the input clocks of the combinational network. The structural situation is shown in Fig. 6, while Fig. 7 reports the corresponding Prolog rule.

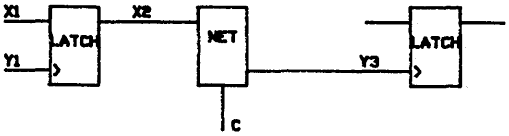


Fig. 6

```
lssd22(X1,X2) :- latch(X1,Y1,X2),
                 network4(C,X2,Y3),
                 latch(_,Y3,_),
                 clock(X1,Y1,C).
```

Fig. 7

The first three predicates on the right side check the applicability of the rule itself. The fourth checks if the rule is violated. More in details, predicate "network4" searches for a net in the circuit description that is connected to the output of the latch(X1,Y1,X2):

```
network4(C,X2,Y3) :- net(T,C,_,Y3),
                     input(X2,T).
```

The "clock" predicate tests for violation looking for equal or dependent clock inputs of the first latch and of the combinational network:

```
clock(X1,Y1,[]).

clock(X1,Y1,[H|T]) :- case(X1,Y1,H),
                      clock(X1,Y1,T).
```

Equality or functional dependency between clocks is encoded in the "case" predicate [19].

4 THE FRAME- AND RULE-BASED DFT EXPERT SYSTEM KERNEL

In the initial phase of this project we decided to use Prolog both as a specification and as implementation language because of its simplicity, power, and availability of a built-in interpreter. All these features allowed a rapid prototyping of the system. At the end of this phase, it contained approximately 60 clauses, 40 for LSSD and 20 for BILBO. The prototype was first developed on an IBM PC and later migrated to a VAX 11/780 running VMS. The only explanation facilities available were messages dispatched by the verifier and the Prolog interpreter's tracing facilities. The main theoretical limits of this approach were that knowledge could be represented only under the form of rules and that the Prolog interpreter has a fixed backward reasoning mechanism. These limits have been encountered by many researchers working on analogous fields and three directions have been proposed to overcome them [20]:

- to extend Prolog both as a language and as an interpreter,
- to build a meta-interpreter over Prolog,
- to define a new, higher level form of DFT knowledge representation and to implement a preprocessor which translates it into executable Prolog clauses.

The first two approaches were difficult and complex, while, as far as the third is concerned, we could benefit from a preprocessor prototype developed at the University of Turin [20]. From the toolmaker's point of view, DFT knowledge may be represented as a hierarchical frame structure, where the body of each frame consists of a production system. This allows to structure DFT knowledge modularly and makes it easier to add new methodologies or to change rules. Moreover, the toolmaker may specify whether such rules must be interpreted either with forward or backward reasoning. The preprocessor translates such specifications and, adding appropriate control statements, generates the actual clauses for the Prolog interpreter. Since we plan to integrate the DFT verifier into a knowledge-based CAD tool, particular attention has been paid to make frame control strategies as general as possible, supporting width-first, depth-first, and heuristic search.

5 CONCLUSIONS

The DFT rule checker we developed is not yet a full system, since it takes into account only 2 methodologies, it relies on a simplified formalism to describe hardware, and it has a reduced user interface. We plan to extend it to other DFT methodologies and to integrate it into a complete CAD system, comprehensive of synthesis, verification, and redesign facilities. This is why particular attention has been paid to the frame- and rule-based expert system which will represent the kernel of the overall CAD tool.

In the preliminary phase, the prototype has been used to verify simple circuits, such as 2 bit binary counters and a systolic priority queue.

We can conclude from our experience that rules are adequate to represent LSSD and BILBO designs and that they may be easily expressed in Prolog. Anyway, the direct description of designs in Prolog yields rough results or needs an excessive number of clauses to represent hardware with the same level of detail of an HDL. Therefore we plan to introduce a high-level language, possibly an already existing HDL, for hardware descriptions, while keeping Prolog clauses as the internal, executable form.

Although it is difficult to produce figures of merit for the current prototype of the system, our experience demonstrates that the combined use of some Artificial Intelligence techniques and DFT methodologies is a good approach to the testability problem of VLSI devices.

6 REFERENCES

[1] T.W. Williams, K.P. Parker: "Design for Testability - a Survey" IEEE Transactions on Computers, vol. C-31, no. 1, January 1982, pp 2-15.

[2] V.D. Agrawal, S.K. Jail, D.M. Singer: "A CAD System for Design for Testability" VLSI Design, pp 46-54, October 1984.

[3] V.D. Agrawal et al.: "Automation in Design for Testability" Proc. 1984 Custom Integrated Circuits Conference, pp 159-163.

[4] M.S. Abadir, M.A. Breuer: "A knowledge-based system for designing testable VLSI chips" IEEE Design & Test, August 1985, pp 56-68.

[5] T.J. Kowalski, D.J. Geiger, W.H. Wolf, W. Fichtner: "The VLSI Design Automation Assistant: from algorithms to silicon" IEEE Design & Test, August 1985, pp 33-43.

[6] H.S. Fung, S. Hirschhorn: "An automatic DFT system for the Silc silicon compiler" IEEE Design & Test, February 1986, pp 45-57.

[7] H.C. Godoy, G.B. Franklin, P.S. Bottorff: "Automatic checking of logic design structures for compliance with testability ground rules" 14th Design Automation Conference, June 20-22 1977, New Orleans, USA, pp 469-478.

[8] W.F. Clocksin, C.S. Mellish: "Programming in Prolog" Springer-Verlag, 1981.

[9] T.W. Williams: "Design for Testability" NATO Advanced Study Institute on Computer Design Aids for VLSI Circuits,

SOGESTA, Urbino, Italy, July 1980.

[10] B. Koenemann, J. Mucha, G. Zwiehoff: "Built-in logic block observation techniques" Dig. 1979 Test Conference, October 1979, pp 37-41.

[11] M.R. Barbacci, T. Uehara: "Computer Hardware Description Languages: the Bridge between Software and Hardware" IEEE Computer, vol 18, Number 2, February 1985, pp 6-8.

[12] M.J.C. Gordon: "LCF-LSM: a system for specifying and verifying hardware" Technical Report 41, Computer Laboratory, University of Cambridge, UK, 1984.

[13] D.D. Sidhu: "Logic Programming Applied to Hardware Design Specification and Verification" 17th Annual Microprogramming Workshop, New Orleans, USA, November 1984, pp 309-313.

[14] L. Wos, E. Lusk, R. Overbeek, J. Boyle: "Automated Reasoning" Prentice Hall, 1984.

[15] P.W. Horstmann: "A Knowledge-Based System Using Design For Testability Rules" 14th Int. Conf. on Fault Tolerant Computing, June 20-22 1984, Kissimmee, USA, pp 278-284.

[16] H.G. Barrow: "VERIFY: a program for proving correctness of digital hardware designs" Artificial Intelligence, Vol.24, 1984, pp 437-491.

[17] P.W. Horstmann: "Expert Systems and Logic Programming for CAD" VLSI Design, pp 37-46, November 1983.

[18] B. Moszkowski: "A temporal Logic for multi-level reasoning about hardware" CHDL '83: IFIP 6th Int. Symposium on Computer Hardware Description Languages and their Applications, Pittsburgh, PA (USA), May 1983, pp 79-90.

[19] GP. Cabodi, P. Camurati, P. Prinetto: "DFT Rules Verification" Internal Report DAI 14/85, Dipartimento di Automatica e Informatica, Politecnico di Torino, October 1985.

[20] L. Console, G. Rossi: "Implementing inference strategies in Prolog-based expert systems" 8th European Meeting on Cybernetics and System Research, Vienna (Austria), 1986.

A RULE BASED SYSTEM FOR THE OPTIMAL STATE ASSIGNMENT OF CONTROLLERS

E. DUPONT, J. IDT, G. SAUCIER

Laboratoire "Circuits et Systêmes" IMAG
46, avenue F. Viallet 38031 GRENOBLE FRANCE

ABSTRACT :
We present a rule based system for choosing a state assignment which enables the layout of controllers on PLAs of minimal area. The knowledge base is built from the study of the elementary simplification mechanisms in the next state equations and in the output equations. The comparison between the state graph and the rule base yields a set of constraints between the codes of the states. Then the system solves the problem of finding a state assignment consistent with the constraints, and yields the simplified equations.

INTRODUCTION

Automatic synthesis of controllers has been extensively studied in the literature.

The starting point is the description of the controller in an abstract form, in a high level language or through a classical flowchart. The final implementation may use different types of memory points and different types of logic (random logic, PLAs, ROMs...). The logic, essentially for PLAs, may furthermore be partitioned according to precise constraints (maximum size, regularity etc...). In order to minimize the controller area, the crucial problem is the choice of the state assignment.

The minimization of the next state equations or output equations has been extensively studied in the past [1..8]. More recently, heuristics applied to larger and more "realistic" controllers have been presented [9, 10]. Considered from a practical point of view, it is obvious that the optimization process depends a lot on the target structure (microprogrammed or PLA based controller, relative importance of the next state equation part and of the output equation part etc...). Therefore, the optimization rules for the state encoding must be parameterized. This is obtained in our approach by applying an artificial intelligence technique. The constraints on the state encoding are expressed by a set of rules. These rules depend on the choice of the memory points and are chosen by the designer among a set of rules. He can also create and add his own rules. An inference system applies these rules and leads to an optimal encoding. This software is a part of a larger software dealing with all the aspects of controller design. This software is organized around a high-level, object oriented language CADOC [14,15] and leads to several hardware implementations.

I - CONTROLLER DESCRIPTION

The controller may be described, as said before, in different manners (abstract descriptions like regular expressions, high level language, control flow graph). The example that we shall take throughout this paper is an arbiter circuit for MC 68000 microprocessor : it is described by its control flowgraph (figure 1). The circles are the states of a Mealy machine. The transitions of the machine are labelled with the input conditions leading to the next states and with the output signals associated with the couples (state, inputs). The meaning of the graph is illustrated using state 3 of figure 1 :

CH2345-7/86/0000/0915$01.00 © 1986 IEEE

- Transition from state 3 to state 4 if dtak is true ; the output signals bgak, as, uds, lds are emitted.
- Transition frome state 3 to state 3 if dtak is false , the signals bgak, as, uds, lds are emitted.

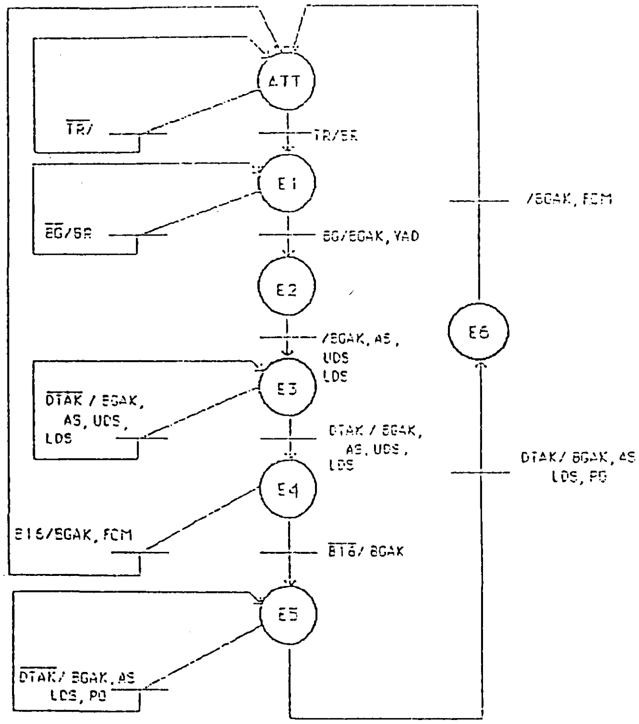


Figure 1

II - SET OF RULES FOR THE STATE ASSIGNMENT

The set of rules for the state assignment depends on the type of memory points (JK, RS, D flip-flops) and the designer's objective. We shall give as an example the set of rules for an implementation on D flip-flop and PLA. A first subset of rules concerns the next state equations, a second one the output equations. The weight given to these two aspects depends on the designer's choice or on the relative importance of the logic blocks.

2.1 - Notations

Let n denote the number of bits used for the state assignment.

The code of state i is denoted C(i).

$\{Y_i\}$ denotes the set of state variables equal to 1 in the code of state i.

Example :

$\{Y_i\} = \{y_1, y_3\}$ if C(i) = (1010)

f(i) denotes the product term associated with the state i.

$$f(i) = \prod_{\substack{j=1 \\ y_j \in \{Y_i\}}}^{n} y_j \prod_{\substack{k=1 \\ y_k \notin \{Y_i\}}}^{n} \overline{y_k}$$

The "always true" input condition is denoted by 1 in the figures.

Inclusion : C(i) is included in C(j) if $\{Y_i\} \subset \{Y_j\}$
Notation : C(i) c C(j).

Adjacence : C(i) and C(j) are adjacent if they differ only by one bit.

Notation : C(i) adj C(j).

In this case, the sum f(i)+f(j) reduces to one minterm, denoted $\lfloor f(i)+f(j) \rfloor$.

2.2 - Example of optimization rules for next state equations for an implementation on PLA with D flip-flops

The first two rules shown here deal with the "fork" and "join" situations in a flowchart and are based on the contribution of a node in the next state equations. It is well known that for D flip-flops the next state equations have the following form :

$$\{Yi\} = \sum_{j \in (\text{antecedent states of state } i)} f(j) \text{ x (input condition of the arc } j \to i)$$

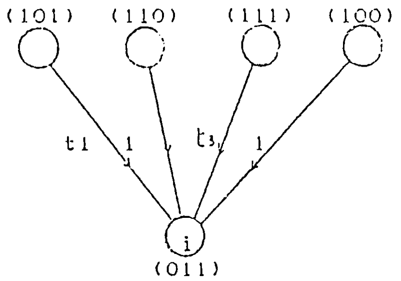


Figure 2

The contribution of state i in Figure 2 to the equations is :

$$Y_2,Y_3 = t_1Y_1\overline{Y}_2Y_3 + Y_1Y_2\overline{Y}_3 + t_3Y_1Y_2Y_3 + Y_1\overline{Y}_2\overline{Y}_3$$

To minimize the contribution of a node, rules a and b may be applied.

<u>RULE a</u>

If a node is connected to p antecedents ($2^{k-1} < p < 2^k$) with the same input condition on the arcs,

Then : - assign to k out the n bits of the code of the antecedents a p-subset of the 2^k codes.

- do not use the (2^k-p) remaining codes in the state assignment.
- let the other (n-k) bits be "invariant".

Example : 6 states are connected to state i by arcs with transition "1", in a graph with 50 states (encoding on 6 bits) (Figure 3).

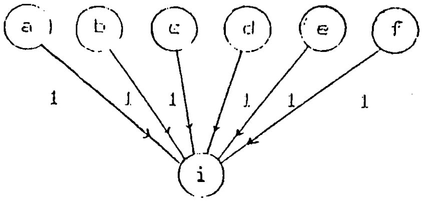


Figure 3

A possible encoding is :

0 0 0 0 1 1
0 0 1 0 1 1
0 1 0 0 1 1
1 0 0 0 1 1
1 1 0 0 1 1
0 1 1 0 1 1

The codes : (101011) (111011) cannot be used for the state assigment.

The contribution of state i is : $\overline{Y}_4\ Y_5\ Y_6$

Remark : - This rule allows p product terms to be replaced by one product term

- The degrees of freedom are the position of the "invariant" bits and the code assignment within the set of k bits.

<u>RULE b</u>

If a node is connected to p successors ($p = 2^k$) with a sum of input conditions appearing on the arcs equal to 1 then :

- assign to the successors an exhaustive encoding on k out of n bits
- let the (n-k) remaining bits be invariant or "replicate" one or several of the first k bits.

Example :

A possible state assignment is (on 4 bits) :

i	0 0 1 0
j	0 1 1 0
k	1 0 1 1
l	1 1 1 1

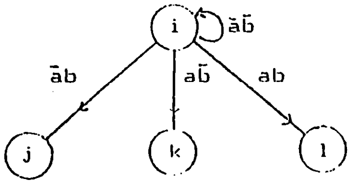


Figure 4

Equations : $Y_1 = a.f(i)$

$Y_2 = b.f(i)$

$Y_3 = f(i)$

$Y_4 = a.f(i)$

Remark : we generate three product terms af(i), b(fi) and f(i) instead of four product terms.

Remark : - This rule allows $p = 2^k$ product terms to be replaced by (k+1) product terms. Only k input conditions are appearing in the equations, instead of 2^k.

- The degrees of freedom are the position and value of the "non significant" bits.

- In the case k = 1, the conclusion of rule b is : C(j) c C(k) or C(k) c C(j), j and k being the successors.

RULE c

If p nodes are connected through a path with the same input conditions on the transitions, then for p' nodes ($p' = 2^k-1 < p$) assign on k of the n bits an optimal sequence of codes (see (13)); the (n-k) remaining bits are invariant.

This rule leads to 2(k-1) product terms instead of 2^k-2.

We can notice that, if k=2, $2(k-1) = 2^k-2 = 2$. Rule c must be restricted to k>3, ie p'>7

The degrees of freedom are the positions and the values of the "non significant" (n-k) bits. As an example, let us consider the sequence of figure 5.

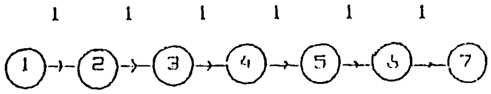


Figure 5

A possible state assignment is (on 5 bits) :

1	1 0 0 1 0
2	0 1 0 1 0
3	1 0 1 1 0
4	1 1 0 1 0
5	1 1 1 1 0
6	0 1 1 1 0
7	0 0 1 1 0

We have 4 product terms instead of 6.

OTHER RULES

Other rules, which address more complex situations, have been defined. Some graphical examples are shown :

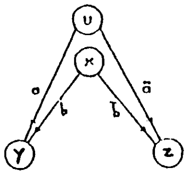
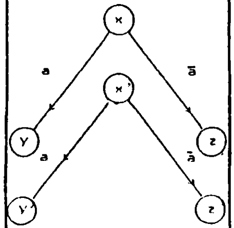
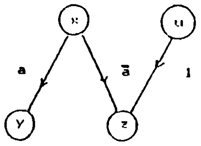
	Encoding constraints : C(u) adj C(x) C(z)cC(y) OR C(y)cC(z) 3 product terms : af(u), bf(x), [f(u)+f(x)] OR āf(u), b̄f(x), [f(u)+f(u)]
	Encoding constraints C(x) adj C(x') C(y)cC(z) AND C(y')cC(z') OR C(z)cC(y) AND C(z')cC(y') C(y) XOR C(z)=C(y') XOR C(z') 3 product terms : āf(x), āf(x') [f(x)+f(x')] OR af(x), af(x') [f(x)+f(x')]
	Encoding constraints C(x) adj C(u) C(z) c C(y) 2 product terms : af(x), [f(x)+f(u)]

Figure 6

All the rules are not listed here. They may address more complex situations or dedicated targets of the designer (for example, partitioned PLAs).

2.3 - Rules for the output equations

Two examples of basic rules will be given here. The second rule shows the dependancy and the contradiction which may exist between two rules.

RULE a'

If p states send (for the same input condition in a Mealy machine) one or more identical control signals then for $2^{k-1}<p\leq 2^k$

- assign to k of the n bits the 2^k possible values
- assign to the (n-k) bits an invariant code.

Remark

The gain is weighted by the number of identical outputs associated with the nodes.

RULE b'

If one or several control signals are sent by one and only one state then do not apply rules a and b to this state.

This last rule shows that the output rules may conflict with the next state equations rules.

III - GENERAL STRATEGY OF THE INFERENCE SYSTEM

The flowchart of the system is given on Figure 7. The system is organized in modules $M_1 \ldots M_6$. The rule base is independent of the modules, so that the set of rules may be enlarged with new rules, and that only part of the rules may be used.

The first **module M_1** asks for the designer's graph, strategy and "initial" constraints (for example, which state must be encoded (0...0)). M_1 performs a selection of the suitable rules in the rule base, according to the chosen target. The source code is 200 LISP lines.

A second **module M_2** performs inferences between the selected rules and the graph. All selected rules are tried, and in case of success, their consequences are memorized.

The heuristic is the following : try first the rules liable to yield a high gain (for example, rule c) ; compute the gains ; if these gains are small (case of loose constraints), apply the more complex rules (figure 6). The module M_2 also collects the "external" constraints provided by the designer. Its output is a set of constraints (either between the codes of individual states, or between the encodings of groups of states) associated with gains and with symbolic simplified equations.

The source code is 1000 LISP lines.

The **module M_3** then analyses these results : as a matter of fact, satisfying different constraints may lead to conflicts ; the effective state assignment is a complex problem and requires a high CPU time.

M_3 has to deal with two sorts of constraints : inclusions and adjacences between codes. Some constraints are incoherent, whatever the number of bits may be, others are related to the number of bits. Let us give two examples of such impossibilities :

(C(0)cC(1), C(1)cC(2) ; C(2)cC(3) ; C(3)cC(4))

is impossible in the case of an encoding with 3 bits, but possible on 4 bits.

$$\left.\begin{array}{l}\text{C(1) adjacent to C(2)}\\ \text{C(2) adjacent to C(3)}\\ \text{C(1) adjacent to C(3)}\end{array}\right] \text{ is impossible.}$$

The result of M_3 is a new set of constraints, rid of such impossibilities.

The task of the following **module M_4** is to find a state assignment satisfying all the constraints, if there exists one. The procedure is not quite the same according to the size of the problem, but it is based on the same technique : try to encode the states the one after the other, while satisfying the constraints at each step ; backtrack in case of failure, until either a solution is found or all combinations have been tried. The problem is to try all possible solutions as far as they are not equivalent. At each step equivalent choices are identified to avoid trials of equivalent encodings after backtracking.

In the case of large size circuits, a two-level encoding and backtrack mechanism is used : first find a possible partial encoding of the different sets of states, then assign codes in each set. The source code is about 1200 LISP Lines.
If no solution is found, the program asks for the designer's advice :
- try to code on n+1 bits, instead of n ?
- eliminate some constraints according to gains using **module M_5** and return to M4.
The source code is about 500 LISP lines.

A last **module M_6** collects the results, fetches the simplified equations related to constraints which have been satisfied, and deduces the equations. The source code is about 700 LISP lines.

Its result is an encoding of the states and the equations to be implemented. These equations can be furthermore processed by a rule based boolean minimizer and lead to an automatic layout on PLAs, gate arrays or complex MOS gates [13].

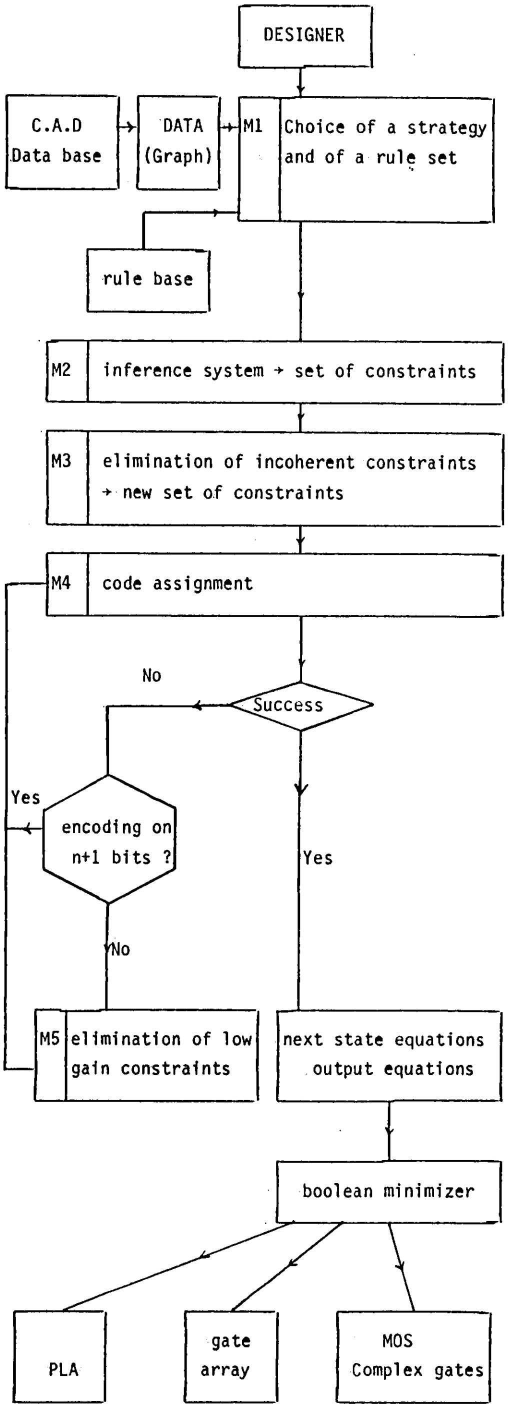

DESIGNER
C.A.D Data base
DATA (Graph)
M1
Choice of a strategy and of a rule set
rule base
M2
inference system → set of constraints
M3
elimination of incoherent constraints → new set of constraints
M4
code assignment
No
Success
Yes
encoding on n+1 bits ?
No
Yes
M5
elimination of low gain constraints
next state equations output equations
boolean minimizer
PLA
gate array
MOS Complex gates

Figure 7 : Flowchart

IV - EXAMPLE : ENCODING OF THE GRAPH OF AN ARBITER CIRCUIT

The state graph of the case study is given on figure 1 ; two solutions are proposed. In section 4.1 only the next state equations are optimized. In section 4.2 the output equations are also taken into account.

4.1 - Next state equations optimization

An external encoding constraint is given by the designer : the state 0 (ATT) must be encoded with (0..0). The graph has 7 states, which will be denoted (0,1,...6).
We shall try to code on 3 bits.
The application of rule b is yielding the constraints of table 1.
The application of the complex rules described on figure 6 are yielding the constraints of tables 2 and 3.

ANTECEDENT STATES	SUCCESSOR STATES	CONSTRAINTS	MINTERNS GENERATED
0	0 1	C(1) c C(0) OR C(0) c C(1)	($\overline{tr}$f(0) and f(0)) OR (trf(0) and f(0))
1	1 2	C(1) c C(2) OR C(2) c C(1)	(bgf(1) and f(1)) OR ($\overline{bg}$f(1) and f(1))
3	4 3	C(3) c C(4) OR C(4) c C(3)	(dtak f(3), f(3)) OR ($\overline{dtak}$ f(3), f(3))
4	0 5	C(0) c C(5) OR C(5) c C(0)	(b_{16} f(4), f(4)) OR ($\overline{b_{16}}$ f(4), f(4))
5	5 6	C(5) c C(6) OR C(6) c C(5)	(dtak f(5), f(5)) OR ($\overline{dtak}$ f(5), f(5))

Table 1

ANTECEDENT STATE	SUCCESSOR STATE	CONSTRAINTS	MINTERMS GENERATED

3 , 5	(4 3),
	(6 5)
		C(5) adj C(3)
		[C(3)cC(4) and C(5)cC(6)] OR
		[C(4)cC(3) and C(6)cC(5)]
		C(3)XOR C(4) = C(5)XOR C(6)
			3 minterms :
			(f(3), f(5)) AND
			dtak [f(3) + f(5)] OR
			f(3), f(5) AND
			dtak [f(3) + f(5)]

Table 2

ANTECEDENT STATE	SUCCESSOR STATE	CONSTRAINTS	MINTERMS GENERATED

2 3	(3) and
	(3 4)
		C(2) adj C(3)
		C(3) c C(4)
			2 minterms
			[f(2) + f(3)]
			dtak. f(3)

Table 3

The treatment of the list of constraints yields the following encoding (with 3 bits).

STATE	Y_1	Y_2	Y_3
0	0	0	0
1	0	1	1
2	0	0	1
3	1	0	1
4	1	1	1
5	1	0	0
6	1	1	0

Table 4 : encoding

This symbolic next state equations are :

$$Y_1 = [f(2) + f(3)] + \overline{b_{16}}\ f(4) + f(5)$$

$$Y_2 = bg.\ f(1) + dtak\ [f(3) + f(5)]$$

$$Y_3 = tr.\ f(0) + f(1) + [f(2) + f(3)]$$

There are 7 different product terms :

$[f(2) + f(3)]$	ie $\overline{Y}_2\ Y_3$
$\overline{b_{16}}.f(4)$	ie $\overline{b_{16}}\ Y_1Y_2Y_3$
$f(5)$	ie $Y_1\overline{Y}_2\overline{Y_3}$
$bg.f(1)$	ie $bg.\ \overline{Y}_1Y_2Y_3$
$dtak\ [f(3)+f(5)]$	ie $dtak\ Y_1\overline{Y_2}$
$tr.\ f(0)$	ie $tr.\ \overline{Y}_1\overline{Y}_2\overline{Y}_3$
$f(1)$	ie $\overline{Y}_1Y_2Y_3$

Only 4 input conditions appear in the equations, though 8 input conditions are given in the graph.

4.2 - Taking into account the output equations

The strategy chosen here is to look for a minimization of the next state equations, while taking into account the output equations.
The method is to prevent the applications of the next state assignment rules which would lead to the suppression, in the next state equations, of a product term which is a prime implicant of the output equations.
The symbolic equations of the outputs are :

$\mathrm{fcm} = b_{16}\ f(4) + f(6)$

$\mathrm{br} = \mathrm{tr}\ f(0) + \overline{\mathrm{bg}}\ f(1)$

$\mathrm{vad} = \mathrm{bg}\ f(1)$

$\mathrm{uds} = f(2) + \overline{\mathrm{dtak}}\ f(3) + \mathrm{dtak}\ f(3)$

$\mathrm{bgak} = b_{16}\ f(4)+f(6)+\mathrm{bg}\ f(1)+f(2)+\overline{\mathrm{dtak}}\ f(3)+$
$\quad \mathrm{dtak}\ f(3)+b_{16}\ f(4)\overline{+\mathrm{dtak}}\ f(5)+\mathrm{dtak}\ f(5)$

$\mathrm{as} = f(2)+\overline{\mathrm{dtak}}\ f(3)+\mathrm{dtak}\ f(3)+\overline{\mathrm{dtak}}\ f(5)+\mathrm{dtak}\ f(5)$

$\mathrm{lds} = f(2)+\overline{\mathrm{dtak}}\ f(3)+\mathrm{dtak}\ (f3)+\overline{\mathrm{dtak}}\ f(5)+\mathrm{dtak}\ f(5)$

$\mathrm{p0} = \overline{\mathrm{dtak}}\ f(5)+\mathrm{dtak}\ f(5)$

Irreducible minterms in the output equations

A simple algorithm enables them to be found :
$f(6)$; $b_{16}f(4)$; $\overline{\mathrm{bg}}\ f(1)$; $\mathrm{tr}\ f(0)$; $\mathrm{bg}\ f(1)$; $f(5)$
In this case, we prevent the application of rule b to the pair of states (1,2), leading to the elimination of one of the product terms : bgf(1) or $\overline{\mathrm{bg}}$f(1) which are prime implicants of the output equations.
The treatment of the both systems (next state equations, output equations) leads to the following list of constraints :
(C(0) c C(1)) ; (C(3) c C(4)) ; (C(0) c C(5)) ;
(C(3) adj C(5)) ; (C(2) adj C(3)) ;
((C(3) c C(4)) and (C(5) c C(6))) OR
((C(4) c C(3)) and (C(6) c C(5))) ;
C(3) XOR C(4) = C(5) XOR C(6).

This leads in this case to the same encoding as previously, but the next state equations are different (for Y_3)

Results

Next state equations

$Y_1 = \lfloor f(2)+f(3)\rfloor + \overline{b_{16}}\ f(4) + f(5)$

$Y_2 = \mathrm{bg}\ f(1) + \mathrm{dtak}\lfloor f(3) + f(5)\rfloor$

$Y_3 = \mathrm{tr}\ f(0) + \mathrm{bg}\ f(1) + \overline{\mathrm{bg}}\ f(1) + \lfloor f(2)+f(3)\rfloor$

Output equations

$\mathrm{fcm} = b_{16}\ f(4) + f(6)$

$\mathrm{br} = \mathrm{tr}\ f(0)+\overline{\mathrm{bg}}\ f(1)$

$\mathrm{vad} = \mathrm{bg}.f(1)$

$\mathrm{uds} = \lfloor f(2)+f(3)\rfloor$

$\mathrm{bgak} = \lfloor f(2)+f(3)\rfloor+f(6)+b_{16}\ f(4)+\overline{b_{16}}\ f(4)$
$\quad +\mathrm{bg}\ f(1)+f(5)$

$\mathrm{as} = \lfloor f(2)+f(3)\rfloor+f(5)$

$\mathrm{lds} = \lfloor f(2)+f(3)\rfloor+f(5)$

$\mathrm{p0} = f(5)$

Set of product terms

The encoding leads to 9 product terms :

$\lfloor f(2)+f(3)\rfloor$	ie $(\overline{Y}_2Y_3)$;
$\overline{b}_{16}\ f(4)$	ie $(\overline{b_{16}}\ Y_1Y_2Y_3)$;
$\mathrm{bg}.f(1)$	ie $(\mathrm{bg}.\overline{Y}_1Y_2Y_3)$;
$\mathrm{dtak}\lfloor f(3)+f(5)\rfloor$	ie $(\mathrm{dtak}.Y_1\overline{Y}_2)$;
$\mathrm{tr}\ f(0)$	ie $(\mathrm{tr}.\overline{Y}_1\overline{Y}_2\overline{Y}_3)$;
$\overline{\mathrm{bg}}\ f(1)$	ie $(\overline{\mathrm{bg}}\ \overline{Y}_1Y_2Y_3)$;
$b_{16}\ f(4)$	ie $(b_{16}Y_1Y_2Y_3)$;
$f(6)$	ie $(Y_1Y_2\overline{Y}_3)$;
$f(5)$	ie $(Y_1\overline{Y}_2\overline{Y}_3)$

Performance

The program is developed on Vax 11/780. The example presented here requires 22 sec CPU, 25 sec when the output equations are taken into account.

Area gain

The circuit is implemented on a PLA, the area of which is proportional on one hand to the number of product terms, on the other hand to the sum : ni+3n+nO, where ni is the number of controller inputs, n is the number of encoding bits, nO is the number of outputs.

The state assignment enables to obtain 9 product terms (instead of 11 in the worst case) and 5 inputs (instead of 8).

We shall choose $g = \frac{S'-S}{S'}$ as an estimate of the gain

S' = area in the worst case

S = area corresponding to the state assignment

g = 28 %

Other examples have been treated and the characteristics of the circuits, the state assignment and the gain estimate are presented in table 5. The circuits labelled 1,2,3,4 are presented in the references [12], [17], [16], [18]

				worst case		obtained results		
Circuit	ns	nO	n	NPT	ni	NPT	ni	Area gain
1	20	8	5	38	18	27	13	37%
2	4	4	2	12	4'	°	3	38%
3	7	3	3	14	2	11	2	21%
4	11	0	4	17	6	9	4	53%

Table 5

In table 5, ns is the number of states, NPT the number of product terms, n the number of bits, ni the number of input conditions taken into account
in the PLA, nO the number of outputs.

CONCLUSION

We have presented here a rule based system for optimal encoding of the states of a controller, taking into account the next state and output equation minimization. The rules imply constraints on the encoding such as adjacences, inclusions...The set of rules may be modified to take into account the target structure (types of flip-flops, type of combinatorial circuits).

The example presented here addresses only a small size circuit but it demonstrates the principle of the method. The system works fairly well for such circuits, it is being improved for larger circuits, in order to reduce the CPU time.

The system is very flexible, because of its organization with an independent rule base, which may be extended by addition of new rules.

The encoding sytem is included in a complete design system, allowing a top down design, from the specification to the layout.

ACKNOWLEDGMENTS.

The authors thank C. Bellon for her helpfull cooperation.

BIBLIOGRAPHY.

(1) J.HARTMANIS, R.E.STEARNS,
Algebraic Structure Theory of Sequential Machines,
Prentice Hall, 1966.

(2) J.HARTMANIS,
"On the State Assigment Problem for Sequential Machines"
IRE Trans. Elect. Comp., Vol. EC-10, pp.157-165, June 1961.

(3) D.B.ARMSTRONG,
"A Programmed Algorithm for Assigning Internal Codes to Sequential Machines",
IRE Trans. Comp., Vol. EC-11, pp.466-472, August 1962.

(4) R.KARP,
"Some Techniques for State Assignment for Synchronous Sequential Machines",
IEEE Trans. Elect. Comp., Vol.EC-13, pp.507-518, Oct.1964.

(5) T.A.DOLOTTA, E.G.MC CLUSKEY,
"The Coding of Internal States of Sequential Machines",
IEEE Trans. Elect. Comp., Vol EC-13, pp.549-562.

(6) G.SAUCIER,
"Encoding of asynchronous Sequential Machines",
IEEE Trans. on E.C., Vol.E.C.16, n°3, pp.365-369, 1967.

(7) G.SAUCIER,
"State assigment of asynchronous Sequential Machines using Graph Techniques,
IEEE Trans. on Comp., mars 1972.

(8) G.SAUCIER,
"Next state equations of asynchronous sequential machines", IEEE Trans. on Comp., mars 1972.

(9) G.De MICHELI, A.SANGIOVANNI-VINCENTELLI, R.BRAYTON,
"KISS : a Program for Optimal State Assignment of Finite State Machines",
Int. Conf. on Comp.Aid. Design, Santa Clara, Nov.1984.

(10) G.De MICHELI,
"Optimal Encoding of Control Logic",
IEEE ICCD 1984, pp.16-22.

(11) G.De MICHELI, A.SANGIOVANNI-VINCENTELLI, T.VILLA,
"Computer Aided Synthesis of PLA-Based Finite State Machines",
ICCAD, Santa Clara, pp.154-157, Sep.1983.

(12) E.FLAMAND,
"A Complete and Automatic System for Sequencer Design",
IEEE ICCD 1984, pp.324-340.

(13) S.HANRIAT, J.IDT,
"Compilateur de Fonctions Booléennes sur Réseaux Prédiffusés",
Colloque National Conception de Circuits à la Demande,
pp. 493-518, Grenoble, 1985.

(14) P.AMBLARD, M.CRASTES de PAULET, J.RARIVOMANANA, G.SAUCIER,
"CADOC : a functional Specification and Simulation Tool for VLSI",IEEE EDA 84, Warwick, march 1984, pp.147-151.

(15) C.BELLON, M.CRÁSTES de PAULET, S.HANRIAT, J.RARIVOMANANA, G.SAUCIER,
"CADOC System : a Tool for Multilevel Description and Test Generation for VLSI Circuits",
7th Int. Conf. CHDL 85, Tokyo, aug. 1985.

(16) G. De MICHELI,
"Optimal state assignment for finite machines",
Research Report, 1985.

(17) C. MEAD, L.CONWAY,
"Introduction to VLSI systems",
Addison Wesley, 1981.

(18) G. THUAU,
"Conception logique et topologique en technologie MOS",
Thèse de 3ème cycle, Grenoble, 1983.

Constructive Solid Geometry: A Symbolic Computation Approach

L. Leff
David Y. Y. Yun

Department of Computer Science and Engineering
Southern Methodist University

ABSTRACT

A method of parametcrizing an object that is represented by constructive solid geometry (CSG) is provided. A method is developed for generating the constraint equations on the parameters which provide a sufficient condition so that the object remains geometrically similar as the parameters are varied. These constraints and the symbolic form of CSG are important to the problem of geometric optimization which is part of Computer Aided Engineering.

1. Introduction

Constructive Solid Geometry (CSG) is now the method of choice for representing mechanical engineering objects such as machine parts inside a computer and is replacing other methods such as boundary representations in the CAD-CAM systems of the 80's. [5]

In CSG, an object is represented as the boolean operations, union, difference and intersection, as applied to regular sets such as the set of points enclosed by rectangles and circle in two dimensions or rectangular pyramid and right circular cylinder in three dimensions.

Tilove [5] has shown how an object represented in Constructive Solid Geometry can be converted into its boundary representation. This is important for displaying the object and determining the equivalence of two objects among other purposes.

A method was developed to represent an object that has dimensions whose values are symbolic expressions based on the boundary representation. [1] Each point where two segments of the boundary representation intersect has symbolic values for its coordinates. The connectivity of these points by means of faces and edges are also stored. A method for representing part families and for setting up CSG objects whose dimensions are variables is discussed in the spirit of tolerancing in [3]. In constructive solid geometry, as the dimensions of the primitives vary, the object changes shape. In a boundary representation, the faces may intersect each other as the locations of the points of intersection vary when the parameters change. For CSG objects, we provide a method for generating a set of equalities and inequalities, such that their satisfaction provides a sufficient condition for the object to be similar to the original one as the parameters are varied. Two objects constructed by parametric CSG are defined to be similar when their representation by means of the techniques of [1] are equal. A more formal definition follows later in the paper.

This set of inequalities can be put into a geometric optimization program. Geometric optimization programs are used to determine the structure of minimum weight or cost. The constraints include those relating to the partial differential equations, e. g. the stress in the object must not exceed a given value, the geometric constraints insuring that varying the parameters do not change the form of the symbolic solution, as well as additional ones entered by the user.

In this paper, we illustrate the techniques and concepts with the two dimensional case where the primitive used is a rectangle whose faces are parallel to the x and y axis. For a discussion of arbitrary curved surfaces, canonical forms and other aspects of geometric optimization, see [2].

2. Constructive Solid Geometry

In constructive solid geometry, we represent an object and the union, difference or intersection of simple objects or of other objects so defined. The mechanical engineer doing constructive solid geometry can enter the location of two simple objects. He can then indicate that they should be combined into a new object by taking the intersection, union or difference of the sets of points represented by the objects.

This process can continue recursively, since the intersection, union and difference operations can be applied to objects previously created and to additional simple objects. These operations can be expressed by an expression where the operands are primitives and the operators are union, difference or intersection. We can define the syntax of this expression as follows:

E -> (E O E) | P

where E represents an expression, O an operator, and P a primitive.

These expressions can be represented in the computer by the equivalent binary tree with the operators at the nodes and the primitives as leaves. Each primitive will be represented by a record with four real numbers, the x and y position of the lower left hand corner and the height and width. The operators will simply be tree elements with left and right sons and an indicator for the type of operator (union, intersection or deletion).

Figure 1 shows four different CSG trees corresponding to the same physical object.

3. CSG Tree to boundary conversion algorithm

A frequent operation in mechanical engineering CAD-CAM is to convert the constructive solid geometry to a boundary representation. This is needed for drawing the object on a graphics device. In addition, it is possible to tell whether two objects (e. g. A and B) represent the same by taking the symmetric difference of A and B, (A–B)∪(B–A). If the result of the symmetric difference has no boundaries, then A represents the same object as B.

We will use this later as the main part of our canonical simplification function for CSG trees and in modified form as part of the geometric constraint generator.

The algorithm [4] takes each edge of each primitive and propagates it down the CSG tree. When the edge reaches a primitive it is divided into those segments on, off or in the primitive. Then

CH2345-7/86/0000/0925$01.00 © 1986 IEEE

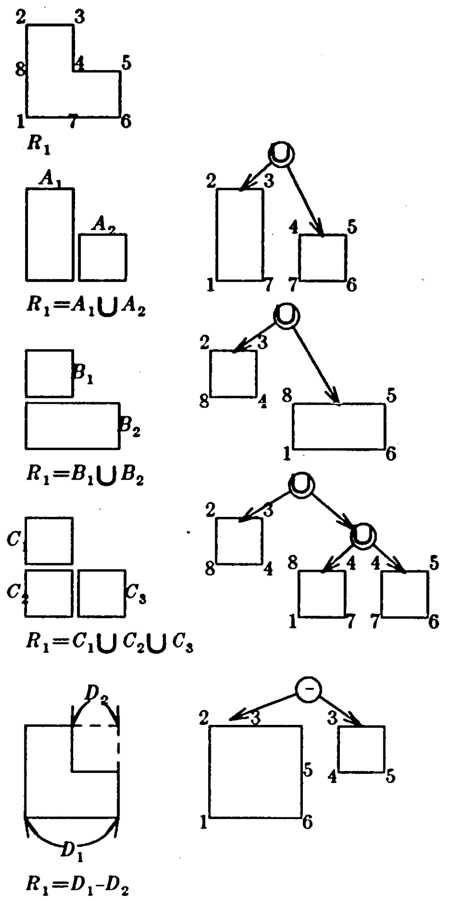


Figure 1

at each node, the results of dividing the segment for each subtree are merged together to produce the classification of the edge for the object represented by the subtree at that node.

We do this with the following algorithm:

```
function edgelist (T) ; T is a CSG tree (returns list of edges
on the boundary of the object
represented)
        edgelist <- nil
        for each primitive P_i
                for each edge E of P_i
                        (EinS, EonS, EoutS) <- M [E,T]
                        merge (EinS,Eon,EoutS) with edgelist
                end /for/
        end /for/
remove edges in or out of object leaving edges on
end /procedure/
```

Algorithm to Classify an edge against a primitive

```
function M(Edge, T) ; T is a CSG Tree (returns edge divided into
segments with on, off or in indicated plus for on whether shape is
above or below)
if T is a primitive then
        perform a primitive classification
else
        classify the Edge with respect to the left hand and right
        hand sons of T by calling M recursively
        for each part of the edge after it is divided at any point
        where it intersects any primitive
                determine its classification using on, off or in informa-
                tion and neighborhoods as passed up from calling M
                on left and right hand subtrees
end /function/
```

This algorithm works by taking the edge of each primitive and determining whether or not it is on the object's perimeter. The union of these edges will be the boundary of the object.

The working of the algorithm is illustrated with the object, $A_1 \cup (A_2 \cup A_3)$, which is illustrated by figure 2. Figures 3,4 and 5 show the line segments generated when each of the four edges of A_1, A_2, and A_3 are respectively in the outer loop of the function edge list.

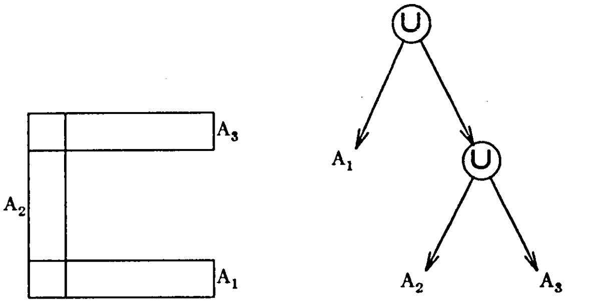


Figure 2

When we take the edges of A_3 and find out which are in the object, we get:

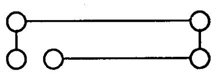


Figure 3

For A_2, we get

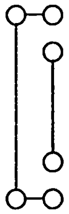


Figure 4

For A_1, we get

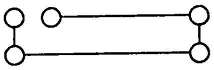


Figure 5

The union of these shapes form the perimeter of the total object.

The algorithm M [5] classifies the edges. Each edge is classified against both the left and right hand subtrees of any node. The result coming back up will indicate what parts of the edge are on, in or off the object represented by the subtree. These can be combined by a complicated rule system which depends on the classification of the edge of each subtree, the type of operator and in the case of on segments which side the object the segment is on. Figure 6 shows how the edge 1-2 is classified in A in the trees A∪B, A∩B and A-B

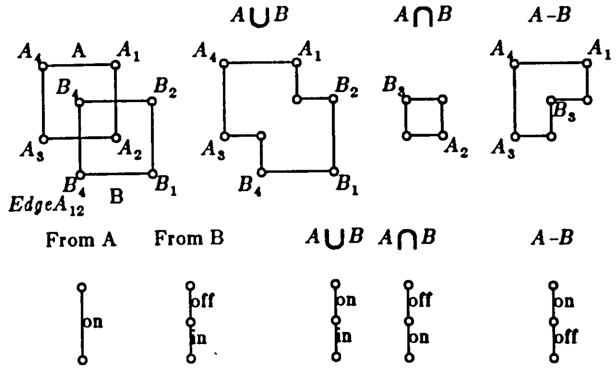


Figure 6

4. Symbolic Constructive Solid Geometry

In symbolic constructive solid geometry, each rectangle has the x and y position, height and depth represented both as an arbitrary symbolic mathematical expression and as a constant. The constraints generated will insure that any value substituted for the parameters into the symbolic expressions will yield an object geometricly similar to the one given when the primitives have the dimensions given by the constants. The constant dimensions can also be used in displaying the object or as starting values for the iterations in geometric optimization.

In addition to the expression defining the CSG primitive, the user inputs a table listing for each primitive both the symbolic and constant values for the x and y coordinates of the lower left hand corner as well as Δx and Δy, the width and height respectively. The user can also provide the symbolic location of a primitive by defining a vector whose components are expressions between a corner of another primitive and a corner of the new primitive. The graph formed by these connections should form a forest where the nodes are the primitives and an edge exists between two nodes when there is a connection between the corresponding primitives.

(In the discussion that follows values that are symbolic will have a .s prefix while the values that are constants will not.)

Figure 7 illustrates a T-bar and the tree corresponding to the CSG expression entered with parametric CSG. Tables 1 and 2 show the other information.

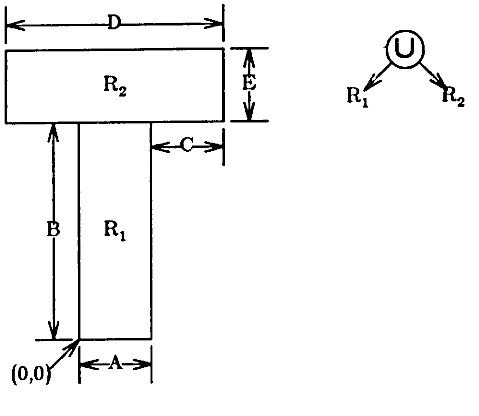


Figure 7

	x	y	Δx	Δy	x.s	y.s	Δx.s	Δy.s
R_1	0	0	1	3	0	0	A	B
R_2	-1	3	3	1	-	-	D	E

Table 1: Rectangle Table for Figure 7

RECT1	RECT2	CORNER1	CORNER2	Δx.s	Δy.s
1	2	NE	SE	C	0

Table 2: Connection Table for Figure 7

Figure 8 illustrates a more complicated representation, a square with a hole in it formed by four rectangles. Notice the different methods of connecting juxtaposed rectangles. Rectangles 1 and 2 are connected with a connection list item. The system determines that rectangles 2 and 3 are juxtaposed from the values of the constants, i. e. that G = 0 and H = B + E. The connections between rectangles 3 and 4 are determined in the same manner. Of course, the end of rectangle 4 is inside rectangle 1.

	x	y	Δx	Δy	x.s	y.s	Δx.s	Δy.s
R_1	0	0	5	1	0	0	A	B
R_2	0	1	1	3	-	-	F	E
R_3	0	4	5	1	G	H	J	I
R_4	4	0.5	1	3.5	K	L	M	N

Table 3: Rectangle Table for Figure 8

RECT1	RECT2	CORNER1	CORNER2	Δx.s	Δy.s
1	2	nw	sw	0	0

Table 4: Connection Table for Figure 8

5. Algorithm for Generating Geometric Constraints

We define similarity as mentioned in section 1. Let E2 be the object produced from the CSG tree for E1 by simply changing the values of the parameters in the symbolic CSG tree defining E1. Let G1 and G2 be graphs formed from E1 and E2 respectively. The nodes correspond to the positions in two dimensional space where a horizontal and a vertical boundary segment intersect. There will exist an edge between node i and node j of the graph if there is a primitive that has a corner at the point corresponding to node i and one node at the point corresponding to node j. If the new set of values for the parameters obeys the constraints, then G1 and G2 will be isomorphic and for all pairs of nodes mapped by the isomorphism, the two nodes will have the same expressions for x.s and y.s.

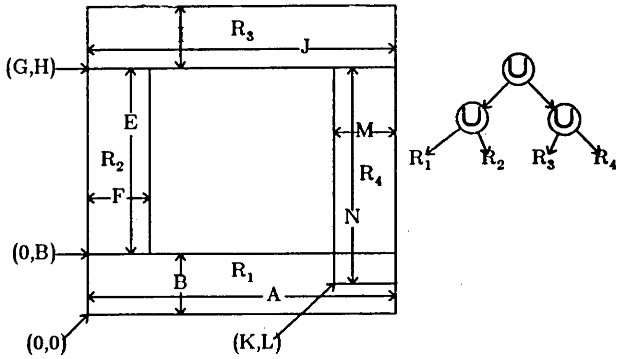


Figure 8a

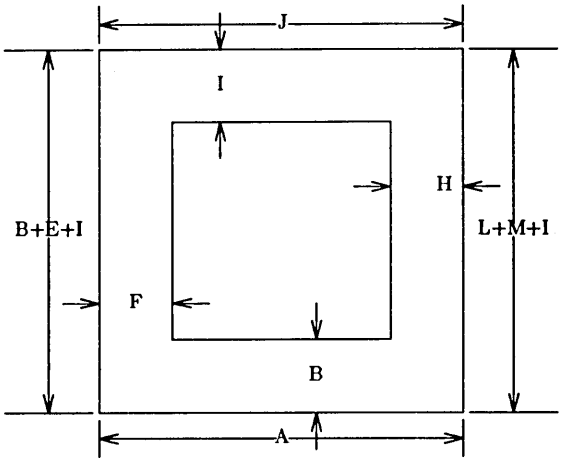


Figure 8b

This algorithm uses as its principal component an adaptation of the Tilove algorithm for finding the boundary segments. In addition to generating the boundary segments of the object, it generates the constraints necessary to insure that those are the boundary segments. There are five steps.

1. For each expression that appears in the $\Delta x.s$ or the $\Delta y.s$ field of the rectangle term, add the inequality that the expression is > 0 to the inequality list.
2. Perform a depth-first search of the graph formed by the items on the connection list. Compute the symbolic values for the x and y coordinates of the four corners of each rectangle: x.sw.s,y.sw.s,x.se.s,y.se.s,x.nw.s,y.nw.s,x.ne.s, and y.ne.s.
3. Run a modified version of the Tilove algorithm for finding the boundary segments of an object constructed with CSG. It generates the boundary segments as before. However, the end points of each segment are labeled with the symbolic values for the coordinates as well as the numeric values. In the classification of a segment against a primitive and in the M routine for two lists of the same segment, we generate equalities and inequalities needed to provide a sufficient condition that any classification done would remain the same as the parameters are varied. (See below for details.)
4. Convert the equality list into a set of substitutions. In the general case, this can be done with a Groebner basis algorithm. If all the dimensions and locations were simply single variables (as is often the case in practice), then this can be done by reducing the coefficient matrix to a reduced row echelon form. Apply the set of substitutions to the inequalities.
5. Remove any redundant inequalities and make obvious simplifications by combining terms such as replacing $B < B + D$ with $0 < D$.

The resulting list of inequalities is the set of geometrical constraints.

The following is a detailed description of the part of the primitive classification routine that generates the inequalities and equalities. Only the part for classifying a horizontal segment is shown. The part for classifying vertical segments is analogous with the roles of x and y reversed. The notation AddI (blah) means add the inequality blah to the list of inequalities, AddE (blah1) means add the equality blah1 to the list of equalities. The shorthand GI (blah2) means: if list = nil then list = blah2 else list = list or blah2. The routine receives as input a rectangle and a horizontal edge where the coordinates left and right end points are referred to be the subscripts 1 and 2 respectively and whose Y value is referred to as Y_0.

```
List = nil
If $Y_0$>Y.nw or $Y_0$<Y.sw or $X_0$>=X.se or $X_1$≤X.sw then
        If $Y_0$>Y.nw then
                GI (Y.nw.s>$Y_0$.s)
        If $Y_0$ < Y.sw then
                GI ( $Y_0$.s<Y.sw.s)
        If $X_0$≥X.se then
                GI ( $X_0$.s≥X.se.s)
        If $X_1$<=X.sw then
                GI ( X1.s <= X.sw.s )
If list not = nil then AddI(list)
else
        If $X_0$<X.sw then
                AddI ( $X_0$.s<X.sw.s )
        If $X_0$>X.sw then
                AddI ($X_0$.s>X.sw.s)
        If $X_0$=X.sw then
                AddE( $X_0$.s=X.sw.s)
        If $X_1$>X.se then
                AddI(X.se.s<$X_1$.s)
        If $X_1$<X.se then
                AddI (X.se.s>$X_1$.s )
        If $X_1$=X.se then
                AddE(X.se.s=$X_1$.s)
        If $Y_0$>Y.sw and $Y_0$<Y.nw then
                AddI($Y_0$.s>Y.sw.s)
                AddI($Y_0$.s<Y.nw.s)
        If ($Y_0$=Y.sw) then
                AddE($Y_0$.s=Y.sw.s)
        If ($Y_0$=Y.nw) then
                AddE($Y_0$.s=Y.nw.s)
end /if/
```

When the M routine is invoked at a given operator node, it calls itself recursively for the left and right hand subtrees. After control is returned after these two calls, the routine has two chains of horizontal segments to be classified against each other. From these two chains, we generate additional equalities and inequalities that must be satisfied.

Let A_1–A_n and A_1.s–A_n.s represent the list of x coordinates of the segments returned from the call from the left hand subtree and their symbolic values respectively arranged in ascending order by numerical x value. B_1–B_m and B_1.s–B_m.s represent the x coordinates for the right hand subtree.

We generate inequalities and equalities as follows:

```
For all i
        If i <  n then if $\exists_j$ ($B_j$>$A_i$ $\wedge$ (j=1 $\vee$ $B_{j-1}$<$A_i$ )) $\wedge$
        $\exists_k$ ($B_k$<$A_{i+1}$ $\wedge$ (k=m $\vee$ $B_{k+1}$>$A_{i+1}$)) then
                AddI( $A_i$.s<$B_j$.s )
                AddI( $A_{i+1}$.s>$B_k$.s )
```

```
For all i
    if $i < m$ then if $\exists_j (A_j>B_i \wedge (j=1 \vee A_{j-1}<Bi)) \wedge$
    $\exists_k (A_k<B_{i+1} \wedge (k=n \vee A_{k+1}>B_{i+1}))$then
            AddI( $B_i.s<A_j.s$ )
            AddI( $B_{i+1}.s>A_k.s$ )
if $B_i<A_i \wedge (i=m \vee B_{i+1}>A_i)$ then
        AddI( $B_i.s<A_i.s$ )
if $A_i<B_i \wedge (i=n \vee A_{i+1}>A_i)$then
        AddI( $A_i.s<B_i.s$ )
if $B_i>A_n \wedge (i=m \vee B_{i-1}<A_n)$ then
        AddI( $B_i.s>A_n.s$ )
if $A_i>B_m \wedge (i=n \vee A_{i-1}<B_m)$ then
        AddI ( $A_i.s>B_m.s$ )
for each i and j such that $A_i=B_j$ AddI ( $A_i.s=B_j.s$)
```

Note that when we pass up the list of segments we insure that only one of the two symbolic expressions, $A_i.s$ and $B_j.s$ that are equal is used as a coordinate value for an endpoint.

To illustrate these points, I will show how the inequalities are generated for the parametric CSG objects in Figures 7 and 8. In step 1, we generate the constraints: $A > 0$, $B > 0$, $D > 0$ and $E > 0$. For the T-shaped object, step 1 computes that the southeast corner is at (A+C,B). x.sw.s and y.sw.s are A+C-D and B respectively. All inequalities are generated by the primitive classification of the lower edge of rectangle 2 against rectangle 1. We generate the following line and inequalities:

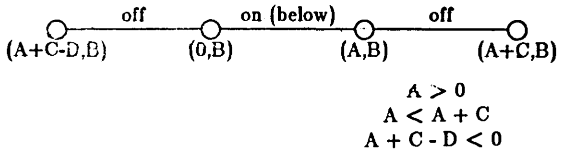


Figure 9

In step 5, the second inequality is simplified and this set is combined with the set generated in step 1. The resulting set of inequalities is:

$A > 0$ $C > 0$ $B > 0$
$D > 0$ $E > 0$ $A + C - D < 0$

When we apply the algorithm to the object in figure 2, step 1 generates the following inequalities: $A > 0$, $B > 0$, $F > 0$, $J > 0$, $I > 0$ and $M > 0$. Step 2 assigns the x.s and y.s values for R_2 as 0 and B respectively. In step 3, we illustrate the algorithm as it works on the upper edge of R_1, the lower edge of R_3, and the lower edge of R_4. These edges happen to be sufficient to generate all the applicable inequalities. In Figure 11, we see the steps in classifying the top edge of R_1. Steps 1,2, 3 and 4 represent the classification of the edge against the primitives R_1, R_2, R_3 and R_4 respectively. In step 5 we combine the results from R_1 and R_2 and in step 6, we combine the results from R_3 and R_4. Finally step 7 is the final combination of the edges. In figure 12, we see the same steps for the lower edge of R_3.

Lastly we illustrate the processing of the lower edge of R_4. We only show the first four steps since no inequalities or equalities are added in the later steps.

Going into step 4, we have the inequalities:

$H > B$	$F<G+J$	$K<G+J$
$K>G$	$F<A$	$H>B$
$L<B$	$K<A$	$F<K$
$A>0$	$B>0$	$F>0$
$J>0$	$I>0$	$M>0$
$(K>F)$or$(L<B)$	$G<F$	

and the equalities:

$B+E=H$ $G=0$ $L+N=H$
$G+J=K+M$ $K+M=A$

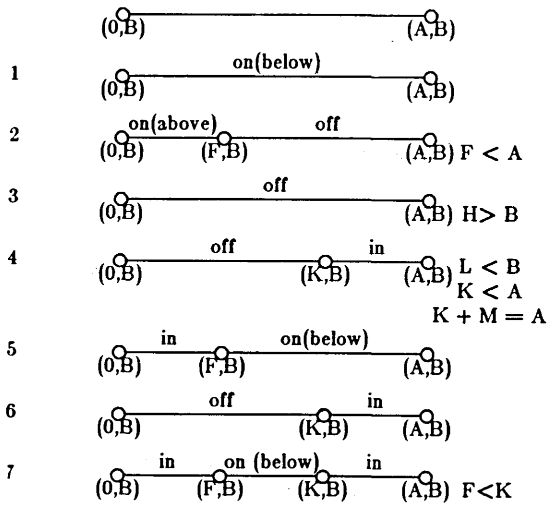


Figure 10

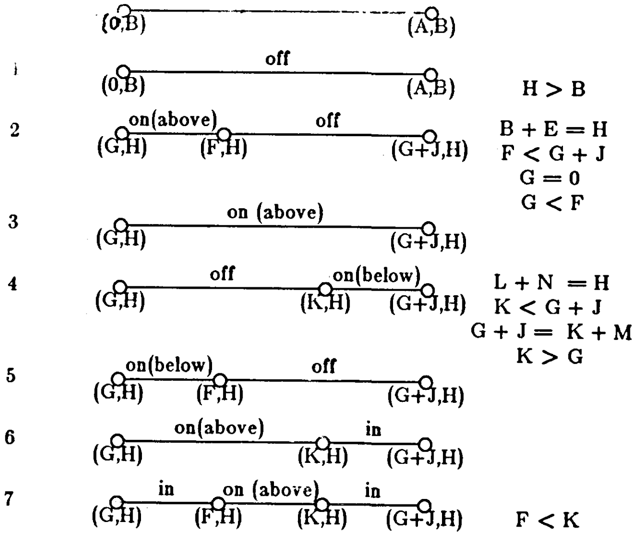


Figure 11

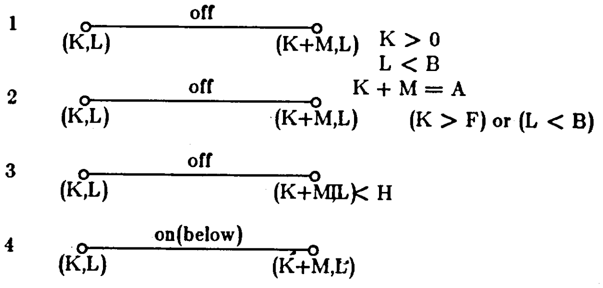


Figure 12

In step 4, the equalities are converted into a substitution list as follows:

B -> L+N-E
G -> 0
H -> L+N
J -> K+M
A -> K+M

In step 5, we apply the substitutions generating and eliminate redundancies getting as our final result for the constraints:

$K+M > 0$	$F < J$	$K < J$
$K > 0$	$F < K+M$	$H > B$
$L < B$	$F < K$	$L + N - E > 0$
$F > 0$	$I > 0$	$M > 0$

6. Computational Complexity Issues

6.1. Generating Geometric Constraints

Let N be the number of primitives. Then the number of edges in the boundary, n, must be $O(N^2)$ [5]. Also the number of points on the boundary where two segments intersect is O(n). k is the number of different types of terms in the expressions for distances. For example, if we had the expressions for distances as n^2, n, $3*n$, $2*n$ then we would have $k = 4$. We count variables raised to different powers separately but not terms with different constant factors.

In the conventional algorithm for finding the boundary segments, the number of times that a pair of line segments must be matched is $O(N^3)$. In our modified algorithm, each time we match a pair of line segments we generate O(1) new inequalities. Since this generation does not involve manipulating the expressions, this modification does not change the $O(N^3)$ behavior. However, the total number of equalities and inequalities is $O(N^2)$. For each edge of a primitive, we have up to four equalities or inequalities generated for each of the O(N) primitives. Since there are O(N) edges, there are $O(N^2)$ inequalities.

Step 1 is trivially O(N). Step 2 to go through the connection lists corresponds to depth first search on a graph. Since there is a maximum of one connection item per primitive which might involve adding polynomials where the result might become as large as O(k), this step is O(kN).

In this subsection the times given will be for the important practical case of the dimensions being simple parameters. Equality constraints are only generated when two rectangles have edges that end at the same place. Thus there can only be O(N) such constraints. In the special case where each dimension is a single variable, the number of variables, k, = O(N) so the time to convert the coefficient matrix to row echelon form is $O(N^3)$.

We perform up to N substitutions on each inequality, each of which can involve adding two k term polynomials so the total time for this step is $O(N^3k)$.

Each simplification takes up to O(k) time per inequality. Eliminating redundant inequalities, if we do this by using an O(z log z) sort first, will take $O(N^2 \log Nk+N^2 k \log k)$. The k log k term comes from the time to put the expressions composing each inequality in lexicographic order so they can be compared in O(k) time. However $k = O(N)$ so the total time for this step is $O(N^3k)$.

The actual evaluation of the inequalities which will occur once per iteration in geometric optimization will be $O(N^2k)$ operations since we have $O(N^2)$ inequalities with up to k terms apiece.

References

1. C. Eastman, J. Lividini and D. Stoker, A Database for Designing Large Physical Systems, *National Computer Conference 44*, (1975), 603-611.

2. L. L. Leff and D. Y. Y. Yun, Constructive Solid Geometry: A Symbolic Computation Approach, 86-CSE-14, Southern Methodist University, Department of Computer Science, 1986.

3. A. A. G. Requicha, Representation of Tolerances in Solid Modeling: Issues and Alternative Approaches, in *Solid Modeling by Computers*, M. S. Pickett and J. W. Boyse (ed.), Plenum Publishing Corporation, 1984, 3-22.

4. R. B. Tilove, Set Membership Classification: A Unified Approach to Geometric Intersection Problems, *IEEE Transactions on Computers C-29*, 10 (Oct. 1980), 874-883.

5. R. B. Tilove, Exploiting Spatial and Structural Locality in Geometric Modelling, TM-38, Production Automation Project, University of Rochester, Rochester, New York, Oct. 1981.

CREATION AND SMOOTH-SHADING OF STEINER PATCH TESSELLATIONS

David E. Breen

Center for Interactive Computer Graphics
Rensselaer Polytechnic Institute
Troy, NY 12181

Abstract

Motivated by recent work with Steiner patches, a method has been developed to create C0 tessellations of Steiner patches which approximate high order biparametric surfaces. An accompanying technique has also been created to smooth-shade the tessellation in order to produce high quality color raster images. The whole process is an extension of the tessellation and rendering process using polygons. The Steiner patch method provides a better approximation with fewer geometric elements and non-jagged silhouettes.

The tessellation algorithm utilizes the properties of the Bernstein-Bezier representation of Steiner patches. The algorithm fits Steiner patches to ordered surface point and normal data. The resulting tessellation is C0 everywhere and C1 at the data points. The unique normal vectors at the data points are interpolated over the tessellation during rendering in order to produce a smooth looking surface.

Introduction

The tessellation of complex curved surfaces into polygonal meshes is a common and successful method used to facilitate rendering. The method transforms a mathematically complex geometric description (e.g. bicubic patches) into a far simpler geometric approximation - a polygonal mesh[1]. This transformation greatly simplifies the mathematics and algorithms of rendering. With the simplification also comes significant increases in rendering speeds. Polygons are commonly used because of their mathematical simplicity. Many algorithms have been developed to create smooth shaded images of polygonal meshes[2, 3, 4]. These methods utilize some form of interpolation in order to blend the faceted surfaces of polygonal meshes.

The work presented here is based on the ideas previously mentioned. Complex surfaces are tessellated into a network of simpler components in order to simplify the mathematics and shorten rendering times. Once a tessellated mesh is created, a shading algorithm is applied to the mesh in order to produce a smooth shaded image of a tessellated surface. The simpler components being used are Steiner patches[5]. The Steiner patch is the simplest biparametric patch. Therefore this application of it is a logical extension of current rendering techniques using polygons.

This work has been motivated by the results and mathematics produced by Sederberg and Anderson[6]. They have presented a method for ray tracing Steiner patches. Their results are based on a parametric surface implicitization method developed by Sederberg[7].

The implicitization and ray tracing algorithm was studied and implemented by the Computer Animation Group at RPI's Center for Interactive Computer Graphics. The software to ray trace Steiner patches was incorporated into the group's ray tracer, BART. One of the main goals of the group is to produce high quality ray traced computer animation. Because this task is so computationally expensive and time consuming, the group investigated ways to save time and CPU cycles[8]. Creating meshes of approximating Steiner patches which are subsequently ray traced offers that possibility.

Geometry

Before continuing, it is necessary to briefly describe the geometric entities which are discussed and utilized in this paper. The two most referenced are Steiner patches and superquadrics. The Steiner patch is the essential geometric element in the tessellation algorithm. The algorithm has been applied to a certain type of biparametric surface - the superquadric.

Steiner Patches

The Steiner patch receives its name from the German mathematician Jakob Steiner, who studied it extensively in 1844 during a stay in Rome[9]. The Steiner patch is easily visualized as a triangular quadratic Bernstein-Bezier patch. In this form, the patch can be represented by six control points and a triangular parameter space (domain)[10]. See Figure 1. The six points form a triangular control net which contains and controls the patch. Three of the six points are on the patch surface at the corners, P00, P20, P02. The patch is bounded by three quadratic curves P00-P20, P00-P02, P02-P20. The most important feature, at least to this project, is that each corner control point and its two neighboring interior control points define the tangent plane at the corner control point. This feature of the Steiner

CH2345-7/86/0000/0931$01.00 © 1986 IEEE

patch is central to the approximation algorithm.

A Steiner patch represented in the Bernstein-Bezier basis is written as:

$$\vec{B}_2(u,v) = \sum_{\substack{i+j <= 2 \\ i,j >= 0}} \vec{P}_{i,j} \left(\frac{2!}{i!\, j!\, (2-i-j)!} \right) u^i v^j (1-u-v)^{2-i-j} \qquad (1)$$

$$u >= 0, \quad v >= 0, \quad u+v <= 1$$

Note the bounds of the parameters U,V. The edges of the patch are defined by the U=0, V=0, U+V=1 curves and are parabolas.

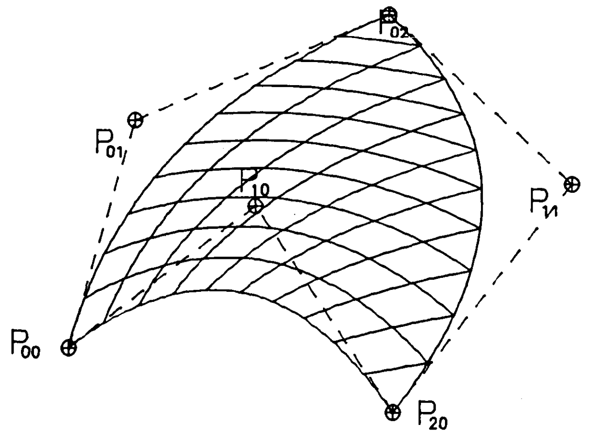


Figure 1. Steiner Patch with Control Points

Superquadrics

Superquadrics are a generalization of the quadric surfaces. The mathematics of this two-parameter family of geometric primitives was formalized by Barr[11]. He defines the parametric form of superellipsoids, superhyperboloids of one and two sheets and supertoroids. An implicit inside-outside function is also provided for each of them. Specifically our work applied the tessellation and smooth shading algorithm to superellipsoids. Exponentiating the sine and cosine terms in the parametric form of the ellipsoid gives a variety of curved, beveled and pinched shapes which is the superellipsoid.

Tessellation and Smooth-Shading Algorithm

In order to create a smooth-shaded rendering of a Steiner patch mesh, three steps are required.

1. Create a CO continuous Steiner patch tessellation which approximates the original surface. Create it in such a way to guarantee tangent plane continuity at common patch vertices.

2. Ray-trace, calculating ray intersections and determining the u,v values of the intersection.

3. Instead of calculating the actual normal to the surface at the intersection, calculate the interior normals as a linear combination of the normals at the vertices.

The tessellation algorithm can be summarized as the following.

1. Triangulate the parameter space of the original surface.

2. Calculate the points and tangent planes associated with each u,v value at the triangle vertices.

3. Using these points and tangent planes, an interpolating Steiner patch is fit to each triangular segment.

Subdivision of Parameter Space

The first step in the algorithm is to subdivide the parameter space of the patch which is being approximated by a network of Steiner patches. More accurately stated, the parameter space should be triangulated, because the Steiner patch is defined on a triangular domain. There are basically two ways to subdivide the parameter space, recursively and iteratively. The recursive subdivision begins by breaking the parameter space into two or four triangles. At each level of subdivision, each triangle is split into four more triangles. The number of triangles grows as 4 raised to the nth power, where n is the number of subdivisions. The iterative subdivision splits the original parameter space into n^2 squares. Each square is then split into two or four triangles. The same iterative and recursive techniques can be applied to a triangular parameter space.

Subdivision of a spherical parameter space, like the superquadric's, can be a bit trickier. It can be split up like a triangular space. The parameters of the first octant can be mapped to a triangle. The top vertex will be the value $\eta = \pi/2$, where η is the angle of elevation. The left bottom vertex is the value $\omega = 0$, $\eta = 0$, where ω is the azimuthal angle. The right bottom vertex is the value $\omega = \pi/2$, $\eta = 0$. Another possible method would be to subdivide first along isoparametric lines and then split any rectangular subdivisions into two or four triangles.

A particular subdivision method cannot be recommended at this point. The method must be chosen to meet the requirements of a specific surface, application and implementation. Factors, such as the growth of the number of triangles per subdivision, best approximation to original surface and ease of implementation, must be considered before a decision is reached. The iterative method is recommended if the growth of the number of the triangles is a concern. It also offers a smoother transition between levels of approximation. The recursive technique is substantially easier to implement.

Irregardless of the technique, one rule must be followed when subdividing. The subdivision must be uniform. All vertices of the triangles must only overlap vertices of adjacent triangles. A vertex of a triangle should never lie on an edge of a neighboring triangle. See Figure 2. Ignoring this precaution will undoubtedly lead to cracks in the composite surface. This result will become evident, as the algorithm is further explained.

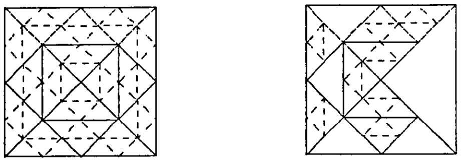


Figure 2. Uniform & Non-Uniform Subdivisions

How much should a surface be subdivided? Again, this is a question which does not have a single answer. The level of subdivision must be determined based on a number of factors, accuracy of approximation, visual effects and/or number of triangles.

Determination of Bernstein-Bezier Control Points

Once the parameter space has been triangulated, a Steiner patch is calculated for each triangle. This is done by calculating six Bernstein-Bezier control points for each resultant triangle. If each triangle in parameter space is mapped into a Steiner patch, a C0 tessellation of the original surface is produced.

Corner Points

The three corner points are the easiest of the six to calculate. The parameter values at the three vertices of the triangle are plugged into the approximated surface equation. This produces three points in 3-space. The points are the corner control points of the approximating Steiner patch. See Figure 3.

Interior Points

To determine the interior mid-edge control points is a bit more difficult. To begin, recall the properties of the Bernstein-Bezier control points for a Steiner patch. The corner control point and its two adjacent interior control points define the tangent plane of the patch at the corner point.

Now consider any of the interior control points. Each one lies on two tangent planes. They are the tangent planes of the patch at the two adjacent corner points. Intersecting the two tangent planes gives a line on which the interior mid-edge control points are allowed to lie. See Figure 4.

To determine a unique point on the line, requires a third plane. The third plane can be created by picking three points on the appropriate edge of the approximated surface. The three points are defined in parameter space as two vertices on a triangle and the midpoint of the edge lying between them. See Figure 5. Mapping these three parameter values into 3-space gives the three points. These three points define the third plane needed to determine a unique interior control point. See Figure 6.

There are numerous ways to calculate the third intersecting plane. The previous method was chosen because it provides orthogonal boundaries between the octants of geometric objects defined in a spherical parameter space. This will aid in the implementation of the algorithm and will be further explained at a later point.

The algorithm for calculating the interior control points on an approximating Steiner patch is:

For each edge of the triangular parameter space.

1. Calculate the points in 3-space which correspond to the two endpoints.

2. Calculate the tangent plane of the approximated surface at the two endpoints, by taking the cross-product of the tangent vectors emanating from an endpoint.

3. Calculate the point in 3-space which corresponds to the midpoint of the edge common to the two endpoints in parameter space.

4. Determine the plane which is defined by the three previously calculated points.

5. Intersect the three previously calculated planes. Their intersection point is the interior control point for that edge.

The algorithm is summarized in Figure 3 through Figure 7.

If followed, the algorithm will give three interior control points. With the three corner points previously calculated, the Steiner patch is defined. If the algorithm is performed for all triangles in the parameter space, the original surface will be approximated with Steiner patches. Discussion of the accuracy of the approximation is presented in my thesis[8].

Continuity Considerations

The tessellation algorithm creates a C0 mesh of Steiner patches which is C1 continuous at the data points, but is C1 discontinuous along Steiner patch boundaries.

When the parameter space of an approximated surface is uniformly subdivided, the algorithm guarantees positional continuity. When adjacent triangles share an edge in parameter space, they also share the points and tangents planes in 3-space which define the control points along their common edge. Since they have common control points along their border, there will be positional continuity along their common boundary edge[1,2].

At any vertex where Steiner patches come together, a single tangent plane exists. The algorithm guarantees that all adjacent interior control points to a corner point all lie on one plane. This guarantees that all patches at the corner point share the same tangent plane. The common tangent plane is an important geometric by-product of the tessellation algorithm that is exploited by the smooth-shading algorithm.

Smooth-Shading the Tessellation

Normal vectors which give a smooth-shading over a C1 discontinuous tessellation must meet certain criteria. They are

1. A single normal vector must be assigned to each common vertex.

2. The interior normals of one patch should be independent of other patches.

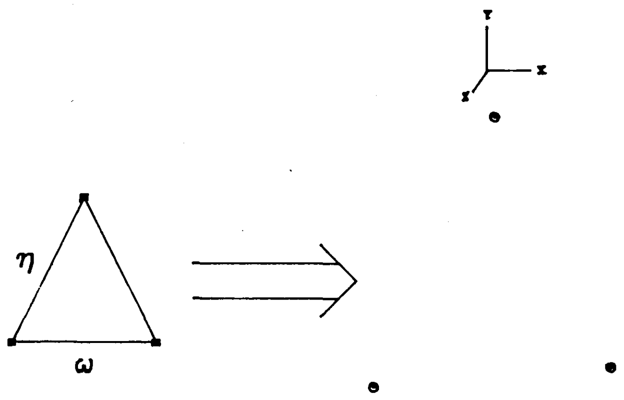


Figure 3. Calculate Corner Points

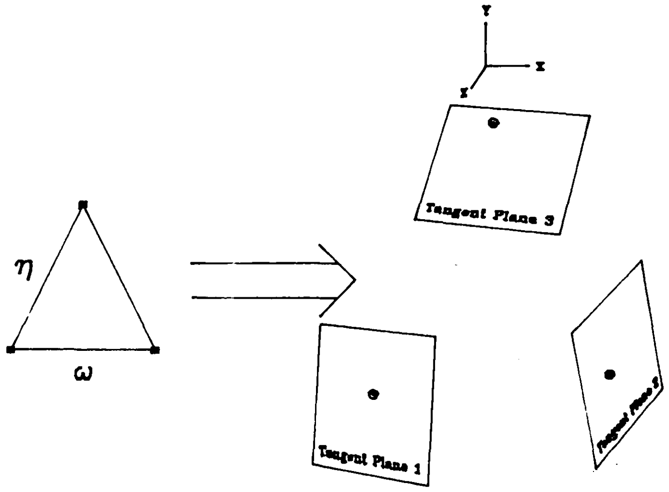


Figure 4. Calculate Tangent Planes at Corner Points

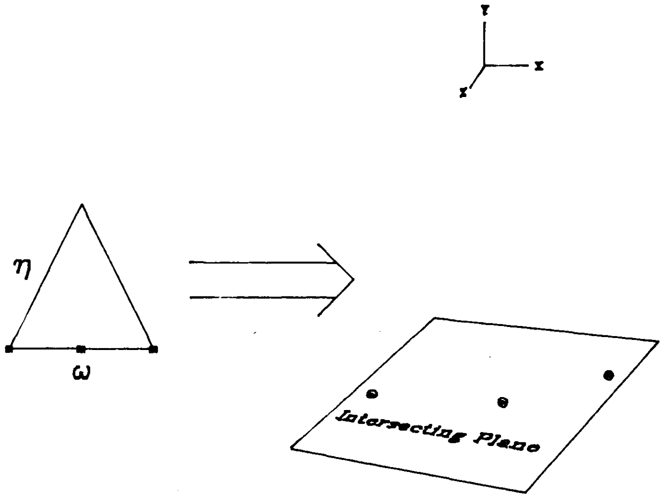


Figure 5. Determine Intersecting Plane for All Edges

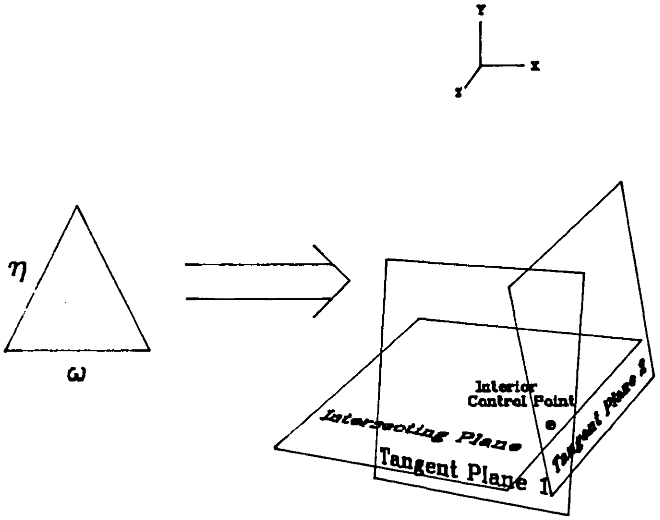


Figure 6. Intersect All Planes To Give Interior Control Point

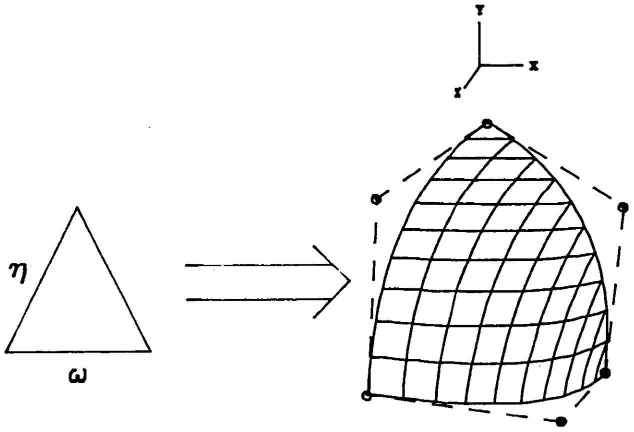


Figure 7. Six Points Define a Steiner Patch

3. The normal function should be continuous on the patch.

4. The normals along a boundary edge should only be a function of the normals at the endpoints of the boundary, i.e. an interpolation.

A normal vector function for each patch which meets these criteria has been developed and implemented for Steiner patch tessellations. It is based on the concept of normal vector interpolation presented by Phong[3], namely,

$$\vec{N}(u,v) = \vec{N0} * u + \vec{N1} * v + \vec{N2} * (1 - u - v) \qquad (2)$$

where N0 is the Steiner patch normal at (1,0), N1 is the normal at (0,1) and N2 is the normal at (0,0).

The first criteria is met by the tessellation algorithm itself. It guarantees that there is a unique tangent plane and therefore normal direction at each vertex of the tessellation. The formula explicitly states that the normals in the interior of the patch are a function of the normals at the vertices of the patch, making the normals independent of other patches. The function is continuous. It can also be noted that the normals along the boundary edges are a function only of the normals at the edge endpoints. This fact and the first criterion guarantee that there will be smooth and continuous shading across patch boundaries.

In order to smooth shade the C0 tessellation, the Steiner patches are ray traced. When a patch is intersected by a ray, the u,v value of the intersection is determined. Rather than calculating the actual normal of the Steiner patch at the intersection, the above formula is used during the shading calculation. The new normals provide a smooth shading across the whole network of Steiner patches.

Implementation

The tessellation and smooth shading algorithm has been implemented and incorporated into the CICG's animation system, The Clockworks[13, 14, 15]. The Clockworks is an object-oriented computer animation system implemented in C[16]. As previously stated, the algorithm was implemented to approximate superellipsoids. The superellipsoid was the first geometric primitive supported by the geometric modeling subsystem of The Clockworks, so it was a natural choice.

The first step in the implementation was to subdivide the parameter space of the superellipsoid. Because it is symmetric, Steiner patches are only calculated for the first octant and are reflected into the the other seven. The first octant of a superellipsoid is bounded by $\eta = 0$ to $\eta = \pi/2$ and $\omega = 0$ to $\omega = \pi/2$. η is the elevation angle and ω is the azimuthal angle. The octant was interpreted as a triangle and subdivided.

After some initial investigation an iterative subdivision implementation was chosen over a recursive one. The iterative subdivision was more difficult to implement but offered a smoother transition between levels of subdivision; therefore providing more subdivision options. For the iterative technique, the number of triangles per octant is n^2, where n is the level of the subdivision. Since the Steiner patches of the first octant are reflected into the other seven, the total number of Steiner patches for the whole superquadric is $8 * n^2$.

Results

The following pictures illustrate the preliminary results of the superquadric implementation. Figure 8 is an image comprised of four superquadrics ray-traced directly. Figure 9 is the same four superquadrics approximated by 72 Steiner patches each. This is a level 3 subdivision. The patches are ray-traced directly. The C1 discontinuities of the tessellation are visible near the poles of all four surfaces. Figure 10 is the same as Figure 9 except that smooth-shading is applied in order to blend the apparent surface normals. Figure 11 displays one superquadric approximated with four different levels of subdivision. The Steiner patches are ray-traced directly. The bottom right shape is the original superquadric. Again C1 discontinuities are evident across most patch boundaries. Figure 12 is the same as Figure 11 except that smooth-shading has been applied to to the tessellations removing visual discontinuities along patch boundaries. Figure 13 is two superquadrics ray traced directly. Figure 14 is the smooth shaded tessellation of the same geometry. It is only subdivided to two levels, which produces 32 Steiner patches.

Analysis

Figure 10, Figure 12, and Figure 14 provide visual evidence of the algorithm's capability. Still

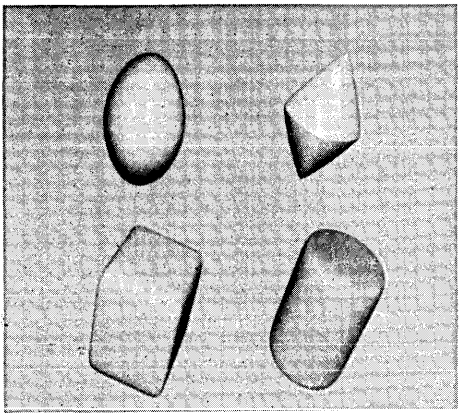


Figure 8. Four Superquadrics

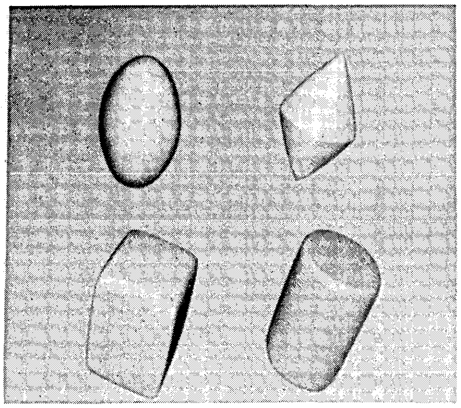


Figure 9. Four Tessellated Superquadrics

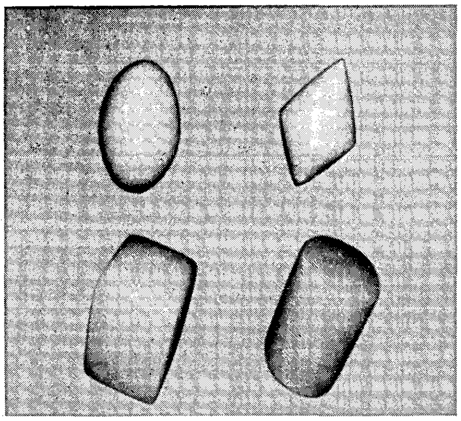


Figure 10. Smooth-Shaded Tessellations

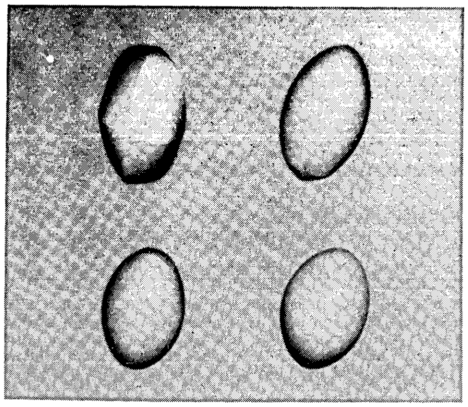


Figure 11. Four Levels of Tessellation

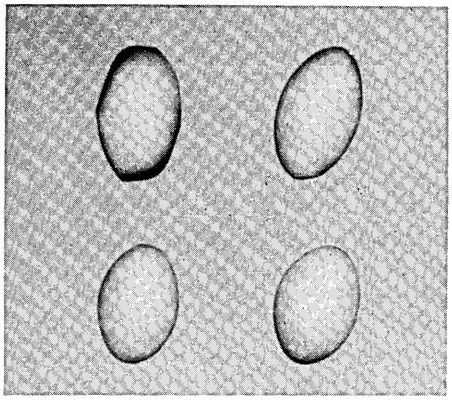


Figure 12. Smooth-Shaded Tessellation

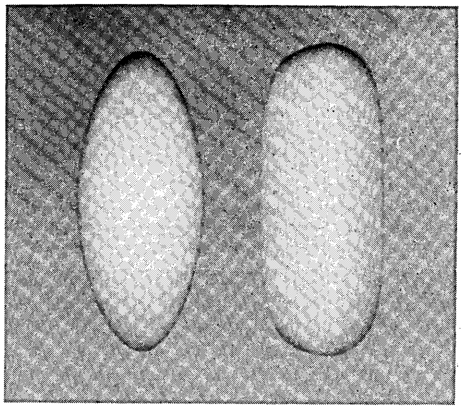


Figure 13. Two Superquadrics

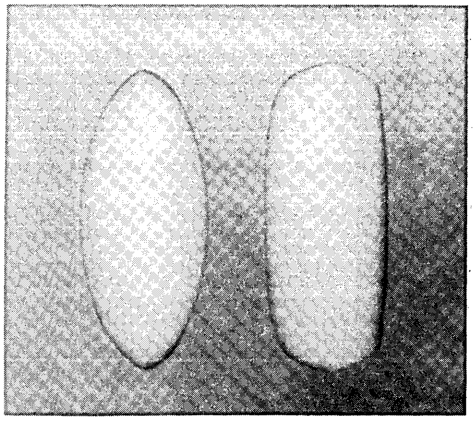


Figure 14. Smooth-Shaded Tessellations

many questions have to be answered. Do Steiner patches offer faster rendering speeds? Are Steiner patches worth the effort? Why use Steiner patches instead of polygons?

Preliminary tests were run to determine if Steiner patches had fulfilled their promise. Steiner patches were first explored in the hope that they would offer the user a way to shorten rendering times. The following tests were conducted and showed that there are certain advantages to using Steiner patches over other geometric primitives.

Steiner Patches Versus Superquadrics

The first tests that were conducted ray-traced superquadrics directly and then their Steiner patch tessellations. Is it faster to ray trace a superquadric directly or its Steiner patch tessellation? The geometry of Figure 8 and Figure 13 was used to answer this question. The results are favorable with the superquadric ray-tracing times being 125 CPU-minutes and 90 CPU-minutes on a Data General MV/10000 for the images of four and two superquadrics respectively. Ray-tracing the tessellations of these superquadrics with 72 Steiner patches each took 70 CPU-minutes and 56 CPU-minutes. Clearly there is a savings in time of 44% and 38% when using the Steiner patch approximation. Examining Figure 10 and Figure 14 shows that acceptable visual results are also produced.

Normalizing BART and STRAW

The next question investigated was "Is it more time-efficient to tessellate and smooth-shade a polygonal approximation rather than a Steiner patch approximation when ray tracing curved surfaces?". Even though the software at RPI prevents the forming of an outright conclusion, the results produced from benchmarking do give a strong indication that using Steiner patches is more efficient for ray tracing curved surfaces than polygons.

Using a rough midpoint error estimation, it was determined that 32 Steiner patches can approximate the surface of a superquadric as well as 200 triangles[8]. The method used to make the error estimation first calculates the midpoint along an edge of an approximating Steiner patch. Next the associated point in the approximated surface's parameter space is determined. It will be a midpoint on the edge of a parameter space triangle. That point is mapped into 3-space. The distance between the previous two points is calculated and averaged with the corresponding distance for each edge of the tessellation. The average value of these distances can be used to gauge the quality of the tessellation. A similar technique is applied to the approximating triangles. This is far from an exact error calculation, but is satisfactory for the scope of our initial investigations.

Our results indicate that a Steiner patch tessellation of level 2 will provide the same level of approximation to a superquadric as a planar approximation of level 5. A level 2 tessellation corresponds to 32 approximating entities, while level 5 translates to 200. It appears that 32 Steiner patches can approximate a superquadric as well as 200 triangles. It is this 32 to 200 ratio that was used to determine if it is more time-efficient to tessellate and smooth-shade Steiner patches rather than triangles when ray tracing curved surfaces.

Unfortunately a common software system which ray-traces both triangles and Steiner patches is unavailable at RPI. BART, a ray-tracer developed at RPI's Center for Interactive Computer Graphics, ray-traces superquadrics, Steiner patches, spheres, cubes and cylinders. STRAW, a ray-tracer developed at RPI's Image Processing Lab by Mike Potmesil, ray-traces spheres, polygons, quadric surfaces, and bicubic patches[17, 18]. These two programs were used to run a benchmarking time test of Steiner patches versus triangles. Another factor to be considered is that BART and STRAW reside on different machines. STRAW runs on a PRIME 500 and BART is on a DATA GENERAL MV/10000.

Since conclusions are being drawn from the use of two different pieces of software running on two different machines, a method had to be found to normalize the performance of the two programs. Common geometric primitives were identified. Each program had two geometric entities in common, spheres and cubes. BART detects spheres and cubes as degenerate superquadrics. STRAW supports spheres as a separate primitive. A cube can be constructed from six polygons in STRAW. Scenes with these two common primitives were constructed and ray-traced. One scene contained six spheres of various sizes. The other contained six cubes of various sizes. The CPU time needed to generate the images was noted. The ratio of the time it took to render the same geometry on the different programs can be used as a normalization factor. BART, on the average, used 10 CPU-minutes to generate the image of the spheres and 60 CPU-minutes to generate the cubes. STRAW needed an average of 45 CPU-minutes to render the spheres and 80 CPU minutes to render the cubes. From these rendering times a worst and best case normalization factor can be deduced. STRAW running on a PRIME 500 can take anywhere from 4.5 to 1.33 times longer to render a geometric scene than BART running on a DATA GENERAL MV/10000. These factors will be used in an attempt to normalize the results of the final test.

Steiner Patches Versus Triangular Polygons

The superquadrics in Figure 13 were tessellated with 32 Steiner patches and 200 triangular polygons each. The each set of the tessellated superquadrics was then ray-traced. These tessellations produced similar approximation error. On the average, BART used 42.5 CPU-minutes to render the image. The same superquadric tessellated with triangles rendered by STRAW used 13.5 CPU-hours. These are striking differences in rendering times. Even if the worst normalization factor of 4.5 is applied to STRAW's time of 13.5 hours on a PRIME 500, the rendering time is normalized to 3 CPU-hours on an MV/10000. This time still far exceeds the 42.5 CPU-minutes of BART's image. Ray-tracing a Steiner patch tessellation appears to be significantly quicker than ray-tracing a comparable polygonal tessellation.

In order to understand what might be happening during the rendering of the two tessellations, the facts must be reviewed. Using the worst case normalization, the triangular tessellation takes 4 times longer to render than the Steiner patch tessellation. There are 6.25 times more triangles being rendered than Steiner patches. It is assumed that the majority of the CPU-time of ray tracing an image without shadows, reflections and refractions (i.e. ray-casting) is spent calculating the intersections of a ray with the geometric primitives. This implies that it takes only 56% (1-6.25/4) more work to determine if and where a ray intersects a Steiner patch rather than a polygonal planar patch. The inverse statement is that the determination of the existence and position of a ray intersection on a triangular polygon is only 36% less work than Steiner patches. At first glance this is difficult to accept considering the complexity of Steiner patches and the simplicity of polygons. Examining the code of STRAW and BART provides further insight into the results.

The next step taken was the actual counting of operations needed to determine the existence and calculate the intersection of a line with a Steiner patch and triangular polygon. To calculate the intersection of a line and a Steiner patch takes approximately 450 multiplies/divisions and 400 additions/subtractions[19]. To calculate the intersection for a triangular polygon takes approximately 55 multiplies/divisions and 55 additions/subtractions. These numbers were initially deceiving, because it showed that Steiner patches took 820% more work than triangular polygons, not 56%. Further study of the code showed that 80 multiplies and 80 adds of the Steiner patch calculations were in bounding box and bounding hull checking. While no bounding surface calculations were being made for any individual triangular polygons. It is these last figures and the bounding surface structures in both renderers which gave the final insight into the results.

In BART bounding boxes and convex hulls are computed for each Steiner patch. While in STRAW the triangular polygons for a single superquadric are grouped together into a polyhedron. A single bounding sphere is then placed around the whole tessellation, not around any particular polygon[20]. Therefore when calculating a ray intersection with a Steiner patch, the ray intersections with the convex hull and bounding box are first computed. In most cases, there will be no intersections. Therefore the line-Steiner patch intersection operation will only take 80 multiplies and 80 adds. On the other hand STRAW will have to attempt to intersect almost every triangular polygon for every ray because the polygons have no bounding box information. This operation takes 55 multiplies and 55 adds. For the usual case it clearly makes sense to compare the triangular polygon intersection operations with the Steiner patch bounding box intersection operations. When that is done, the original result is produced. 80 / 55 = 1.45. In words, the bounding box and bounding hull intersection takes 45% (approximately 56%) more operations than then the triangular polygon intersection.

These results clearly show that the Steiner patch's greatest advantage is its convex hull. By exploiting the convex hull property of the patch, advances in rendering can be made. Steiner patch tessellations not only take up less space, but by utilizing their geometric properties wisely less time can be spent rendering complex geometry.

This experiment does not prove outright that using Steiner patches when tessellating and rendering curved surfaces will always prove to be more time-efficient than other methods. It does though, give an indication that Steiner patches offer an improved method for rendering in some instances. An obvious example would be where the Steiner patches are relatively small and polygons cannot be reasonably bounded. It is in this situation where the convex hull property of the Steiner patches can be exploited.

Conclusion

The generation of images which consist of Steiner patches offer visual proof of the validity of the tessellation and smooth-shading algorithm. The algorithm provides a positionally continuous mesh of Steiner patches which approximate a surface of arbitrary order. The Steiner patches can then produce a smooth looking surface by simply applying normal vector interpolation across each patch.

Tessellating and smooth-shading Steiner patches offers an improved and quicker way to render curved surfaces. In the case of superquadrics, a significant improvement in calculation times is evident. Ray tracing Steiner patches appears to be more efficient than ray tracing either superquadrics directly or their polygonal tessellations. One of our major goals has been attained by Steiner patches. They are time-savers during rendering.

The visual results of Steiner patch tessellations are good. With 3 levels of approximation (72 patches), a very good copy of a superquadric can be made with Steiner patches. There are visual differences though. Steiner patch tessellations can sometimes appear lumpy and the shading is not as evenly distributed as it could be. Even with this in mind, Steiner patches can offer a remarkable approximation to superquadrics.

Steiner patch tessellations also offer clear-cut advantages over polygonal tessellations. They not only reduce rendering times, but also use less memory during processing because fewer Steiner patches are needed to approximate a curved surface as compared to polygons. Also the silhouette of a Steiner patch tessellation is curved, not jagged like a polygonal tessellation, making Steiner patches visually superior to polygonal patches. The one disadvantage of Steiner patches is the complexity of the code needed to intersect a Steiner patch with a line. This is a one time cost that can pay for itself if the code is extensively used.

Acknowledgements

I would like to acknowledge the assistance of Dr. Michael Wozny, Dr. Harry McLaughlin, Mr. Phillip Getto and Mr. Douglas Lyon. The raster images of this paper were produced by software written by Phillip Getto and Gray Lorig.

This work was supported by NSF Grant No. ISP79-20240 and other industry grants. Any opinions, findings, conclusions or recommendations do not necessarily reflect the views of the National Science Foundation or the industrial sponsors.

References

[1] D. F. Rogers, PROCEDURAL ELEMENTS FOR COMPUTER GRAPHICS, McGraw-Hill Book Co., New York, NY, 1985, pp. 264-5.

[2] H. Gouraud, "Continuous Shading of Curved Surfaces," IEEE TRANSACTIONS ON COMPUTERS, Vol. C-20, No. 6, June 1971, pp. 623-628.

[3] B. Phong, "Illumination for Computer Generated Pictures," COMMUNICATIONS OF THE ACM, Vol. 18, No. 6, June 1975, pp. 311-317.

[4] J. D. Foley and A. Van Dam, FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS, Addison-Wesley Publishing Co., Reading, MA, 1982, pp. 580-584.

[5] W. Sederberg and D. C. Anderson, "Steiner Surface Patches," IEEE COMPUTER GRAPHICS AND APPLICATIONS, Vol. 5, No. 5, May 1985, pp. 23-36.

[6] T. W. Sederberg and D. C. Anderson, "Ray Tracing of Steiner Patches," SIGGRAPH '84 PROCEEDINGS, published as COMPUTER GRAPHICS, Vol. 18, No. 3, August 1984, pp. 159-164.

[7] T. W. Sederberg, "Implicit and Parametric Curves and Surfaces for Computer Aided Geometric Design," Ph.D. Thesis, Purdue University, 1983.

[8] D. E. Breen, "Creation and Smooth Shading of Steiner Patch Tessellations," RPI CICG Technical Report TR-85016, August 1985. (Also Master's Thesis)

[9] "Jakob Steiner", DICTIONARY OF SCIENTIFIC BIOGRAPHY, Charles Scribner's Sons, New York, Vol. XIII, 1976, pp. 12-22.

[10] W. Bohm, G. Farin, J. Kahmann, "A Survey of Curve and Surface Methods in CAGD," "Triangular Patch Surfaces," COMPUTER AIDED GEOMETRIC DESIGN, Vol. 1, No. 1, 1984, pp. 38-48.

[11] A. H. Barr, "Superquadrics and Angle-Preserving Transformations," IEEE COMPUTER GRAPHICS AND APPLICATIONS, Vol. 1, No. 1, January 1981.

[12] I. D. Faux and M. J. Pratt, COMPUTATIONAL GEOMETRY FOR DESIGN AND MANUFACTURE, Ellis Horwood Ltd., Chichester, UK, 1979, pp. 210-223.

[13] D. R. McLachlan, "CORY: An Animation Scripting System," RPI CICG Technical Report TR-85006, May 1985. (Also Master's Thesis)

[14] D. E. Breen, A. A. Apodaca, P. H. Getto, "The CLOCKWORKS: An Implementation of an Object-Oriented Computer Animation System in a Conventional Programming Environment," RPI CICG Technical Report TR-86013, May 1986.

[15] D. W. Shen, "CLOCKWORKS Animation Software Reference Manual," RPI CICG Technical Report TR-86016, May 1986.

[16] B. W. Kernighan and D. M. Ritchie, THE C PROGRAMMING LANGUAGE, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[17] M. Potmesil and I. Chakravarty, "Synthetic Image Generation with a Lens and Aperture Camera Model,", ACM TRANSACTIONS ON GRAPHICS, Vol. 1, No. 2, April 1982.

[18] D. Lyon, STRAW USER'S MANUAL, RPI Image Processing Laboratory Report, December 1984.

[19] Mr. Phillip Getto, Private Conversations, August 1985.

[20] Mr. Douglas Lyon, Private Conversations, August 1985.

An Algorithm for Normal Vector Interpolation on Polygonal Surfaces

Phillip H. Getto

Rensselaer Polytechnic Institute
Center for Interactive Computer Graphics
Troy, New York

Abstract

A new algorithm for the interpolation of ''smoothed'' normals across a polygonal surface is presented. The algorithm produces surface normals which are invariant under rotation of the image plane about its normal. Different versions are derived for use with visible surface algorithms which use scan line coherence or do not use any coherence, such as ray tracing. An analysis of the time complexity for each version of the algorithm is shown.

Introduction

Computer synthesized imagery of curved surfaces is often generated in three steps. First, the surfaces are approximated by polygonal surfaces. Second, the visible polygons or parts of polygons are determined. Finally, the intensity of the visible polygon or polygons is calculated for each point on the image plane. The intensity is a function of many parameters, including the surface normal, material characteristics and light sources. The specific parameters and calculations vary from shading model to shading model.[1,2,3,4] However, all but a few, use the surface normal extensively. To produce realistic images of curved surfaces the polygonal normals must be massaged in some way. Otherwise, discontinuities of the normal across the surface will be visible.

Gouraud [5] originally described a method to interpolate intensities across the surface to ''smooth'' out the resulting image. The intensity of each vertex of a polygon is calculated from either the normal to the original surface at the vertex or the average of the normals to the polygons sharing the vertex, using any shading model desired. The intensity of a point on an edge is a weighted average of the intensities of the vertices on the edge. The intensity of an interior point is a weighted average of the intensities at the points of intersection of the scan line and the edges of the polygon.

Phong[6] described an improved algorithm, in which normals are interpolated across the edges and scan lines rather than intensities. The intensity of each pixel is computed using any desired shading model and the interpolated normal.

Both Gouraud's and Phong's interpolation techniques have serious drawbacks. First, only local information is used to compute the intensity of a pixel. That is, the intensity is a function of the intensities / normals only at the vertices of the edges intersecting the current scan line. Because of the locality of the information used to compute the intensities / normals, a vertex on an edge intersecting the current scan line will be used, even if there exists a closer vertex. If only a subset of the vertices on a polygon are used to interpolate an intensity / normal, it would be desirable to use the closest ones, as they would presumably better represent the point in question. With these algorithms, it is not possible for all of the vertices on a polygon to contribute to an interpolated intensity / normal. This can lead to some very undesirable artifacts.

The second problem with the Gouraud and Phong techniques is the lack of rotation invariance. By rotation invariance, we mean the intensity / normal of a point on a polygon should remain the same when the image plane is rotated about its normal, (vis-a-vis, the eye is rotated), or alternatively, when the entire model, including lights, is rotated about the axis normal to the image plane. This is not the case in the Gouraud and Phong schemes.

To illustrate, consider an octagon, as in Figure 1 below.

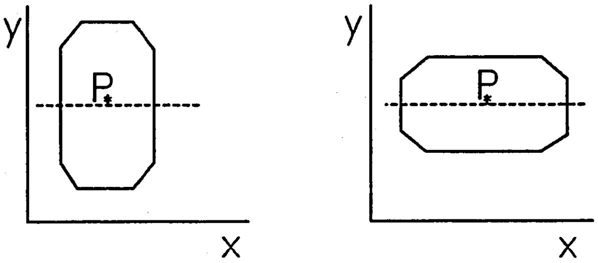


Figure 1: Normal interpolation over an octagon, using Phong's algorithm

Without loss of generality, assume the current scan line passes through the central vertical edges. If we are to rotate the image plane about 90 degrees its normal, (turn the page on its side), we would note the current scan line would intersect the two central horizontal edges. Since the edges do not share end points the normals can be assumed to be different. Thus, at the intersection point of the two scan lines, the resulting interpolated intensity / normal

would be different.
The remainder of this paper describes an algorithm, which overcomes the problems described above. Variations of the algorithm are presented for point sampling, ray tracing,[3,4,7,8] and scan line coherent visible surface algorithms. An analysis the complexity of the algorithm is made, for each version.

Normal Vector Interpolation Algorithm

The Gouraud and Phong schemes suffer from artifacts and lack rotation invariance, because they always interpolate along the current scan line, and they do not use global information about the polygon. If algorithms which always interpolated along a scan line, were to use a combination of the all of the normals the interpolated values would be rotation invariant. In fact, for triangles, the Gouraud and Phong interpolated normals are rotation invariant, because normals from all three vertices have a part in all interpolations.

Alternatively, if the line along which the interpolations are computed were to remain constant, relative to the polygon, under a rotation of the image plane about its normal, then the interpolated normals would be rotation invariant; even if only subset of the the polygon's normals are used in the computation. We will now present an algorithm which uses a combination of these two methods to be immune from scan line artifacts and be rotationally invariant.

The question of interest is how to chose the line along which we will interpolate. For the sake of clarity in what follows immediately, we will restrict ourselves to convex polygons. An easy solution is to select the line of constant u or constant v in the plane of the polygon going through the point of interest as in Figure 2 below.

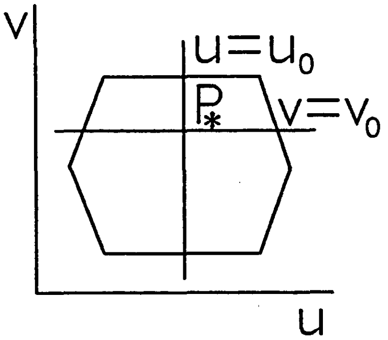


Figure 2: Interpolation along lines of constant u or v

One could use an interpolation scheme similar to Phong's, except the interpolation would be along the line $u = u_*$ or $v = v_*$. This solution provides the desired rotation invariance. However, it is subject to the same artifacts as Phong's scheme, because it does not use the normals at vertices on the edges not intersecting the line of interpolation. A better solution which can use all of the normals follows.

Chose the line of interpolation, for a given point $\overline{P}_*$, to the be line through the centroid of the polygon, $\overline{P}_c$, and $\overline{P}_*$, as in Figure 3 below.

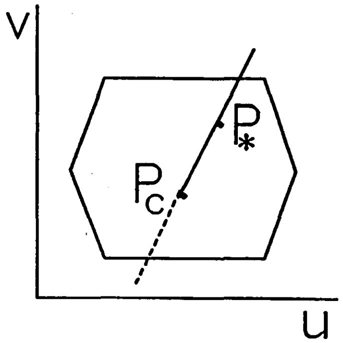


Figure 3: Interpolation along the ray from $\overline{P}_c$ through $\overline{P}_*$

Instead of interpolating the normal at $\overline{P}_*$ by averaging the normals at $\overline{P}_{e_1}$ and $\overline{P}_{e_2}$, let it be the weighted average of the normals at $\overline{P}_c$ and $\overline{P}_{e_1}$. If we define the normal at $\overline{P}_c$ to be the average of the normals at all of the vertices, then $\overline{N}_*$ is a function of all of the normals of the polygon. In effect we are interpolating along a ray from the centroid through the point of interest. As the point moves around the polygon different edges are used in the interpolation, in effect triangulating the polygon.

The following presents the algorithm in a more formal manner, first comparing its results to that of a baricentric parameterization of a triangle. The algorithm is then extended to convex polygons and to non-convex polygons, possibly including holes. We will then derive a method for incrementally calculating the interpolation parameters. A time complexity analysis will be given for each version.

Random Point Sampling Version

Consider, for the moment a triangle, (for which the Gouraud and Phong techniques have the desired characteristics). If we represent a point on the triangle using a baricentric coordinate system we have the following,

$$\overline{P}(\overline{\lambda}) = \lambda_0\overline{P}_0 + \lambda_1\overline{P}_1 + \lambda_2\overline{P}_2, \tag{1}$$

$$\sum_{i=0}^{2} \lambda_i = 1, \quad \lambda_i \geq 0.$$

Then, the optimal interpolated normal is expressed as,

$$\overline{N}(\overline{\lambda}) = \lambda_0\overline{N}_0 + \lambda_1\overline{N}_1 + \lambda_2\overline{N}_2. \tag{2}$$

The λ's provide a natural weighting of all of the normals, and are rotation invariant, exactly fitting the desired characteristics. However, it is not possible, in general, to determine the λ's for a planar polygon with more than three vertices. One obvious solution is to limit the polygons, for which we will interpolate normals, to triangles; expression (1) could then be solved for the λ's. If we are willing to restrict ourselves to triangles, we could use the Gouraud or Phong techniques which have the desired characteristics for triangles. This solution may be acceptable in certain situations, especially when triangular data is given; however, it seems too limiting in general. Furthermore, triangulating complex polygons is slow and generates many more polygons.

A better solution in most cases is to find some sort of extension to the baricentric coordinates, which still preserves the desired characteristics. Before we show our proposed extension on a general polygon, we first prove it is equivalent to the baricentric coordinate system for a triangle. The algorithm is then developed to handle convex polygons and finally non-convex polygons possibly containing holes.

Consider the triangle below.

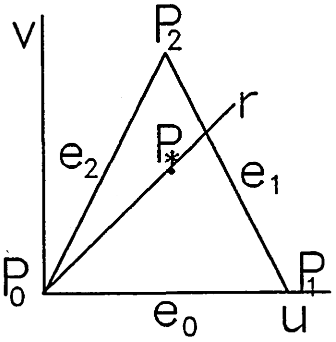


Figure 4: Representative triangle

We define the origin of the plane containing the triangle to be at $\bar{P}_0$, and $\bar{u} \parallel (\bar{P}_1 - \bar{P}_0)$. Note, $(u_0, v_0) = (0, 0)$, $v_1{=}0$, $u_1{\neq}0$, and $v_2{\neq}0$. Let $\bar{P}_*$ be the point of interest on the triangle.

We can represent $\bar{P}_*$ by a ray from $\bar{P}_0$ through $\bar{P}_*$;

$$\bar{r}_*(\alpha) = \bar{P}_0 + \alpha(\bar{P}_* - \bar{P}_0) = \alpha\bar{P}_*, \quad \alpha \geq 0. \qquad (3)$$

We claim this representation of the triangle is equivalent to the baricentric representation; further we can extend the definition to arbitrary polygons possibly containing holes. We will now show the two representations are equivalent.

In baricentric coordinates we have $\bar{P}(\bar{\lambda}) = \lambda_0\bar{P}_0 + \lambda_1\bar{P}_1 + \lambda_2\bar{P}_2 = \lambda_1\bar{P}_1 + \lambda_2\bar{P}_2$, because $\bar{P}_0 = \bar{0}$. Thus,

$$\begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} u_* \\ v_* \end{bmatrix},$$

but $v_1 = 0$,

$$\begin{bmatrix} u_1 & u_2 \\ 0 & v_2 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} u_* \\ v_* \end{bmatrix},$$

substituting we get,

$$\lambda_1 = \frac{u_*v_2 - v_*u_2}{u_1v_2}, \text{ and } \lambda_2 = \frac{v_*}{v_2}.$$

Now, define the equation of edge k as,

$$\bar{e}_k(\beta) = \bar{P}_k + \beta(\bar{P}_{k+1} - \bar{P}_k), \quad 0 \leq \beta \leq 1. \qquad (4)$$

Define $\bar{P}_e$ as the intersection of $\bar{r}_*(\alpha)$ and $\bar{e}_k(\beta)$. Then,

$$\bar{P}_e = \bar{r}_*(\alpha_e) = \bar{e}_k(\beta_e),$$

substituting and rearranging,

$$\alpha_e\bar{P}_* - \beta_e(\bar{P}_{k+1} - \bar{P}_k) = \bar{P}_k.$$

By Cramer's Rule,

$$\alpha_e = \frac{u_1v_2}{u_*v_2 - v_*(u_2 - u_1)}, \text{ and}$$

$$\beta_e = \frac{u_1v_*}{u_*v_2 - v_*(u_2 - u_1)}.$$

At α_e, $\bar{r}_*(\alpha_e) = \bar{P}_e = \alpha_e\bar{P}_*$, or $\bar{P}_* = \bar{P}_e/\alpha_e$. Substituting into the equation for edge 1 and rearranging,

$$\bar{P}_* = \frac{(1-\beta_e)}{\alpha_e}\bar{P}_1 + \frac{\beta_e}{\alpha_e}\bar{P}_2,$$

which has the same form as the expression in baricentric coordinates. We must now show the coefficients of $\bar{P}_1$ and $\bar{P}_2$ are λ_1 and λ_2 respectively.

$$\frac{1-\beta_e}{\alpha_e} = \frac{\dfrac{u_*v_2 - v_*u_2}{u_*v_2 - v_*(u_2 - u_1)}}{\dfrac{u_1v_2}{u_*v_2 - v_*(u_2 - u_1)}}$$

$$= \frac{u_*v_2 - v_*u_2}{u_1v_2} = \lambda_1,$$

$$\frac{\beta_e}{\alpha_e} = \frac{v_*}{v_2} = \lambda_2.$$

Extension to Convex Polygons To extend the interpolation algorithm from triangles to arbitrary convex polygons is straight forward; but it provides an interesting insight. Given a convex polygon chose some point, denoted $\bar{P}_c$, then calculate the surface normal there. The choice of $\bar{P}_c$ is arbitrary, but a good choice is the centroid of the polygon. $\bar{N}_c$, the normal at $\bar{P}_c$, can be taken to be the average of the normal at each of the vertices, the normal to the original surface at $\bar{P}_c$, or something completely different. (The first choice will lead the following algorithm to make the normal at any point on the interior of the polygon a function of the normal at each of the vertices.) We then define the ray from $\bar{P}_c$ through the point $\bar{P}_*$ as,

$$\bar{r}_*(\alpha) = \bar{P}_c + \alpha(\bar{P}_* - \bar{P}_c), \qquad (5)$$

in an analogous way to the definition of $\bar{r}_*(\alpha)$ for a triangle. We can now compute the intersection of the $\bar{r}_*(\alpha)$ and $\bar{e}_k(\beta)$.

$$\bar{P}_e = \bar{r}_*(\alpha_e) = \bar{e}_k(\beta_e)$$

implies,

$$\bar{P}_c + \alpha_e(\bar{P}_* - \bar{P}_c) = \bar{P}_k + \beta_e(\bar{P}_{k+1} - \bar{P}_k).$$

Rearranging and applying Cramer's Rule,

$$\alpha_e = \frac{u'_k \Delta v_k - v'_k \Delta u_k}{u'_* \Delta v_k - v'_* \Delta u_k}, \text{ and}$$

$$\beta_e = \frac{u'_* v'_k - v'_* u'_k}{u'_* \Delta v_k - v'_* \Delta u_k},$$

where $\bar{P}'_* = \bar{P}_* - \bar{P}_c$, $\bar{P}'_k = \bar{P}_k - \bar{P}_c$, and $\Delta\bar{P}_k = \bar{P}_{k+1} - \bar{P}_k$. It is interesting to note, the definition of equation (5) and the choice of $\bar{P}_c$ have the effect of triangulating the polygon, as shown in Figure 5 below.

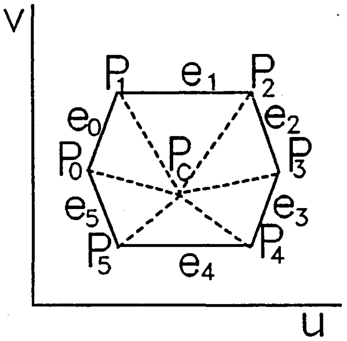


Figure 5: Triangulated polygon

The normal at the point $\bar{P}_*$ is given by an expression similar to (5)

$$\bar{N}_*(\alpha) = \bar{N}_c + \alpha(\bar{N}_* - \bar{N}_c). \tag{6}$$

However, unlike the expression for $\bar{P}_*$, $\bar{N}_*$ is unknown, in fact, it is the desired quantity. To solve for $\bar{N}_*$ we will write the expression for $\bar{N}_e$, the normal at $\bar{P}_e$, in terms of $\bar{N}_*(\alpha)$.

$$\bar{N}_e = \bar{N}_*(\alpha_e) = \bar{N}_c + \alpha_e(\bar{N}_* - \bar{N}_c). \tag{7}$$

But $\bar{N}_e$ along edge k at $\bar{P}_e$ is given, as in previous algorithms by,

$$\bar{N}_e = \bar{N}_k(\beta_e) = \bar{N}_k + \beta_e(\bar{N}_{k+1} - \bar{N}_k). \tag{8}$$

Equating equations (7) and (8) and solving for $\bar{N}_*$ gives,

$$\bar{N}_* = \bar{N}_c + \left[\frac{1}{\alpha_e}\right](\bar{N}'_k + \beta_e \Delta\bar{N}_k),$$

where $\bar{N}'_k = \bar{N}_k - \bar{N}_c$, and $\Delta\bar{N}_k = \bar{N}_{k+1} - \bar{N}_k$. The interpolated normals from our algorithm are identical to those interpolated using the baricentric coordinate parameters, for a given triangle.

Extension to Non-convex Polygons The extension of the algorithm to handle non-convex polygons, possibly containing holes, muddles the issue somewhat, and is more of an engineering solution than a clean mathematical one. With that caveat, let us consider Figure 6 below, a non-convex polygon containing a single hole.

The basic idea is to remember the edges whose intersection with $\bar{r}_*(\alpha)$ are closest to $\bar{P}_*$ on the positive and negative sides. That is, the edges where α_e is the least positive and the greatest negative are denoted $e+$ and e_-. In this case, $e+$ and e_- are edges e_0 and e_9 respectively, as they have the closest intersections to $\bar{P}_*$. The interpolation of $\bar{N}_*$ is a weighted average of the normals at the intersection points. The normals at the intersection points are interpolated from the normals at the edges' vertices. This procedure is analogous to Phong's, except we interpolate along $\bar{r}_*(\alpha)$, rather than the current scan line.

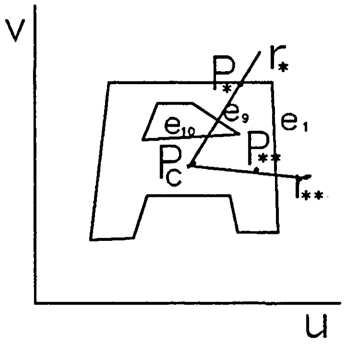


Figure 6: Non-convex polygon with a hole, $\bar{P}_c$ inside

It should be noted that if $\bar{P}_c$ is inside the polygon, it must be considered when determining the closest points to $\bar{P}_*$. If $\bar{P}_c$ is, as shown in Figure 7, outside of the polygon, then the value of $\bar{N}_c$ is not important because it will not be used. When $\bar{P}_c$ is inside, as in Figure 6, $\bar{N}_c$ will be used and can be computed as in the convex case. For example, in Figure 6, the normal at $\bar{P}_{**}$ would be a combination of $\bar{N}_c$ and $\bar{N}_{e_1}$, as in the convex case.

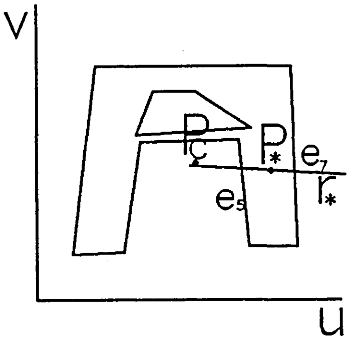


Figure 7: Non-convex polygon with a hole, $\bar{P}_c$ outside

Complexity The time and space complexity of our interpolation scheme is not significantly greater than that required to determine if $\bar{P}_*$ is inside the polygon. One well known method to determine if a point is inside a polygon is to intersect a ray, originating at the point, in a given direction, with each edge of the polygon. If the number of intersections is odd the point is inside, otherwise it is outside. If we rewrite equation (5) as $\bar{r}_*(\alpha) = \bar{P}_* + \alpha(\bar{P}_* - \bar{P}_c)$, then we count intersections, for inside / outside determination, when $\alpha \geq 0$. However, the edges whose intersections with $\bar{r}_*(\alpha)$ have the least positive and greatest negative values of

α, must be saved to use in the interpolation. If the greatest negative value of α is less than -1, then $\overline{P}_c$ is closer to $\overline{P}_*$ than any other edge's inside / outside test ray intersection point. In the original form of equation (5), we count intersections where $\alpha \geq 1$ for inside / outside determination, and use $\overline{N}_c$ if the greatest value of α that is less than 1 is also less than 0.

The intersection of the $\overline{r}_*(\alpha)$ and $\overline{e}_k$ can be made robust by writing $\overline{r}_*(\alpha)$ as an implicit function of u and v. An intersection occurs if the vertices of an edge are on opposite sides of the ray, or if one is on or near the ray, and the other is on a predetermined side, say positive. Edges with both vertices on the ray, or one on the ray and the other on the wrong side (negative), are not counted.[9]

The calculations can be sped up by precomputing and storing various values, such as $\overline{P'}_k$, $\overline{N'}_k$, $\Delta\overline{P}_k$, and $\Delta\overline{N}_k$, for $k = 0,\ldots,n$. Note, it is not necessary to retain the original values of P_k or N_k for the interpolation scheme.

Scan Line Coherent Version

The previous development assumed the polygon was randomly sampled and as such did not make use of any form of coherence from one point to the next. We will now present a development which uses the coherence of a polygon, along a scan line, so that the ray inside / outside test is not required to determine the edge k. The values of α_e and β_e will be computed incrementally, and the current edge k will be tracked around the polygon. The presentation here is for convex polygons; the extension for non-convex polygons is analogous to that for the point sampling version.

In the previous version the vertices of the polygons were represented in the (u,v) coordinates of the plane containing the polygon. Here we assume the polygons have been projected onto the image plane, and the vertices are represented in the (u,v) space of the image plane.

Define the inside / outside test ray in terms of the current pixel as,

$$\overline{r}_{i,j}(\alpha) = \overline{P}_c + \alpha(\overline{I}(i,j) - \overline{P}_c), \tag{9}$$

where i,j are the indices of the current pixel on the screen, and $\overline{I}(i,j) = (u(i),v(j))$ is the current point on the image plane.

When $\overline{I}(i,j)$ is on $\overline{e}_k$,

$$\overline{P}_e = \overline{I}(i,j) = \overline{P}_c + \alpha_{e_{i,j}}(\overline{P}_e - \overline{P}_c),$$

which implies $\alpha_{e_{i,j}} = 1$. In that case, solving equation (4) at $\overline{I}(i,j)$ for $\beta_{i,j}$ gives,

$$\beta_{i,j} = \frac{\overline{I}(i,j) - \overline{P}_k}{\Delta\overline{P}_k} = \frac{u(i) - u_k}{\Delta u_k}, \text{ or } \frac{v(i) - v_k}{\Delta v_k},$$

for $\Delta u_k \neq 0$, and for $\Delta v_k \neq 0$ respectively. As we step from pixel i to pixel $i+1$ along scan line j we would like to incrementally calculate $\alpha_{e_{i,j}}$ and $\beta_{e_{i,j}}$. Solving for them at $(i+1,j)$ we have,

$$\alpha_{e_{i+1,j}} = \frac{u'_k \Delta v_k - v'_k \Delta u_k}{u'(i+1)\Delta v_k - v'(j)\Delta u_k}, \text{ and} \tag{10}$$

$$\beta_{e_{i+1,j}} = \frac{u'(i+1)v'_k - v'(j)u'_k}{u'(i+1)\Delta v_k - v'(j)\Delta u_k}. \tag{11}$$

We can compute the difference $\Delta\alpha_{e_{i,j}} = \alpha_{e_{i+1,j}} - \alpha_{e_{i,j}}$, in general as the following.

$$\Delta\alpha_{e_{i,j}} = \frac{u'_k \Delta v_k - v'_k \Delta u_k}{u'(i+1)\Delta v_k - v'(j)\Delta u_k} - \tag{12}$$

$$\frac{u'_k \Delta v_k - v'_k \Delta u_k}{u'(i)\Delta v_k - v'(j)\Delta u_k}.$$

After some rearranging,

$$\Delta\alpha_{e_{i,j}} = \left[\frac{\Delta v_k (u'(i) - u'(i+1))}{u'(i+1)\Delta v_k - v'(j)\Delta u_k}\right]\alpha_{e_{i,j}}.$$

However, if we look at the difference of the numerators and denominators independently, we can do better. Let us denote the numerator of f as $n(f)$, and likewise the denominator as $d(f)$. By inspection of equation (12), $\Delta n(\alpha_{e_{i,j}}) = 0$, and $\Delta d(\alpha_{e_{i,j}}) = (u'(i+1) - u(i))\Delta v_k$. We can now write $\alpha_{e_{i+1,j}}$ as a function of the numerator and denominator of $\alpha_{e_{i,j}}$, as follows.

$$\alpha_{e_{i+1,j}} = \frac{n(\alpha_{e_{i,j}})}{d(\alpha_{e_{i,j}}) + (u'(i+1) - u'(i))\Delta v_k}. \tag{13}$$

Recall, $\alpha_{e_{i,j}} = 1$ when $\overline{I}(i,j)$ is on edge k, which occurs at the beginning of the each span, and $\Delta n(\alpha_{e_{i,j}}) = 0$. Therefore, $n(\alpha_{e_{i,j}}) = 1$, which implies,

$$\alpha_{e_{i+1,j}} = \frac{1}{\frac{1}{\alpha_{e_{i,j}}} + (u'(i+1) - u'(i))\Delta v_k}.$$

Recalling equation (6), we note that $\alpha_{i,j}$ is not required, only $1/\alpha_{e_{i,j}}$ is needed. Thus, let us define $\hat{\alpha}_{e_{i+1,j}}$ as $1/\alpha_{e_{i,j}}$, or

$$\hat{\alpha}_{e_{i+1,j}} = \hat{\alpha}_{e_{i,j}} + \Delta u(i)\Delta v_k.$$

Similarly, we will compute Δ's for the numerator and denominator of $\beta_{e_{i+1,j}}$ separately and combine the results at the end.

$$\Delta n(\beta_{e_{i,j}}) = (u'(i+1) - u'(i))v'_k.$$

Note $d(\beta_{e_{i,j}}) = d(\alpha_{e_{i,j}}) = \hat{\alpha}_{e_{i,j}}$. Finally,

$$\beta_{e_{i+1,j}} = \frac{n(\beta_{e_{i,j}}) + \Delta u(i)v'_k}{\hat{\alpha}_{e_{i+1,j}}}.$$

The normal at $\overline{I}(i+1,j)$ can then be calculated directly from equation (6).

If $\beta_{e_{i,j}}$ is not between 0 and 1, then $\overline{r}_{i,j}(\alpha)$ does not intersect edge k. If $\beta_{e_{i,j}} > 1$ then proceed to edge $k+1$, otherwise to edge $k-1$, and recompute $\alpha_{e_{i,j}}$ and $\beta_{e_{i,j}}$. It is possible to skip an edge entirely if it is very short compared to the step size. If $\hat{\alpha}_{e_{i,j}} = 0$ the ray is parallel to the edge, and computation must proceed to the next edge.

Complexity The number of computations above Phong's algorithm is a small constant per pixel. Instead of incrementally computing the interpolated normal from the interpolated edge normals, we incrementally compute $\hat{\alpha}_{e_{i+1,j}}$ and $\beta_{e_{i+1,j}}$. Computing α requires 1 add and 1 multiply, $(u'(i+1) - u'(i)$ is assumed to be calculated elsewhere). Unfortunately, β requires, 1 add, 1 multiply and 1 divide. From $\alpha_{e_{i+1,j}}$ and $\beta_{e_{i+1,j}}$, $\overline{N}_{i+1,j}$ is computed using 6 adds, and 6 multiplies, given $\overline{N}'_k$ and $\Delta\overline{N}_k$. Phong's algorithm requires 3 adds per pixel; a difference of 5 adds, 8 multiplies, and 1 division per pixel. For an image with 256K pixels, this would require, at most, about 4M extra operations. If not all of the pixels were covered the number would be further reduced. There is an additional cost when moving from one edge to the next, for which we have not accounted. However, even if it were to require a similar number of operations, current microprocessors could process the extra computations in about 10 to 15 seconds. Empirical studies are underway to quantify the differences.

Conclusions

We have examined two existing algorithms for the interpolation of intensities or normals across a polygonal surface. Both algorithms will "smooth" out most visual discontinuities in the rendering of the surface. However, the algorithms have two primary shortcomings; the inability to use the normals at more than four of a polygon's vertices, and the lack of invariance under a rotation about the image plane normal.

We have presented an algorithm which overcomes both of these objections. Rotation invariance is accomplished by always interpolating along a fixed ray, from a given point, $\overline{P}_c$, through the point of interest. By selecting $\overline{N}_c$ as the average of the normals at all of the vertices on the polygon, the interpolated normal becomes a function of the normals at all of the vertices on the polygon. The algorithm is well suited for visible surface algorithms which use random point sampling, that is, do not use coherence, such as ray tracing. It is, in the authors opinion, easier to implement and faster to execute, than a Phong model when using such a visible surface algorithm.

A method of incrementally calculating the interpolation parameters, for use with visible surface algorithms which use scan line coherence, has been derived. Analysis indicates that it does not use significantly greater resources than previous algorithms.

References

1. J. F. Blinn, "Models of Light Reflection for Computer Synthesized Pictures," *Computer Graphics* **11, 2** pp. 192-198 (1977).

2. R. L. Cook and K. E. Torrance, "A Reflectance Model for Computer Graphics," *ACM Transactions on Graphics* **1, 1** pp. 7-24 (January 1982).

3. R. A. Hall and D. P. Greenberg, "A Testbed for Realistic Image Synthesis," *IEEE Computer Graphics and Applications* **3, 8** pp. 10-20 (November 1983).

4. T. Whitted, "An Improved Illumination Model for Shaded Display," *Graphics and Image Processing* **23, 6** pp. 343-349 (June 1980).

5. H. Gouraud, "Contiunuous Shading of Curved Surfaces," *IEEE Transactions on Computers* **c-20, 6** pp. 623-629 (June 1971).

6. B. T. Phong, "Illumination for Computer Generated Pictures," *Graphics and Image Processing* **18, 6** pp. 311-317 (June 1975).

7. H. Weghorst, G. Hooper, and D. P. Greenberg, "Improved Computational Methods for Ray Tracing," *ACM Transactions on Graphics* **3, 1** pp. 52-69 (January 1984).

8. A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications* **6, 4** pp. 16-26 (April 1986).

9. B. A. Navsky, *Polygon Clipping and Star Decomposition for Rendering Solid Models,* Rensselaer Polytechnic Institute, Troy (May 1985). MS thesis and technical report TR-85007

INTERNATIONAL DEVELOPMENT ARENA

Computer Developments in Japan

TRACK CHAIR: Prof. Ryoichi Mori
University of Tsukuba

TRACK SECRETARIAT: Mr. Kenji Naemura
NTT Electrical Communications Laboratories

GUARDED HORN CLAUSES AND EXPERIENCES WITH PARALLEL LOGIC PROGRAMMING

Jiro Tanaka, Kazunori Ueda, Toshihiko Miyazaki, Akikazu Takeuchi,
Yuji Matsumoto, and Koichi Furukawa

ICOT Research Center
Institute for New Generation Computer Technology
1–4–28, Mita, Minato-ku, Tokyo 108 Japan

This paper tries to overview the various activities of ICOT related to Guarded Horn Clauses (GHC). We describe a new parallel logic language GHC, first proposed by Ueda[1,2]. The main features of this language are its simplicity and ease of implementation. The implementation of GHC and programming efforts in the language are also described. We also summarize the current status of Kernel Language Version 1 (KL1). KL1 is the language system for the PIM hardware[3]. The overall structure of KL1 and the work on its distributed implementation are described.

1. INTRODUCTION

The final goal of the Fifth Generation Computer Project is the development of a logic-based high-speed parallel computing system. Our objective is the design and development of a "logic programming language" which allows parallel execution. Kowalski[4] has pointed out that a set of Horn clauses also allows "parallel" execution as well as sequential execution. A lot of work has been devoted to the Or-parallel execution of Prolog programs. We consider that this approach may be useful for uncontrolled all-solution-search problems. However, it may be inadequate for solving parallel programs in general. What we want is a more expressive, general-purpose parallel programming language which includes important concepts such as processes, communication and synchronization.

2. PARALLEL LOGIC PROGRAMMING LANGUAGE

2.1 Design goals

The design requirements for our parallel logic programming language can be summarized as follows[1]:

- Parallelism. It must express parallelism "by nature." Sequential languages with parallel constructs added are inadequate. It should have as little sequentiality as possible in order to preserve parallelism inherent in a Horn-clause program.

- Expressiveness. It must be an expressive, general-purpose parallel programming language. In particular, it must be able to express important concepts in parallel programming such as processes, communication, and synchronization.

- Simplicity. We do not have much experience with parallel programming. Therefore, it must be a "simple" language and we must establish a foundation of parallel programming first.

- Efficiency. There are various typical parallel problems to be described in the language. It is important that we can execute such programs as efficiently as comparable programs written in existing parallel programming languages.

2.2 GHC

The languages we are interested in are the parallel logic languages such as Parlog[5] and Concurrent Prolog[6]. These languages seem to come closest to satisfying the above requirements. Although there are differences, the basic computation mechanisms of these languages are quite similar: Horn clauses with guards are used for defining predicates, goals are executed in parallel, and they have some synchronization mechanisms between goals.

In this section, we describe the new parallel logic language GHC, which was proposed by Ueda[1,2]. It inherits many features from Parlog and Concurrent Prolog. What is most characteristic about this language is that the guard is the only syntactic construct added to Horn clauses. In GHC, synchronization is realized by the suspension mechanism of guards.

A GHC program is a set of guarded Horn clauses of the following form:

$$H :- G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_k.$$

The operator | is called a commitment operator. The part of a clause before | is called a guard, and the part after | is called a body. Note that a clause head is included in a guard.

CH2345-7/86/0000/0948$01.00 © 1986 IEEE

A goal clause has the following form:

$$:-\ B_1, ..., B_n. \qquad (n > 0)$$

This can be regarded as a guarded Horn clause with an empty guard.

The semantics of GHC is quite simple. Informally, to execute a program is to refute a given goal clause by means of input resolution using the clauses constituting the program. This can be done in a fully parallel manner under the following rules.

(a) Unification invoked in the guard of a clause cannot instantiate the caller of that clause. If this happens, unification suspends until that caller is instantiated by some other goal. This provides the basic synchronization mechanism of GHC.

(b) When the guard of a clause succeeds, it is first checked that no other clause has been selected for the same goal. If confirmed, that clause is "committed" exclusively for subsequent execution of the goal.

(c) Unification invoked in the body of a clause cannot instantiate the guard of that clause until that clause is committed.

It must be stressed that under the rules stated above, anything can be done in parallel: Conjunctive goals can be executed in parallel; candidate clauses for a goal can be tested in parallel; head unification can be done in parallel; head unification and the execution of guard goals can be done in parallel. However, it is even more important to stress the fact that we can also execute a set of tasks in an arbitrary order as long as it does not change the meaning of the program.

2.3 Program examples

In this section we give program examples showing how GHC programs are described.

Binary Merge

```
merge([A|Xs], Ys,     Zs) :- true |
          Zs=[A|Zs1], merge(Xs, Ys, Zs1).
merge(Xs,     [A|Ys], Zs) :- true |
          Zs=[A|Zs1], merge(Xs, Ys, Zs1).
merge([],     Ys,     Zs) :- true |
          Zs=Ys.
merge(Xs,     [],     Zs) :- true |
          Zs=Xs.
```

The goal merge(Xs, Ys, Zs) merges two streams Xs and Ys (implemented as lists) into one stream Zs. This is an example of a nondeterministic program. Note that no binding can be exported from the guard; the binding to Zs must be done in the body.

Generating Primes

```
primes(Max, Ps) :- true |
          gen(2, Max, Ns), sift(Ns, Ps).
gen(N, Max, Ns) :- N =< Max |
          Ns=[N|Ns1], N1 := N+1,
          gen(N1, Max, Ns1).
gen(N, Max, Ns) :- N >  Max | Ns=[].
sift([P|Xs], Zs) :- true |
          Zs=[P|Zs1], filter(P, Xs, Ys),
          sift(Ys, Zs1).
sift([],     Zs) :- true | Zs=[].
filter(P, [X|Xs], Ys) :- X mod P=:=0 |
          filter(P, Xs, Ys).
filter(P, [X|Xs], Ys) :- X mod P=\=0 |
          Ys=[X|Ys1], filter(P, Xs, Ys1).
filter(P, [],     Ys) :- true | Ys=[].
```

The goal primes(Max, Ps) returns through Ps a stream of primes up to Max. The stream of primes is generated from the stream of integers by filtering out the multiples of primes. For each prime P, a filter goal filter(P, Xs, Ys) is generated which filters out the multiples of P from the stream Xs, yielding Ys.

3. SEQUENTIAL IMPLEMENTATION

Even though our final target of GHC is parallel implementation, sequential implementation of GHC is nevertheless important. Our experience with Concurrent Prolog shows that interpretive execution of this type of language can be very slow[7]. For this reason, we have concentrated our efforts on implementing GHC "compilers."

GHC and Prolog have lots of similarities, so translation of the former to the latter is much simpler than direct compilation to a machine language. We decided to compile GHC programs to DEC-10 Prolog programs since we already have a DEC-10 Prolog compiler which translates a Prolog program to the machine language. The GHC source programs are ultimately compiled into machine code.

The basic technique for compiling a parallel logic programming language to Prolog is given in Ueda's paper[8]. We have already developed two GHC compilers using that technique[9], one developed by Miyazaki and the other developed by Ueda. These compilers have the following features in common.

(1) They both use techniques similar to Concurrent Prolog Compiler[8] which compiles a Concurrent Prolog program to DEC-10 Prolog.

(2) Both systems evaluate the guards of candidate clauses sequentially. Or-parallel execution of candidate clauses is not performed. The body of a clause is evaluated only after that clause is selected.

(3) Since they are both implemented on top of Prolog, there exists an interface to Prolog. The goal prolog(X) calls X as a Prolog goal.

The differences between the two can be summarized as follows:

(a) Ueda's Compiler[8]. This compiler works on the subset of GHC containing no user-defined goals in guards of clauses. We call this subset "Flat GHC" (FGHC) since there are no nesting guards. In this case, run time check of variables is not needed. This compiler does not distinguish "failure" from "suspension," i.e., "failures" are treated as "suspensions." These simplifications make the execution speed of this compiler very fast (approximately 11KLIPS for "append" program on DEC-2065).

(b) Miyazaki's Compiler. This compiler works on the full set of GHC, i.e., we can call user-defined goals from the guard of a clause. In this case, we need run time check of variables. We have realized this by using the "address-comparison method." Execution speed is slower than Ueda's compiler. This compiler distinguishes between "failure" and "suspension."

Although compilation to Prolog provides an offhand way to execute GHC programs, we also need more direct implementation for more realistic applications. Compilation of FGHC programs into the VAX-11/780 machine code has also been attempted. We have started to build a compiler of FGHC into the VAX machine code. Mock-up object code has been tested and timed. The full compiler system is being written in FGHC, C and VAX-11/780 assembly languages.

4. GHC PROGRAMMING

At this stage, it is very important for us to gain experience in parallel programming. Various application efforts based on GHC are currently under way. These works can be summarized as follows:

(1) All-solution-search transformation [10]

We are developing a program transformation technique to transform Horn clause programs into deterministic GHC programs. It can be viewed as a way to provide GHC with the all-solution-search ability. This technique is applicable to a non-trivial class of programs and the transformed program can be executed quite efficiently. The technique is also important in that it exploits And-parallelism for parallel search as well as Codish's work[11].

(2) Process fusion[12]

The language features of GHC encourages the stream-oriented programming where computation is expressed as processes communicating with one another. In general, each process is designed to compute a relatively simple and small task. However, the resulting programs, if naively implemented, may generate too many small processes which may cause inefficiency because of excessive interprocess communication. Process fusion aims to reduce the number of processes by fusing communicating processes. This is analogous to loop fusion in procedural languages. Development of the program transformation method is currently proceeding based on the fold/unfold method of Burstall and Darlington [13].

(3) Parallel parser[14]

The aim of this research is to develop a parsing system which is naturally implemented in a parallel logic programming language. In our framework, a grammar rule written in DCG [15] is compiled into a program of a parallel logic programming language such as Guarded Horn Clauses or Parlog. Words in a given sentence are defined as processes, consecutive pairs of which are connected by streams. Completed subtrees are also represented as processes and partially constructed parse trees are expressed as data structures produced and put into streams by such processes. Parsing proceeds as dynamic construction of such processes and data. Although the construction of the parse trees proceeds from bottom to top, we also utilize top-down predictions implemented as filtering processes. these processes filter out subtrees that are inconsistent with the predicates. The system is most appropriately compared with Martin Kay's Chart Parsing, in that inactive arcs correspond to processes and active arcs are represented as data passed through streams. The important feature of our method is that the grammar rules and the dictionary are completely compiled into a program in the target parallel logic programming language and the system does not need an additional program to interpret the grammar and the dictionary. The derived program has neither side-effects nor duplicate computation.

(4) Algorithmic debugging[16]

Another research effort is devoted to the development of a debugger for GHC. It has been said that debugging parallel programs is a very hard task compared with debugging sequential programs. The reasons is that 1) conceptually several computations are executed in parallel, 2) these computations may interact with each other, and 3) there are new kinds of bugs such as deadlocks. Usually a program is debugged by examining its execution trace. The execution trace of a parallel program is, however, messy since traces of several computations are interleaved. Even if the execution trace is separated into several traces using, say windows, that is not sufficient for debugging. In general, we must distinguish between debugging and understanding program behavior. In the case of debugging, what is required is to find the location of the bug. Monitoring of program behavior will help finding a bug, but it forces a programmer to understand the program behavior. It would be better if a programmer could debug a program only with more abstract knowledge such as input and output specifications of component modules. We have defined abstract semantics of GHC programs and are developing an algorithmic debugger for GHC. The debugger is based on Shapiro's idea of algorithmic debugging[17] and reduces the number of queries based on the "divide and query" strategy. Lloyd and Takeuchi formally examined the properties of the debugger[18]. For simplicity the current version of our algorithmic debugger only deals with the body of GHC clauses

(5) Propositional temporal logic prover[19]

Temporal Logic is an extension of the first order logic including the notion of time. It deals with logical descriptions and reasoning on time. A propositional temporal logic prover based on the omega-graph refutation procedure has been developed in GHC.

Omega-graph refutation is a procedure to decide the validity of a temporal formula. It negates a given formula, computes the initial node formula of the negated formula, constructs the corresponding omega-graph, and checks the existence of a loop called an "omega-loop." If the loop is found, then the given formula is proved to be invalid.

In our implementation, an omega-graph is constructed incrementally from the initial node formula by successive node expansions. It is represented by the network of communicating processes which spawn brother processes as the graph is expanded. New methods have also been implemented for the detection of an omega-loop where the network of processes finds loops by sending and receiving messages.

We have implemented an appropriate mechanism which checks whether access to the unconstructed part of the network is attempted and forces the process to suspend. Therefore, an omega-loop detection can be performed in parallel with omega-graph construction. It has helped to reduce execution time drastically.

5. KERNEL LANGUAGE VERSION 1

In this section we summarize the current status of Kernel Language Version 1 (KL1). KL1 is originally the language for the PIM hardware. The prototype development of PIM-D (Parallel Inference Machine based on Dataflow mechanism) and PIM-R (Parallel Inference Machine based on Reduction mechanism) has been done[3]. KL1 is expected to operate as the interface between PIM hardware and software to be developed for the parallel high-speed logic-based system.

The first conceptual specification of KL1 was made in 1984[20]. We have stated that KL1 should be based on Concurrent Prolog[6] adding set-abstraction, meta-inference and module facilities.

Based on this conceptual specification, lots of energy has been put into the detailed specifications of KL1 and implementation of Concurrent Prolog [7]. This work clarified the problems existing in Concurrent Prolog regarding semantics and parallel execution . It forced us to revise the conceptual specification of KL1 [21].

After extensive investigations, we have decided to adopt Flat GHC (FGHC) as the core of KL1. Discussion on the detailed specifications of KL1 have also made it clear that we need to separate the language into several layers. Thus, KL1 now consists of three layers. These are shown in Figure 1.

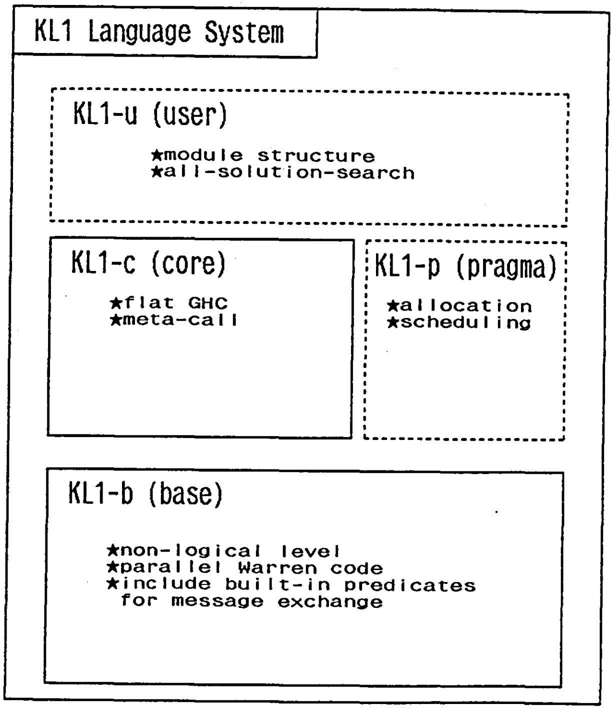


Figure 1 KL1 Language system

KL1-c is the core language of KL1. We assume KL1-c to be FGHC with meta-call predicates.

KL1-p is the pragma language which specifies how the program should be executed in a distributed/parallel environment. KL1-p is not a language as it stands. It is attached to KL1-c to specify process allocation and scheduling information.

KL1-u is the user language to be built on KL1-c and KL1-p. It has the module structure and all-solution-search predicates. KL1-u is in the design stage right now.

KL1-b is the machine language which hardware/ firmware will directly support. This level is not a logical level and it looks like the parallel version of Warren's abstract machine instruction set[22].

6. DISTRIBUTED IMPLEMENTATION

Much research on PIM hardware has been carried out from the "architectural" view point. However, we have noticed that there still exist lots of problems to be solved at the "software" or "firmware" level, such as (1) how to boot the system, (2) how to deliver object programs to each processing element (PE), (3) how to handle input/output and interrupts, and (4) how to balance the load between PEs.

To address these software problems arising from distributed environment, we have decided to start a "Multi-PSI" project. Multi-PSI hardware is simply built up by connecting 6 - 16 Personal Sequential Inference (PSI) machines[23] by high speed grid-network.

6.1 The multi-PE model

We take the multi-PE system where dozens of PEs are grid-network connected as our starting point. There are many design choices. Lots of intensive discussions on these issues have been held inside ICOT. The decisions we have made so far are as follows:

(1) PEs must be connected in a way which allows easy expansion of their number. Our system needs to work even if we have hundreds of PEs.

(2) Each Processing Element (PE) has its local memory. It has no shared memory nor global address space. In our system, PEs communicate by message exchange.

(3) We have employed And-parallelism, since we believe that it plays a more basic role than Or-parallelism in the distributed environment. Each PE executes a program independently or cooperatively.

The conceptual figure of the multi-PE model we assume is shown in Figure 2.

This multi-PE model has the following features.

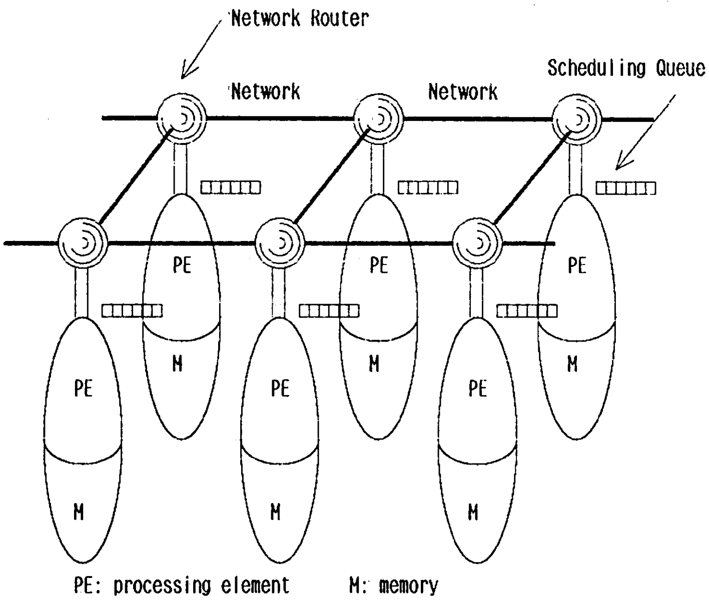


Figure 2 Multi-PE Model

(1) The program, i.e., the collection of clauses, is loaded to every PE at the beginning. A more realistic assumption such as "lazy" fetching of program code may be needed at a later stage.

(2) First, a goal is put into one PE. This automatically starts the computation. Each PE executes the goals sent from other PEs besides processing local goals. When there are no goals left to be processed on any PE, computation terminates.

(3) Each PE has one scheduling queue. Each PE repeats dequeuing a goal from the scheduling queue and reducing it to the resulting goals. These goals are enqueued to the scheduling queue or sent to an other PE.

(4) Since each PE has an independent address space, unification of two variables existing in different PEs makes it necessary to span inter-PE reference chains. Each PE has a variable management table for that purpose.

(5) Unification in one PE may generate various messages to be sent to other PEs. Examples of such messages are "get_value," "unify," and "unify_channels."

6.2 Multi-PE simulator

We have implemented a software simulator of the multi-PE system [24]. The simulator is based on the work by Miyazaki and Murakami [25]. Our system simulates the execution of pre-processed FGHC programs in a multi-PE environment. It is written in Prolog and consists of processes and a network manager as shown in Figure 3.

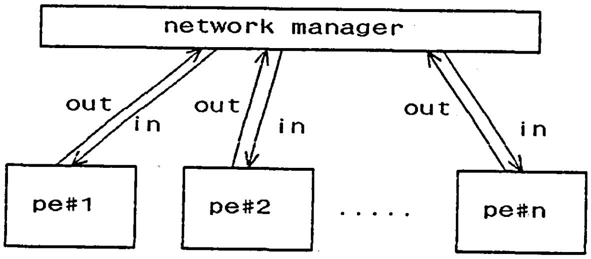


Figure 3 Multi-PE simulator

Each PE checks whether there is a message from its input channel. If so, messages are added to the scheduling queue. Then it executes the first goals in the queue. If there is a message to be sent, the message is put to the output channel. The network manager takes care of message exchange between PEs. Messages inside output channels include the sender's address. The network manager delivers the message to the specified input channel.

Since our system does not have the global address space, sending a goal which includes variables to other PEs requires a somewhat complicated mechanism. In our implementation, each PE has a variable management table to keep track of inter-PE references.

6.3 Implementation on PSI[26]

Implementation of FGHC on one PSI machine was a more realistic starting point to the "Multi-PSI" project. The PSI is a personal workstation and its design concept is very similar to a LISP machine. Conventional techniques have been adopted and ESP[27], the object-oriented extension of Prolog, is firmware supported.

The compiler which translates an FGHC program to Warren-like abstract machine instruction sequences[22] and the emulator of that abstract machine instruction set have already been implemented. Our Warren-like instruction set also includes the primitives for distributed execution. The emulator was written in ESP and was realized using "heap" on PSI machine.

The current version only executes a program on one PSI machine. The execution speed is approximately 0.9 KLIPS on a naive reverse program. We are now devoting a lots of energy to the extension of this work to the Multi-PSI system. ICOT has already finished the design of the connecting hardware. The actual hardware of Multi-PSI Version 1 will be completed by the end of April 1986.

7. SUMMARY

We described the various activities at ICOT related to Guarded Horn Clauses (GHC). The language features of GHC, its implementation efforts and its applications were described. We also summarized the current status of KL1 and efforts aimed at its distributed implementation. These are on-going activities and must be extended in various directions.

8. ACKNOWLEDGMENTS

These research activities have been carried out as a part of the Fifth Generation Computer Project. We would like to thank our colleagues, Kazuo Taki and Takashi Chikayama, for their useful comments and suggestions. We also would like to thank Kazuko Takahashi and Tadashi Kanamori at the Central Research Laboratory, Mitsubishi Electric Corp., for their GHC programming efforts. Thanks go also to Kazuhiro Fuchi, the Director of ICOT for giving us the opportunity to pursue this research.

REFERENCES

[1] Ueda, K.: Guarded Horn Clauses. ICOT Technical Report TR-103, ICOT, 1985.

[2] Ueda, K.: Guarded Horn Clauses. Doctoral Thesis, Faculty of Engineering, University of Tokyo, March 1986.

[3] Murakami, K. et al.: Research on Parallel Machine Architecture for F.G.C.S.. Computer, vol.18, No.6, June 1985.

[4] Kowalski, R.: Predicate Logic as Programming Language. In Proc. IFIP '74, North-Holland, Amsterdam, London, 1974, pp.569-574.

[5] Clark, K., Gregory, S.: PARLOG: Parallel Programming in Logic. Research Report DOC 84/4, Department of Computing, Imperial College of Science and Technology, Revised June 1985.

[6] Shapiro, E.: A Subset of Concurrent Prolog and its Interpreter. ICOT Technical Report TR-003, ICOT, 1983.

[7] Miyazaki, T. et al.: A Sequential Implementation of Concurrent Prolog Based on Shallow Binding Scheme. Proceedings of 1985 Symposium on Logic Programming pp. 110-118.

[8] Ueda, K. and Chikayama, T.: Concurrent Prolog Compiler on Top of Prolog. Proc. of 1985 Symposium on Logic Programming, pp.119-126, 1985.

[9] Furukawa K. et al.: Kernel Language Version 1, explanation materials, ICOT, 1985, in Japanese.

[10] Ueda, K.: Making Exhaustive Search Program Deterministic. ICOT Technical Report TR-145, ICOT, 1985. Also to be presented at the Third Int. Logic Programming Conf., London (1986).

[11] Codish, M.: Compiling OR-parallelism into AND-paralelism, M.Sc Thesis, Weizmann Institute of Science, December 1985.

[12] Furukawa K., Ueda K.: GHC Process Fusion by Program Transformation. In Proc. Second National Conf. of Japan Society of Software Science and Technology, 1985, pp. 89-92.

[13] Burstall, R. and Darlington, J.: A Transformation System for Developing Recursive Programs, JACM Vol.24, No.1, pp.44-67.

[14] Matsumoto, Y.: A Parallel Parsing System for Natural Language Analysis, to be presented at the Third Int. Logic Programming Conf., London, 1986.

[15] Pereira, F.C.N. and Warren, D.H.D.: Definite Clause Grammars for Language Analysis - A survey of the Formalism and a Comparison with Augmented Transition Networks, Artificial Intelligence, 13, pp. 231-278, 1980.

[16] Takeuchi, A.: Algorithmic debugging of GHC programs and its implementation in GHC, unpublished draft, 1986.

[17] Shapiro, E.: Algorithmic Program Debugging, MIT Press, Cambridge, Mass, 1982.

[18] Lloyd, J. and Takeuchi, A.: A Framework for Debugging GHC. To appear ICOT TR, 1986.

[19] Takahashi,K. and Kanamori,T.: On Parallel Programming Methodology in GHC. ICOT Technical Report, to appear.

[20] Furukawa K. et al.: The Conceptual Specification of the Kernel Language Version 1. Technical Report TR-054, ICOT, 1984.

[21] Ueda, K.: Concurrent Prolog Re-Examined. ICOT Technical Report TR-102, ICOT, 1985.

[22] Warren, D. H.: An Abstract Prolog Instruction Set. Tech. Report 309, Artificial Intelligence Center, SRI International, CA, 1983.

[23] Taki, K. et al.: Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). Proc. International Conference on Fifth Generation Computer Systems 1984, ICOT, pp.398-409.

[24] Tanaka, J. et al.: Distributed Implementation of FGHC – Toward the realization of Multi-PSI system –. Unpublished draft, 1986.

[25] Murakami, K.: The study of unifier implementation in multi-processor environment. Multi-SIM study group internal document, ICOT, 1985, in Japanese.

[26] Miyazaki, T. and Taki, K.: The implementation method of Flat GHC on Multi-PSI system. The Logic Programming Conference '86, ICOT, June 1986, pp.83-92, in Japanese.

[27] Chikayama, T. : ESP Reference Manual. ICOT Technical Report TR-044, ICOT, 1984.

"KABU-WAKE" PARALLEL INFERENCE MECHANISM AND ITS EVALUATION

Hideo Masuzawa, Kouichi Kumon, Akihiro Itashiki,
Ken Satoh, and Yukio Sohma

FUJITSU LIMITED
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, JAPAN

Abstract

We have proposed a new parallel inference mechanism called the KABU-WAKE method, and have built an experimental OR parallel inference system based on this method. This paper evaluates use of the KABU-WAKE mechanism in practical applications. Tests on our experimental system using a restricted bottom-up parser show that the KABU-WAKE method is very effective for large problems that contain a lot of parallelism. This paper also describes two difficulties of the KABU-WAKE method made evident through analysis of the status of each processor element during the processing of a problem with a low performance improvement ratio. We also estimated the performance improvement ratio expected with 100 processor elements. This research was sponsored by MITI as a part of the FGCS project.

1. Introduction

AI research is now flourishing. In this field, a problem with one of the important features can be solved by traversing a search tree using trial and error methods. A logic program can also be executed using parallel processing, such as OR parallel, AND parallel, or AND-STREAM parallel processing. OR parallel processing is suitable for nondeterministic algorithms and we therefore think it is well suited to a search-type problem described above.

We proposed a new parallel inference mechanism, called the KABU-WAKE method. We built an experimental OR parallel inference system implementing this method [1], [2]. We ran a simple game problem, the N-Queen problem, on the experimental system to evaluate our method [3], [4]. At present, we are evaluating our method for two practical applications, both of which are parser programs with restricted grammars. We have already presented an evaluation of one of the applications-- a top-down parser for the Japanese language. The algorithm produced a top-down parsing tree from a sentence [6]. This paper evaluates the KABU-WAKE method using a bottom-up parser for the Japanese language. The algorithm makes a parsing tree from terminal symbols at the bottom [5]. By analyzing the status of each Processor Element (PE) during processing, we will explain 1) why the performance improvement ratio (how much performance improves over performance with 1 PE) with fewer than 12 PEs was high using a large problem, 2) why the performance improvement ratio was low using a problem that cannot achieve a high performance improvement ratio with 12 PEs, and 3) estimate the performance improvement ratio expected with 100 PEs using a problem that achieved a high performance improvement ratio with fewer than 12 PEs. Chapter 2 discusses problems in parallel inference processing, and explains the features and principles of operation of the KABU-WAKE method. Chapter 3 outlines our experimental system. Chapter 4 discusses the results of executing the bottom-up parser program on our experimental system.

2. KABU-WAKE method

2.1 Problems in Parallel Inference Processing

Our original question is how to make an OR parallel processing inference system so that performance is proportional to the number of PEs. The following problems need resolution:

(1) Problems within each PE
- i. Reducing overhead for switching and scheduling tasks
- ii. Reducing the time for creating and dividing tasks

(2) Problems between PEs
- i. Achieving dynamic load balancing, because we do not know how much

CH2345-7/86/0000/0955$01.00 © 1986 IEEE

processing a program requires before it is executed.

ii. Reducing the number of subtask transfers

iii. Reducing the amount of time for each subtask transfer

The KABU-WAKE method solves most of these problems for certain applications, as we discuss in the next section.

2.2 Principles of Operation

This section discuss the operation principles of the KABU-WAKE method. Figûre 1-a is an example of a knowledge base written in Prolog. We have omitted the arguments of the predicates to simplify explanation. The arrows indicate the normal flow in a conventional sequential inference. Figure 1-b is a snapshot of processing using the KABU-WAKE method. The bold lines indicate the processing in PE 0, corresponding to the processing indicated by the arrows in Figure 1-a. If a request comes from PE 1, PE 0 divides its task at the node closest to the root of the search tree and transfers a portion to PE 1. If a request then comes from PE 2, PE 0 again divídes a part of the task in the same way. This method enables several PEs to solve one problem.

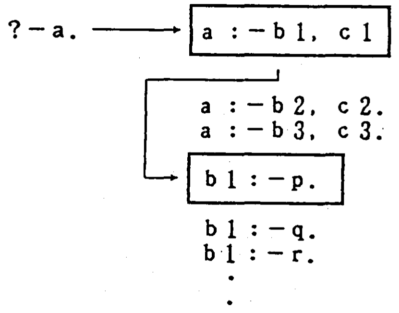


Figure 1-a

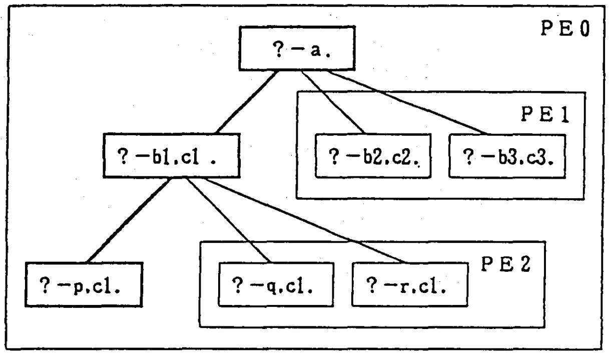


Figure 1-b

Figure 1 Basic KABU-WAKE mechanism

2.3 Features

The KABU-WAKE method has two principal features:

-- Each PE uses the same search strategy as is used by a conventional sequential processor to traverse a search tree, namely, a depth-first search.

-- A PE divides its task and transfers a portion of it only when requested by another PE.

These features have the following advantages:

(1) Low overhead in each PE

i. Each PE processes only one task.

--> There is no overhead for switching and scheduling tasks as in a multiple task environment.

ii. A task is divided at the node closest to the root of the search tree.

--> We think that each PE should divide a task into the largest possible subtasks.

iii. Each PE uses conventional sequential processing.

--> There is almost no overhead for parallel processing during execution in each PE [4]. Also, this method can use the high-speed techniques developed for conventional sequential processing, such as in compilers.

(2) Low overhead between PEs

i. A busy PE divides its task and transfers a portion only when requested by another PE.

--> When all the PEs are busy, there is no communication.

ii. A distributed processing method is used in which only idle PEs receive tasks.

--> Effective dynamic load balancing is possible.

(3) Easy implementation

i. The number of tasks does not exceed the number of PEs.

--> No mechanism is required to temporarily hold extra tasks for PEs.

3. Experimental System

3.1 System configuration

Figure 2 shows the system hardware configuration. The experimental system consists of 16 PEs (one for the man-machine interface) and two networks. These networks are used for different purposes, and were designed specifically for our method.

(1) PE

A PE performs inference processing. It gives part of it's task to another PE if a request comes during processing. A KABU-WAKE interpreter is installed in each PE. This interpreter is a conventional sequential interpreter with added control mechanisms [2],[4] to implement the KABU-WAKE method. Each PE has a copy of the entire database.

(2) CONTROL NETWORK

The control network is a communication route for requesting tasks. This network connects all PEs in a ring via control network adapters (CNAs). The clock frequency is 2 MHz. A packet, which indicates whether each PE is busy or idle, goes around the network [6] in 8-microsecond cycles.

(3) DATA NETWORK

The data network is a communication route for transferring tasks. It is a two-stage network made up of 4 x 4 switches. The communication method is DMA transfer of 8 bits in parallel. The transmission rate of the network hardware without overhead is about 400 KB/s, but it drops to 50 KB/s with the addition of the communication software overhead [6].

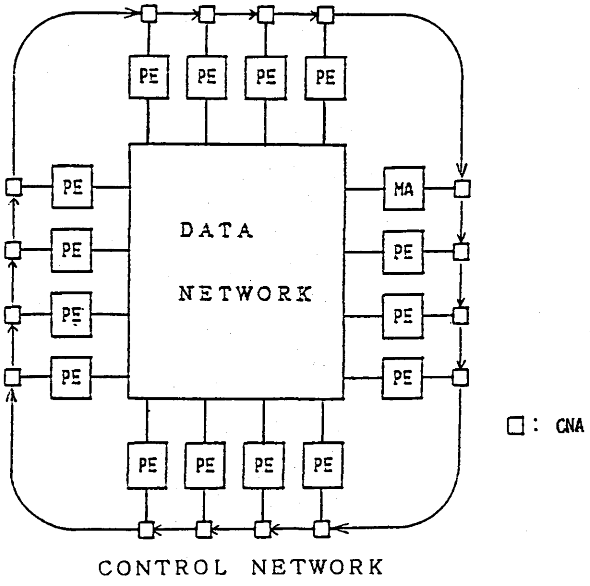


Figure 2 System configuration of experimental machine

3.2 Implementation

The following items need be implemented in our experimental system.

(1) Unbinding of variables

When a search tree is split, the status of the variables in the split tree must be as if all the processes located below the split portion of the search tree had failed and backtracked, i.e., unbinding. To speed up this unbinding operation, we reserved an area for each variable in which was stored the time at which it was bounded. Each time a subgoal is reached, a new level number, corresponding to the depth of subgoal, is assigned. For variables bound during the processing of that subgoal, the same level number as that of the subgoal is flagged. When a tree is split, we can determine whether binding should be released by comparing the level of the subgoal to be split with the variable's level number. In this method, the time required to release variables for splitting is proportional to the number of variables in the subgoal to be split. Using a trailing stack is another alternative, but we think the processing would take more time.

(2) Use of rule numbers

When a tree is split and transferred, it is transferred in the form of a subgoal. However, since some of the definitions for the subgoal are already being processed, the other PEs must start from subsequent definitions. We, therefore, we adopted a method in which a special predicate, called a rule number, is prepared and attached to the subgoal before transfer to indicate where to start execution.

(3) Control of request

Some tasks cannot be split when a request is received from another PE. If this is often the case, PE performance deteriorates. To handle this situation, we used a request reception flag to indicate whether a task can be split. This enables a PE to concentrate on a task without being disturbed by other PEs.

4. Evaluation

Let us now consider the results obtained by running the bottom-up parser on our experimental system. We will first present the results, then discuss them.

4.1 Results

4.1.1 Test programs

We entered three sentences into the bottom-up parser (44rules, 103facts). The characteristics of the three bottom-up parser programs corresponding to the three sentences entered are shown in Table 1. Here, the bottom-up parser is a Japanese-language parser; the algorithm makes a parsing tree from terminal symbols at the bottom. We define a execution time with 1 PE as the amount of processing time. The ratio of the amount of processing time in Sentence 1 to that in Sentence 2 is 1:8 and of Sentence 1 to Sentence 3 is 1:60. The amount of processing time required in Sentence 1 is about 1.3 times as much as in 7-Queen.

Table 1 Logical characteristics of bottom-up parser programs

ITEM PROGRAM	Average number of parallel degrees *1	Number of inference levels
Sentence 1	45	1294
Sentence 2	236	2081
Sentence 3	1427	2388

Notes

*1 : Average number of parallel degrees = ((total number of inferences) ⁄ (number of inference levels))

*2 : The three sentences entered are as blow :

Sentence 1

“wakai otoko wa kouen de akai boushi no onna no ko ga hon wo yomu nowo mita ”

(meaning) :

A young man saw a girl with a red hat reading a book at a park.

Sentence 2

“aoi fuku no wakai otoko wa kouen no ki no benchi de akai boushi no onna no ko ga atarashii ryouri no hon wo yomu nowo mita ”

(meaning) :

A man wearing blue clothes saw a girl with a red hat reading a new cookbook on a wood bench at a park.

Sentence 3

“akai fuku no wakai otoko wa watashi no machi no kouen no ki no benchi de akai boushi no onna no ko ga atarashii ryouri no hon wo yomu nowo mita”

(meaning) :

A young man wearing red clothes saw a girl with a red hat reading a new cookbook on a wood bench at a park in my town.

4.1.2 Performance improvement ratio

Figure 3 shows the relationship between the number of PEs and the performance improvement ratio due to parallel inference. It shows that in a large problem, such as Sentence 3, the performance improvement ratio is almost proportional to the number of PEs. As the amount of processing decreases (Sentence 3 -> Sentence 2 -> Sentence 1), the performance improvement ratio decreases. The ratio of Sentence 1 increases only until the number of PEs reaches 8, then it levels off.

4.2 Discussion

This section discusses the following three items.

(1) Performance improvement ratio of Sentence 3

Figure 3 shows that the performance improvement ratio of Sentence 3 is higher than that of Sentence 1 and 2. We will discuss the cause.

(2) Performance improvement ratio of Sentence 1

Figure 3 shows that the performance improvement ratio of Sentence 1 is low. We will discuss the cause, and describe several difficulties related to the KABU-WAKE method made evident through our analysis.

(3) Estimated performance improvement ratio of Sentence 3

One of the AI applications is a search problem; we think this type of problem requires a great deal of processing. We regard Sentence 3 as such a search-type problem and estimate the performance improvement ratio with 100 PEs.

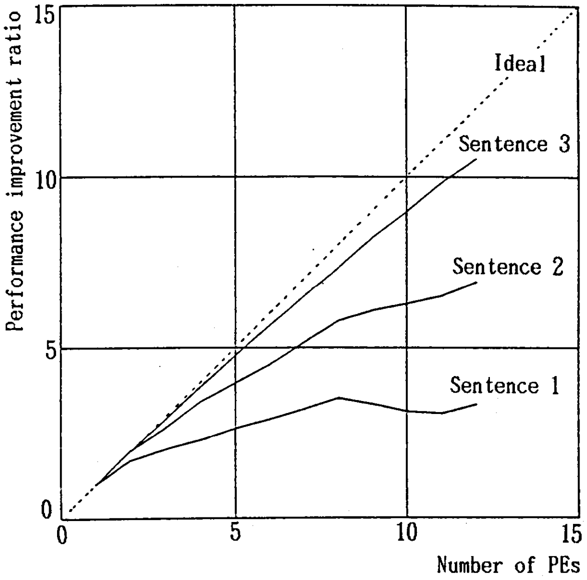


Figure 3 Performance improvement ratio due to parallel inference

4.2.1 Performance improvement ratio of Sentence 3

Figure 4 graphs the relationship

between the processing time (with 1 PEs) and the number of subtask transfers (with 12 PEs). In Figure 4, the xcoordinate is a logarithm of the processing time. Figure 4 indicates that the number of subtask transfers is almost proportional to the logarithm of the processing time, but not to the processing time itself. This means that, as the prcessing time increases exponentially, the number of subtask transfers increases linearly. That is, for similar programs, the longer the processing time, the less communication overhead. This is why the performance improvement ratio of Sentence 3 is higher than that of Sentence 1 and 2.

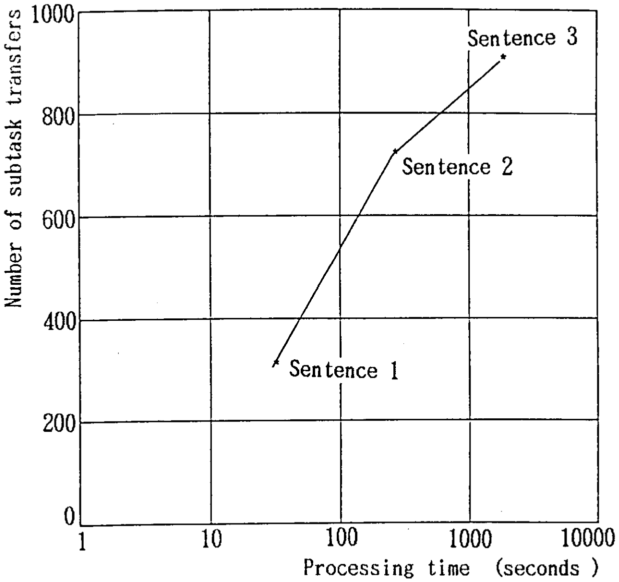


Figure 4 Relationship between processing time and number of subtask transfers (PEs =12)

4.2.2 Performance improvement ratio of Sentence 1

Table 1 shows that the average number of logical parallel degrees in processing by the KABU-WAKE method is 45. However, the performance improvement ratio of Sentence 1 does not reach 4, even when Sentence 1 is processed with 12 PEs.

<1> Results

The experimental results show that when the number of PEs is 12, the rate of processing items (busy, KABU-WAKE, idle) are as follows:

busy time: 30.6%,
KABU-WAKE time: 43%,
idle time: 26.4%

Busy time includes all of the inference processings. KABU-WAKE time includes all of the KABU-WAKE processings (explained later).

The ratio of processing time shows that KABU-WAKE time and idle time accounted for more than 70% of all the time used to process Sentence 1. These two factors (we call them overhead for parallel execution) prevent performance improvement.

<2> Analysis

To discover why the overhead for parallel execution such as both KABU-WAKE time and idle time are large, we discuss some factors which may cause the overhead.

There are two factors which may cause the overhead:

Factor 1: KABU-WAKE processing time per transaction
The time depends on what feature of KABU-WAKE processing is being used.

Factor 2: number of KABU-WAKE processings
The number depends on the nature of parallelism in the search tree of Sentence 1.

(1) Features of KABU-WAKE processing

Figure 5 details the KABU-WAKE processing time in the execution of Sentence 1. The KABU-WAKE processing time is 106 ms: 49 ms for Sender to transfer a subtask and 57 ms for Receiver to receive the subtask. While Sender does Split processing in the KABU-WAKE processing, the Receiver is idle (Wait in Figure 5), and the Wait time is 33 ms. This minimum time is needed for each KABU-WAKE processing. It takes 2 ms for the KABU-WAKE interpreter to process a inference, so the interpreter can process about 53 inferences during KABU-WAKE processing time and about 16 inferences during Wait time. The KABU-WAKE processing time per transaction was long.

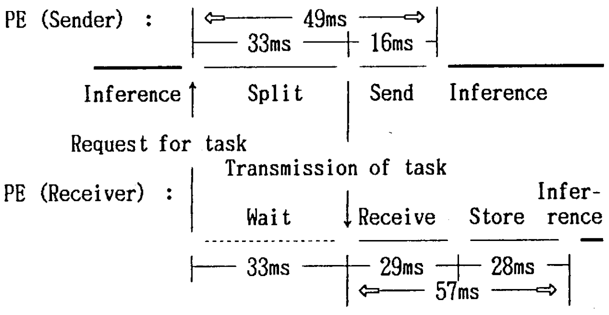
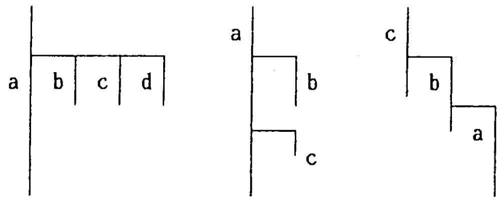


Figure 5 Details of KABU-WAKE processing time

(2) Nature of parallelism in the search tree of Sentence 1.

Almost all the parallelisms are of the type shown in Figure 6, and almost all branches (b, c, and d in Figure 6) in the portions with such parallelisms have a number of inferences smaller than 60.



Almost all branches b, c, or d has a number of inferences smaller than 60.

Figure 6 Characteristics of a sabtask that to can be processed in parallel by a bottom-up parser

In the execution of Sentence 1 with 12 PEs, the transferred subtasks, consisting of fewer inferences than 24, is about 36% of all the transferred subtasks, and consisting of fewer than 60 is about 81%. It means that the number of KABU-WAKE processings was large because the transferred subtasks consisted of a small number of inferences. Further the beginning and the ending of execution was generally not good. Also because of the long KABU-WAKE processing time and the large number of transferred subtasks, there were many idle PEs that sometimes became busy. This cause a lot of idle time. Figure 7 shows the dynamic change of status of each PE during the processing of Sentence 1. In Figure 7, busy, KABU-WAKE, and idle states have the following meanings:

-- Much idle time occurred during the first and last thirds of the execution. This accounted for more than 90% of all idle time.
-- Communication was distributed uniformly from the beginning to the end of execution. In the first third, the number of subtask transfers was 112, in the second third, 108, and in the last third, 125.
-- All of the PEs processed the majority of inferences during the middle third of execution.

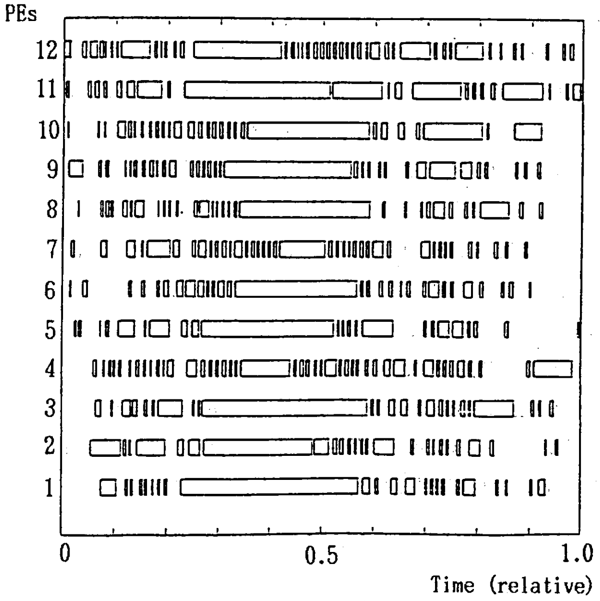


Figure 7 Change of status in each PE during processing of Sentence 1

The KABU-WAKE processing and the search tree of Sentence 1 have the above characteristics. When we return to the factors which cause overhead in parallel execution, the KABU-WAKE processing time per transaction was long and the number of KABU-WAKE processings was large. So, the overhead (KABU-WAKE time and idle time) took up more than 70% of all the time used to process Sentence 1.

Two difficulties in the KABU-WAKE method were revealed during analysis of the above experiment.

i. KABU-WAKE processing time per transaction is longer than it should be.

ii. The number of inferences which make up a transferred subtask have a large influence on the performance improvement ratio.

4.2.3 Estimated performance improvement ratio of Sentence 3

This section describes the estimated performance improvement ratio expected if Sentence 3 is processed with 100 PEs.

The following expression shows how the performance improvement ratio with 100 PEs is obtained.

$$\text{performance improvement ratio} = \frac{\text{(execution time with 1 PE)}}{\text{(execution time with 100 PEs)}}$$

The value of the execution time with 1 PE is based on actual measurement. During execution with 100 PEs, each PE does inference processing, does KABU-WAKE processing, or is idle. So, the execution time with 100 PEs is determined by the following expression.

```
execution time with 100 PEs =
     ((busy time) +
       (KABU-WAKE time) + (idle time))
     ---------------------------------
                   (100)
```

In the above expression, busy time is the sum of inference processing time from PE 1 to PE 100: we can regard the busy time as the execution time with 1 PE. The KABU-WAKE time is determined by the following expression.

```
KABU-WAKE time =
   ((KABU-WAKE processing time
              per transaction) *
      (number of subtask transfers))
```

Here, the value of the KABU-WAKE processing time per transaction is based on actual measurement of the execution of Sentence 3 with less than 12 PEs. To estimate the number of previous subtask transfers when Sentence 3 is processed with 100 PEs, we used the relationship between the number of PEs and the number of subtask transfers obtained by actually measuring the execution of Sentence 3 with less than 12 PEs. (In the KABU-WAKE method, the relation is linear [4],[6].) We estimated the idle time from the relationship between the number of PEs and the amount of idle time (shown in Figure 8) obtained by actually mesuring the execution of Sentence 3 with less than 12 PEs. Figure 8 indicates that the relationship is almost linear.

The above computations give an estimated performance improvement ratio of about 50 when Sentence 3 is processed with 100 PEs.

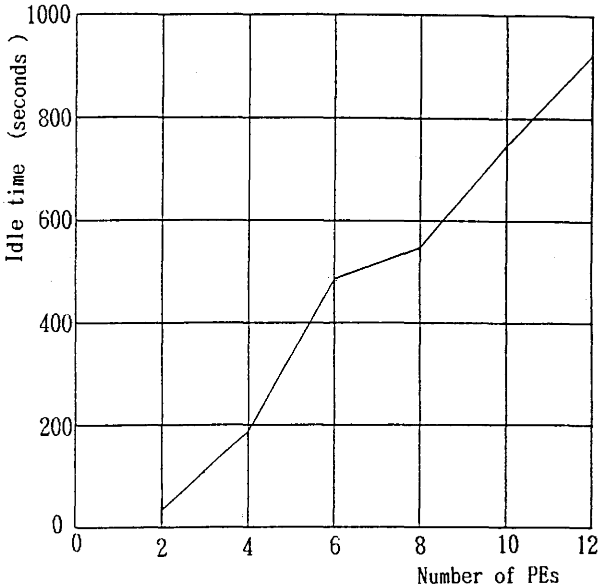


Figure 8 Relationship of number of PEs to idle time

5. Conclusion

We evaluated the KABU-WAKE method by running a bottom-up parser program with restricted a grammar on our experimental system. The results led us to the conclusions listed below.

(1) With the KABU-WAKE method, we can obtain a performance improvement ratio proportional to the number of PEs for large problems with a lot of parallelism. The method has the advantage for parallel inference that, as the amount of processing increases exponentially, the number of subtask transfers increases linearly.

(2) Difficulties in the KABU-WAKE method made evident through analysis of the cause of the low performance improvement ratio when 12 PEs were used is as follows:
  i. The KABU-WAKE processing time per transaction is longer than it should be.
  ii. The number of inferences which make up a transferred subtask greatly influences the performance improvement ratio.

(3) The estimated performance improvement ratio is about 50, when a problem that achieved a high performance improvement ratio with fewer than 12 PEs is processed with 100 PEs.

In our analysis described in 2, above, we saw that the KABU-WAKE processing time had a great deal of influence on the performance improvement ratio. We think it is important to reduce the KABU-WAKE processing time per transaction. For 2 i, above, we have built a KABU-WAKE compiler that reduces the KABU-WAKE processing time per transaction and are awaiting an evaluation of it. For 2 ii, as the KABU-WAKE processing time gets shorter, its influence on the performance improvement ratio becomes smaller. We are investigating other possible solutions to 2 ii that relate to the basic principles of the KABU-WAKE method.

We think that one of the practical applications in AI, the search problem, requires a large amount of processing with a large number of subtasks that can be processed using OR parallel. The KABU-WAKE method is very suitable for this type of application.

FGCS is attempting to develop an AND-STREAM parallel inference machine, which executes the GHC (Guarded Horn Clause) language [7]. From the point of view of GHC, OR parallel Prolog, that is pure Prolog, is a language that is used to describe special types of algorithms such as a search problem. Therefore, when we think of the GHC-oriented machine as a general purpose parallel inference machine, we can regard an OR parallel Prolog-oriented machine as a special-purpose parallel inference machine. As search problems appear in most AI applications, we think the OR parallel Prolog-oriented machine is quite valuable.

6. Acknowledgments

The authors would like to thank Mr. Tanahashi, Section Manager, and Mr. Sato, General Manager, for their unfailing encouragement and Mr. Yamada, Managing Director, for giving us the opportunity to conduct this research.

References

[1] Itashiki, et al., "PARALLEL INFERENCE PROCESSING SYSTEM -- EXPERIMENT OF IMPROVED CLAUSE UNIT PROCESSING METHOD", 30th meeting of Information Processing Society, 7c-7, March 1985. Japanese.

[2] Kumon, et al., "PARALLEL INFERENCE PROCESSING SYSTEM -- IMPROVED CLAUSE UNIT PROCESSING METHOD", 30th meeting of Information Processing Society, 7c-8, March 1985. Japanese.

[3] Kumon, et al., "EVALUATION OF KABU-WAKE PROCESSING", 31st meeting of Information Processing Society, 2c-5, October. 1985. Japanese.

[4] Sohma, et al., "A NEW PARALLEL INFERENCE MECHANISM BASED ON SEQUENTIAL PROCESSING", IFIP TC-10 Working Conference On Fifth Generation Computer Architecture, July 15-18, 1985. Also, J.V.Woods, "A NEW PARALLEL INFERENCE MECHANISM BASED ON SEQUENTIAL PROCESSING", Fifth Generation Computer Architecture, North-Holland, 1986.

[5] Mizoguchi,et al.,"Prolog and its applications", Sohken Syuppan Press, 1985. Japanese.

[6] Kumon, et al., "KABU-WAKE : A NEW PARALLEL INFERENCE METHOD AND ITS EVALUATION", COMPCON 86 Spring, March 4-6, 1986.

[7] Ueda,K., "Guarded Horn Clauses", TR-103, 1985.

A Very Fast Prolog Compiler on Multiple Architectures

Toshiaki Kurokawa Naoyuki Tamura Yasuo Asakawa
Hideaki Komatsu

Science Institute, IBM Japan, Ltd.
5-19 Sanban-cho, Chiyoda-ku, Tokyo 102, JAPAN

Abstract

In this paper, we report on our experiment on Prolog compiler technology. Targeted properties of the compiler are efficiency and portability. The generated code so far attained is so efficient to gain more than 1 MEGA LIPS on IBM 3090.

One of the speciality of the compiler is in the intermediate Virtual Prolog Machine Code, which enhanced efficiency and portability. Another advantage of the compiler is to generate PL.8 code which can be used on multiple machines including the IBM 370 and IBM RT-PC.

We introduce some declarative extensions to the Prolog language, which is compatible with the language and is powerful enough to produce efficient code.

1 Our Goals and Approaches

The goals of our optimizing Prolog compiler technology development were the following:

1. Development of the technology to produce one of the world's fastest Prolog compiler.
2. Both efficiency and portability achieved in the same framework.
3. Conducting the feasibility study of the compiler technology in a reasonably short period.
4. Evaluation of various machine architectures from the standpoint of the Prolog compilation.

To achieve these goals we have employed the following techniques:

1. Extension of the Prolog language for the generation of efficient code, while maintaining compatibility.
2. Adoption of the Virtual Prolog Machine scheme as the intermediate stage of the compilation process for effective optimization.
3. Use of the machine independent and efficient system programming language, PL.8[1], as the object code.

In this paper, we present the newly introduced declarations, an overview of the compilation and an evaluation of the compiler.

We have omitted arguments about Prolog such as in [6], and assumed the readers have some familiarity with Prolog language and programming[3].

2 Prolog Language Extensions

There are two extensions newly introduced to Prolog language: *notrail* and *type*. Both are declarations which can be attached to any predicate. So far, *mode* declaration in DEC-10 Prolog[2] is the only declaration that is used for optimizing compilation.

- *Notrail* declaration indicates that the annotated predicate actually behaves rather as a function so that there is neither backtracking nor ambiguous head selection. This declaration usually follows the *usage* declaration explained below.
- *Type* is declared actually in part of the *usage* declaration where input/output *mode* is also declared for the predicate. With it, we can declare the data type of each argument, not of each variable. So far, we limit type categories among the built-in data types such as *atom, nil, list, integer, structure, variable*, and their combination.

For example, we can define the list concatenation program, *append*, that is deterministic and accepts two input lists (possibly *nil*) and produces one output list. It is actually the same as the well-known *append* predicate but is declared as Figure 1.[1]

```
<- usage append(in:(list+nil),
                in:(list+nil),
                out:(list+nil)).
<- notrail append(*,*,*).

append({},X,X).
append({A|X},Y,{A|Z}) <- append(X,Y,Z).
```

Figure 1: APPEND with declarations

The introduction of *type* annotations is natural, in the sense that most contemporary high-level programming languages, even Lisp, have type declarations. However, the *notrail* declaration is special to the Prolog language, and may need some explanations.

[1] Note that we use VM/Prolog[4] syntax. In DEC-10 Prolog[2], square brackets ([]) would be used to indicate list data instead of curly braces ({, }).

CH2345-7/86/0000/0963$01.00 © 1986 IEEE

This declaration is introduced mainly from the following two reasons:

1. For the compiler writer, it helps the decision to eliminate the trailing of variables. Without it, all variables have to be trailed, or at least checked whether they must be trailed or not, even if the execution is deterministic.
2. For the Prolog programmer, this declaration eliminates the use of the *cut* operation. It is commonly observed with a novice programmer that he either puts in too many cuts or forgets to puts in cuts at all. Most of the Prolog predicates tend to be deterministic, especially when Prolog is used for system programming.

The benefit of the deterministic declaration varies with machine architectures. For the ordinary commercially available machines such as IBM S/370 or IBM RT-PC[5], the gain is significant because the trailing operation requires more instructions compared with special Prolog machines such as Tick and Warren's Pipelined Prolog Machine[11].

We have also introduced two auxiliary declarations: *key* and *entry*. *Key* declaration can be used to specify the argument which will be indexed. *Entry* declaration is used to declare the entry point and an entry declaration is necessary for a compilation unit.

More complicated examples of these declarations are shown in appendix A, which is really used for benchmark tests. That original version is defined in Prolog contest[10].

3 Outline of The Compilation Process

As noted in our goals, our emphasis lies in its *optimization* and *portability*. We adopt a *virtual machine* as an intermediate stage of our compilation so that we can enhance both the optimization and the portability.

The virtual machine, which we have adopted, is based on Tick and Warren's machine[11]. However, considering from the viewpoint of optimizing Prolog compiler, the level of its instructions is too high. For example, its instruction "*get_list A1*" implies the following:

```
test the tag of register A1
if reference then
    do dereference
    and retry
if unbound variable then
    create a list cell
    bind it to the variable
    check whether trailing is necessary or not
    if necessary then do trailing
    set the address of the list cell to register S
    set write mode
if list then
    set the address of the list cell to register S
    set read mode
otherwise
    fail
```

As far as "*get_list A1*" is a primitive instruction and it can not be decomposed into lower level instructions, there is no opportunity for optimization even if it is known that A1 is always a list. Therefore, in our virtual machine many low level instructions are introduced, which basically correspond to each statement in above example. The optimizer in our compiler eliminates redundant type-checking and mode-checking at this level.

Another special point of our approach is the adoption of PL.8 as an object code.

PL.8 compiler provides a low-level optimization. For example, we need not care about the register allocation. It also provides the portability among S/370 and RT-PC.

The compilation process can be summarized in the diagram in Figure 2.

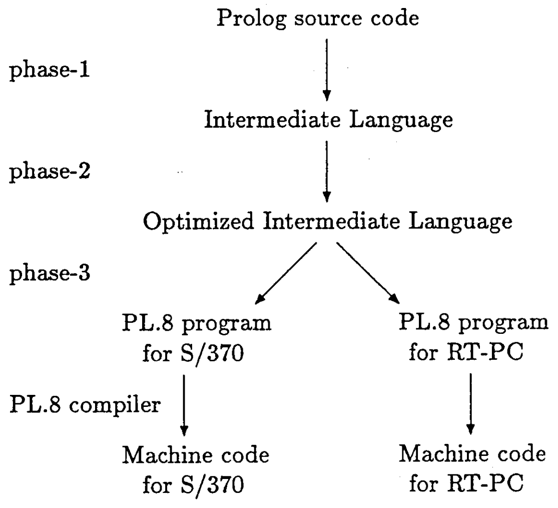


Figure 2: The outline of our compilation process

3.1 Phase-1

In this phase, Prolog program is translated to intermediate language, which is called *WIL*. This is considered as mapping process of Prolog language into the virtual machine. From this point of view, some optimization is done. For example,

- tail recursion optimization[12]
- decision of unification order using *mode* information
- detection of *unsafe variable* using *type* and *mode* information
- selection of the best code for builtin predicates using *type* information
- variable classification

Type and mode information is also generated so that optimization can be done effectively in the following phase.

In Figure 3, a part of output of phase-1, which is generate from *append*, is shown. It corresponds to the indexing part and unification of the first argument of the second clause in Figure 1. This will be expanded to low-level code before optimization process. Expansion is very simple and straightforward. An example of expanded code is shown in Figure 4.

```
asa3:
   assertion(type(a(1)) = list + {}
                         + ref( \= undef));
   select type(a(1)) of {
     ref( \= undef) -> {
        deref(a(1));
        goto(asa3) };
     {} -> goto(asa1);
     list -> goto(asa2);
     otherwise -> failure};
 ..............................
asa2:
   get_list(a(1))
        where trail = no &
              type(a(1)) = list + {}
                         + ref( \= undef) &
              deref = no;
   unify_variable(x(1),0);
   unify_variable(x(2),1);
 ..............................
```

Figure 3: A part of output of phase-1 for *append*

3.2 Phase-2

In this phase, the intermediate code is translated into a graph for optimization. The optimization consists of following two steps:

1. tracing the graph to infer the behavior of the intermediate code by *semantic definition of the intermediate language WIL*
2. modifying the graph by *graph reduction rules*

This optimization process is repeated several times and after that the optimized graph is translated back to the intermediate code. Roughly speaking, by this optimization,

- redundant case instructions
- never-selected case entries
- unreachable instructions

are eliminated. Detailed discussions about this optimization techniques will be found in our accompanying paper [8], especially from the *knowledge-based* viewpoint.

Figure 5 shows the result of optimization of the code shown in Figure 4.

```
asa3 :
    case type(a(1)) of {
        ref( \= undef) ->
            goto(tn82);
        {} ->
            goto(asa1);
        list ->
            goto(asa2);
        atom + int + struct + ref(undef) ->
            goto(tn81)
    };
tn82 :
    deref(a(1));
    goto(asa3);
.......................
asa2 :
    case type(a(1)) of {
        ref ->
            goto(tn64);
        list ->
            goto(tn62);
        {} + atom + int + struct ->
            goto(tn60)
    };
tn64 :
    get_list_w(a(1));
    setmode(write);
tn56 :
    case mode of {
        write ->
            goto(tn58);
        read ->
            goto(tn57)
    };
tn58 :
    unify_variable_w(x(1),0);
tn53 :
    case mode of {
        write ->
            goto(tn55);
        read ->
            goto(tn54)
    };
tn55 :
    unify_variable_w(x(2),1);
tn52 :
.......................
tn54 :
    unify_variable_r(x(2),1);
    goto(tn52);
tn57 :
    unify_variable_r(x(1),0);
    goto(tn53);
tn62 :
    get_list_r(a(1));
    setmode(read);
    goto(tn56);
.......................
```

Figure 4: Example of expanded code

```
  asa3 :
      case type(a(1)) of {
          ref( \= undef) ->
              goto(tn82);
          {} ->
              goto(tn77);
          list ->
              goto(tn62)
      };
  tn82 :
      deref(a(1));
      goto(asa3);
........................
  tn62 :
      get_list_r(a(1));
      setmode(read);
      unify_variable_r(x(1),0);
      unify_variable_r(x(2),1);
........................
```

Figure 5: Output of phase-2

3.3 Phase-3

The code obtained in phase-2 is machine independent. In phase-3, the code is translated into PL.8 program. During this translation information which depends on the target machine is used to optimize the following:

- design of tag
- primitive operations
- the order of case entries

Figure 6 is an example of PL.8 code generated for S/370 from the code shown in Figure 5.

3.4 PL.8 compiler

As described in [1], PL.8 compiler does many kinds of optimization. For example, register allocation, dead code elimination, code motion, value numbering, dead store elimination, straightening, and so on. Off course, those are also effective in our case. These are done in global viewpoint.

PL.8 compiler can also do optimization which is considered in Warren's "Abstract Prolog Instruction Set"[13] level. Value numbering and register allocation algorithm used in PL.8 compiler can do same optimization shown in Figure 7[2].

4 Evaluation

The compiler itself is written entirely in high-level programming languages, namely, VM/Prolog and PL.8. Due to the quality of the debugging environment for VM/Prolog[9] and for PL.8, the prototype compiler has been developed in short term.

The compiler itself runs on VM/CMS on IBM S/370, producing code both for IBM S/370 and for IBM RT-PC.

```
asa3:
select;
when(  shiftr(a1,28)=4 )
goto tn62;
when( a1= ('30000000'xb^('OFFFFFFF'xb &0)) )
goto tn77;
otherwise
goto tn82;
end;
tn82:

 /*** DEREF(a1)***/
 a1 =
 ptr(
 a1
 ,memory)->w;
goto asa3;
........................
tn62:

 /*** G_LIST_R(a1)***/
 s = ('00000000'xb^('OFFFFFFF'xb &a1));
mode = rmode;

 /*** U_VAR_R(x1,0)***/x1=s->strfrm.w0;

 /*** U_VAR_R(x2,1)***/x2=s->strfrm.w1;

........................
```

Figure 6: Output of phase-3(for S/370)

Although full facilities of Prolog are not yet implemented, enough facilities are realized to handle list processing and arithmetics. With these functions, we can measure LIPS[3] values that are presented in Table 1. We also measured the performance by some more complicated programs such as the classical "eight-queen problem", which is shown in appendix A. The measured execution times are shown in Table 2.

As our current prototype compiler have several points to

```
get_list A1
unify_variable X4
unify_variable X5    -->  unify_variable A1
get_variable X6,A2   -->  (deleted)
get_list A3
unify_value X4
unify_variable X7    -->  unify_variable A3
put_value X5         -->  (deleted)
put_value X6         -->  (deleted)
put_value X7         -->  (deleted)
execute append/3
```

Figure 7: Example of-Warren's Abstract Prolog Instructions level optimization

[2] This example is found in [11] and [13].

[3] LIPS (Logical Inferences Per Second) is usually measured by a simple list manipulation program, *i.e.* a simple *append* or a naive *reverse*.

be tuned up, we can reasonably expect the final LIPS values better than those presented in Table 1. Note that the speed on IBM 3090 has exceeded one MEGA-LIPS, which is a good intermediate step to achieve the goals raised in Fifth Generation Computer Project [7]. Also note that in the figures for the IBM RT-PC, all system overheads are included. In other words, we measured real time performance in IBM RT-PC.

	RT-PC	3081K	3090
Without hints	56	611	1000
With hints	87	827	1420

Table 1: LIPS value measured on IBM S/370 and IBM RT PC. The unit is KLIPS. The benchmark program is *simple append* with 1000 elements. Hints mean *mode*, *type*, and *notrail* declarations.

	3081K		RT-PC	
	Without hints	With hints	Without hints	With hints
First	5	14	65	178
All	92	232	1066	2860

Table 2: Execution time of 8-queen problem. The unit is millisecond. Hints mean *mode*, *type*, and *key* declarations.

Table 3 shows the effectiveness of the optimizations which are done in phase-2 and in PL.8 compiler. From this table it is found that the optimizer in PL.8 compiler does the optimization considered in Warren's Instruction level, shown in Figure 7. But it is also found that optimization in low-level virtual machine instructions is also effective and combination of these two level optimization is very effective.

Table 4 shows the effectiveness of newly introduced declarations. According to this table, *mode* declaration, only itself is little useful for optimization. Combination of *mode* and *type* declaration, we call it *usage* declaration, is very effective. *Notrail* declaration is also effective, especially for RT-PC, which is the so-called workstation and which memory is much slower than host-class machines.

5 Concluding Remarks

It is not an easy task to conclude in this kind of evolutionary activities. However, it should be noted that our main emphasis lies in the technology development, not in the product development. In other words, when one tries to make a product-level Prolog compiler based on our technology, he or she may have to trade off efficiency for functionality.

Our prototyping is, in analogy to physics, an experiment conducted only to prove the correctness or feasibility of our approaches. We conducted an extensive review on similar works, and as far as we know, our compiler has achieved the highest performance for Prolog compilation.

	3081K		RT-PC	
	Without hints	With hints	Without hints	With hints
No optimization	1	1.06	1	1.09
Prolog only	1.68	1.92	1.62	1.86
PL.8 only	1.64	1.85	1.68	1.74
Prolog+PL.8	3.30	4.47	2.80	4.35

Table 3: Effect of optimization: relative values of LIPS in case of append. Hints mean *mode*, *type*, and *key* declarations.

	3081K		RT-PC	
	Without notrail	With notrail	Without notrail	With notrail
None	1	1.08	1	1.15
Mode only	1.03	1.26	1.04	1.30
Type only	1.07	1.16	1.23	1.36
Mode+Type	1.24	1.40	1.47	1.97

Table 4: Effect of declaration: relative values of LIPS in case of append. Optimization is done both in phase-2 and PL.8 compiler.

Acknowledgement

Drs. Peter Y. Woon and Tetsunosuke Fujisaki have provided us one of the best environment for this work. Ahmed Chibib at IBM Austin, helped us to fix the usage of PL.8 language.

References

[1] Auslander, M. and Hopkins, M., "An Overview of the PL.8 Compiler", *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction,* Volume 17, Number 6, June 1982.

[2] Bowen, D.L., "DEC system-10 PROLOG USER'S MANUAL", Dept. of Artificial Intelligence, Univ. of Edinburgh, 1981.

[3] Clocksin, W. F. and Mellish, C. S., " Programming in Prolog", Springer-Verlag, 1981.

[4] International Business Machines Corporation, "VM / Programming in Logic, Program Description / Operations Manual", No.SH20-6541-0,1985.

[5] International Business Machines Corporation, "RT Personal Computer Technology", No.SA23-1057, 1986.

[6] Kurokawa, T., "LOGIC PROGRAMMING – What does it bring to the software engineering", *Proceedings of First International Conference on Logic Programming*, pp.134-138, Marseille, 1982.

[7] Moto-Oka, T., (ed.) "Fifth Generation Computer Systems", North-Holland, 1982.

[8] Tamura, N. "Knowledge based optimization in Prolog compiler", to appear in Proc. of the 1986 ACM/IEEE Computer Society Fall Joint Computer Conference

[9] Numao, M. and Fujisaki, T., "Visual Debugger for Prolog", *Proc. of The Second Conference on Artificial Intelligence Applications*, pp.422-427, IEEE Computer Society, 1985.

[10] Okuno, H., "The benchmarks for The Third Lisp Contest and The First Prolog Contest", Information Processing Society of Japan, WGSYM No.20-4, 1984.

[11] Tick, E. and Warren, D.H.D., "Towards a Pipelined Prolog Processor", *Proc. of 1984 International Symposium on Logic Programming*, IEEE Computer Society, 1984.

[12] Warren, D.H.D., "An Improved Prolog Implementation which Optimises Tail Recursion", *Proc. of Logic Programming Workshop*, pp.1-11, 1980.

[13] Warren, D.H.D., "An Abstract Prolog Instruction Set", SRI International Technical Note 309, October 1983.

Appendix

A A program to find all solution of 8-queen problem

```
<- entry queen8.

<- usage try(in:int,in:(list+nil),
             in:(list+nil),out:(list+nil),
             in:(list+nil),in:(list+nil)).
<- usage generate(in:int,out:(list+nil)).
<- usage selectx(in:(list+nil),
                 out:int,out:list).
<- usage notmem(in:int,in:(list+nil)).

<- key(try(*,*,*,*,*,*),2).
<- key(notmem(*,*),2).


queen8 <- start_timer & queen(8,L) & fail.
queen8 <- display_timer.

queen(N,L) <-
    generate(N,L1) & try(N,L1,{},L,{},{}).

generate(0,{}).
generate(N,N.L) <-
    gt(N,0) & diff(N,1,N1) & generate(N1,L).

try(*,{},L,L,*,*).
try(M,S,L1,L,C,D) <-
    selectx(S,A,S1) &
    sum(M,A,C1)  & notmem(C1,C) &
    diff(M,A,D1) & notmem(D1,D) &
    diff(M,1,M1) &
    try(M1,S1,{A|L1},L,{C1|C},{D1|D}).

selectx({A|L},A,L).
selectx({A|L},X,{A|L1}) <- selectx(L,X,L1).

notmem(*,{}).
notmem(A,{B|L}) <- ine(A,B) & notmem(A,L).
```

A RELATIONAL DATABASE MACHINE BASED ON FUNCTIONAL PROGRAMMING CONCEPTS

Yasushi KIYOKI, Kazuhiko KATO and Takashi MASUDA

Institute of Information Sciences and Electronics
University of Tsukuba
Sakura, Niihari, Ibaraki 305, Japan

ABSTRACT

We present a novel approach to a relational database machine for processing knowledge bases. This approach is based on functional programming concepts in order to manage processor resources and memory resources with the theoretical neatness of functional computation. By using demand-driven evaluation as a driving method of functional computation, parallelism can be exploited in executing relational operations (relational database operations) and inference operations based on unification. Furthermore, these operations can be executed avoiding the complexity of resource management within a restricted resource environment. This approach is implemented under a multiprocessor architecture combined with a demand-driven evaluation mechanism. In this paper, we define the basic primitives which are used to implement demand-driven evaluation and function application. We also present a basic algorithm and a system architecture for executing basic operations for knowledge bases by using a demand-driven evaluation mechanism. To ascertain feasibility of our approach, a relational operation system has been implemented on the basis of the approach.

1. Introduction

The relational model [4] has received much attention as a promising data model for implementing database systems with theoretical basis. Furthermore, it has been recognized that the model is significant for implementing knowledge base systems based on mathematical logic, and it has been studied how relational database concepts can be applied to mathematical logic [6].

It is known that basic operations in such knowledge base systems are relational operations and inference operations based on unification. The important feature of these operations is that they deal with a vast amount of data represented as relations.

We consider the basic operations within the framework of functional programming concepts. Functional programming [2] include many attractive concepts, and its fundamental concepts are summarized as follows:

(1) The value of a function expression depends only on its textual context, not on computational history. This notion is referred to as referential transparency.

(2) Functional computations are free of side effects. The parameter passing mechanisms call-by-value, call-by-name and call-by-need have the same semantics.

(3) Functional programs often contain implicit and easily detected parallelism [5].

In this paper, we present a novel approach to a relational database machine for processing knowledge bases. In this approach, functional programming concepts are applied to both relational and inference operations in order to exploit parallelism in a natural way and to manage processor resources and memory resources with the theoretical neatness of functional computation.

In [16], relational operations have been described in a dataflow language based on data-driven evaluation [1]. Processing of relational operations has been considered as stream processing in the data-driven evaluation including the eager and lazy evaluations. Furthermore, in [17], an advanced stream-oriented algorithm which exploits parallelism inherent relational operations has been presented in detail.

In the approach presented in this paper, demand-driven evaluation is used in computing functions of relational and inference operations.. A relational operation, such as the selection operation or the join operation, is defined as a function, and operand relations of the relational operation are manipulated as arguments of the function. The arguments corresponding to operand relations are evaluated as streams of tuples by using the demand-driven evaluation mechanism. By using this evaluation mechanism in functional computation, it becomes possible to execute both relational and inference operations in parallel within restricted computing resources. In this approach, the stream-oriented algorithm [17] can also be realized within the framework of demand-driven evaluation.

In this paper, we also discuss how relational database concepts can combine with inference operations based on unification. In our approach, a set of fact clauses in Horn clauses is represented as a relation, and the fact clauses are manipulated by using functional computation.

The relational database machine proposed in this paper exploits parallelism inherent in knowledge base processing, and makes it possible to execute basic operations for knowledge bases within a restricted resource environment.

2. Approach to Functional Computation

In our approach, each process of relational and inference operations is described on the basis of functional programming concepts. In order to realize several driving methods of functional computation, basic primitives are defined. The basic primitives sup-

CH2345-7/86/0000/0969$01.00 © 1986 IEEE

port various kinds of parameter passing mechanisms required in functional computation.

2.1 Demand-driven Evaluation in Functional Computation

The methods of driving functional computation are classified [20],[21] as follows:

(1) Demand-Driven Evaluation,
(2) Data-Driven Evaluation, and
(3) Sequential Evaluation.

We employ demand-driven evaluation in executing relational operations and in executing inference operations, in order to execute those operations in parallel within a limited resource environment. Demand-driven evaluation generally introduces a fair amount of overhead in issuing demands. However, the demand-driven evaluation allows better control of parallelism, more selective evaluation, and a natural way of handling a large amount of data within a limited resource environment. The advantage of demand-driven evaluation includes the potential for eliminating a vast amount of computation by evaluating only what is necessary for computing the result. Since relational operations and inference operations generally require to manipulate a large amount of data, the potential inherent in demand-driven evaluation is effectively utilized. In executing these operations, granularity of data which is transferred by a single demand can make large. Therefore, when compared with the total amount of data transferred by a single demand, the overhead caused by the demand transfer is insignificant.

In this approach, the following parallelisms inherent in functional computation are exploited:

(1) parallel evaluation for arguments of a function,
(2) parallel execution between a function which generates its return value as a stream [10] and a function which consumes the stream as an actual argument. That is to say, stream-oriented parallel processing between a function of stream-producer (producer function) and a function of a stream-consumer (consumer function).

To exploit the parallelism of (1), demands are simultaneously issued from a consumer function to producer functions which generate actual arguments of the consumer function. Consequently, the independent operations can be executed in parallel.

The parallelism of (2) is exploited between a producer function and a consumer function. In data-driven evaluation, to exploit the parallelism, it is necessary for the consumer function to begin the computation eagerly before the producer function completes producing all intermediate results [1]. To extract the parallelism in demand-driven evaluation, it is necessary for the producer to begin the computation eagerly before the demand arrives from the consumer. In our approach, parallelism is exploited by pre-issuing a demand to the producer function before the consumer function begins computation. When the producer function receives a demand pre-issued from the consumer function, the producer function begins computation. In producing a stream, the producer function does not produce every stream element by a single demand. The producer generates some fixed amount of stream elements by a single demand. After the producer function completes producing the fixed amount of stream elements, it suspends computation and waits for the subsequent demand. As a result, the stream-oriented parallel processing [17] between the producer and consumer functions can be performed within the framework of demand-driven evaluation.

In this approach, each relational operation or each inference operation is defined as a function. Several arguments of the function correspond to operand relations of an operation, and such a argument is evaluated as a stream of tuples in a relation.

To implement demand-driven evaluation, call-by-name or call-by-need is employed as a parameter passing mechanism. That is, an argument of a function is not evaluated until a reference to the argument is encountered in the execution of the function body. In call-by-name, if a formal argument is encountered more than once, the corresponding actual argument is reevaluated each time it is encountered in the function body. In this case, the actual argument can be deleted after a reference to it is completed. Therefore, if call-by-name is employed in evaluating the argument corresponding a stream of tuples, relational and inference operations can be performed within limited memory resources. However, when the same argument is encountered more than once in the function body, it must be reevaluated, that is, the function which generates the stream corresponding to the actual argument must be recomputed. We call this parameter passing mechanism "recomputation mechanism."

On the other hand, in call-by-need, a formal argument is evaluated only once when the first reference is encountered. The evaluated actual argument is used in the other references to the argument. In this parameter passing mechanism, the actual argument must be retained until every reference to it completes. If the actual argument is huge like a stream of tuples in a relation, it seems that memory could be swamped. However, recomputation of the same function is unnecessary. This mechanism is referred to as "caching mechanism"[11]. The decision of a parameter passing mechanism is important in functional computation of relational operations or inference operations, because actual arguments of functions, which are relations, are generally very large. If recomputation mechanisms are used in evaluating every argument, computations may increase drastically. On the other hand, if caching mechanisms are used, the memory overflow may cause heavy overhead. In our approach, both recomputation and caching mechanisms are supported and are used together.

2.2 Basic Primitives

Our approach to parallel processing for relational and inference operations is based on the following strategy:

(1) Each operation for processing knowledge bases is defined as a function. A query is decomposed into several function applications. The relationship between operations, that is the relationship between a producer function of a stream and a consumer function of the stream, can be decided at compile time for a query.

(2) Referential transparency is ensured among functions. A function is allocated to one of the multiple processors connected to a communication network. As a result, parallelisms based on demand-driven evaluation are exploited among functions. Independent functions are evaluated in parallel. Furthermore, stream-oriented parallelism between a producer function of a stream and a consumer function of the stream is also exploited under demand-driven control. If another function is called in the function during computation of the function body, it can be allocated to another processor and these functions can be executed in parallel within the framework of demand-driven evaluation.

(3) An individual function is compiled so as to maximally extract the ability of the processor architecture to which the function is allocated. If a sequential processor is used to compute the individual function, the function is compiled into the sequential object codes. For example, if data-driven-processor is used to compute the individual function, the function is compiled into single-assignment codes.

In this subsection, the basic primitives for realizing demand-driven evaluation and for realizing function application are presented. The basic primitives are implemented at the architecture level of each processor in the relational database machine shown in Section 4.

channel(type, granularity, parameter_passing_method)

The primitive "channel" specifies a channel between a producer function instance of a stream and a consumer function instance of the stream. This primitive returns the identifier "cid" of the channel as a return value. The channel is used to communicate a stream between two function instances. The channel corresponds to a buffer which stores elements of a stream. As the properties of a channel, "type," "granularity" and "parameter_passing_method" are specified. Here, "type" indicates the data type of an element of a stream, and "granularity" indicates a amount of data transferred by a single demand. The buffer size of the channel is decided according to granularity. And, "parameter_passing_method" indicates "recomputation" or "caching" alternatively as the parameter-passing mechanism for a formal argument corresponding to a stream.

new(f, pid, cid, parameters)

This primitive creates a function instance of a function specified by "f." Here, "pid" is the identifier of the processor to which the function instance is allocated, and "cid" indicates the output channel for the stream returned from the function instance. The formal arguments (arguments of input streams and the other arguments) of the function are specified in "parameters." The function instance does not begin computation until a demand is issued to it from the consumer function of its output stream.

When a query is decomposed into relational operations or inference operations, function instances corresponding to those operations are created by using the primitive "new" and they are connected to channels by using the primitive "channel." When a function instance requires to call another function or the function itself (recursive call) during the execution of the function body, these primitives are specified in the definition of the function body.

pre-demand(cid)

This primitive is used to issue a first demand from a consumer function of a stream to the producer function of the stream. The channel for passing stream elements is indicated as "cid." This primitive is one of the basic primitives which implement demand-driven evaluation. By pre-issuing a first demand, the producer function can begin computation eagerly. As a result, parallelism is exploited between the consumer function and the producer function.

get1(cid)

This primitive accesses an element of a stream in the buffer of the input channel indicated by "cid" and returns the element as the return value. If the buffer is vacant, this primitive issues a demand to the producer function of the stream, then waits until the buffer is refilled with stream elements. Each element is deleted from a buffer once it is accessed by this primitive.

get2(cid)

When this primitive is used, the double buffering mechanism must be supported in the channel indicated by "cid." While the producer function stores stream elements in one area of the buffer, the consumer function can access a stream element stored in the other area by using this primitive. When one area of the buffer is vacant, this primitive pre-issues a demand to the producer function to have the area refilled. Then, it begins to access a stream element in the other area. Each element is deleted from a buffer once it is accessed by this primitive.

put1(d)

This primitive stores a stream element, which is indicated by "d," as a return value of a function in the buffer of the output channel. When the buffer is filled with stream elements, that is, when the amount of data indicated as granularity is generated, the execution of this primitive is suspended until the subsequent demand arrives.

put2(d)

When this primitive is used, the double buffering mechanism must be supported in the output channel. While the consumer function accesses a stream element stored in one area by using the primitive "get2(cid)," the producer function can simultaneously stores a stream element indicated by "d" as a return value in the other area of the buffer by the primitive put2(d). When the area of the buffer is filled with stream elements and when the subsequent demand is received, this primitive begins to store a stream element in the other area of the buffer. Otherwise, it waits until the subsequent demand is received.

The primitives "get1(cid)" and "put1(d)" are used when two function instances which communicate via the channel "cid" are allocated to the same processor. On the other hand, the primitives "get2(cid)" and "put2(d)" are used when two function instances which communicate via the channel "cid" are allocated to different processors. When "get2" and "put2" are used between two function instances, the stream-oriented parallelism is exploited between these function instances.

If another function is called in the function body, a new function instance is created by using the primitive "new" in the function body. Furthermore, new channels are specified by using the primitive "channel" to pass streams as actual arguments and to receive a stream as a return value . In this case, the stream elements are passed to the new instance via channel "cid" by primitive **"send1(cid)"** or **"send2(cid)."** The primitives "send1" and "send2" are used as "put1" and "put2," respectively. However, in "send1" and "send2," the channel identifier ("cid") is explicitly specified to pass stream elements to the new function instance. The output stream of the new function instance, that is the return value, is received by a primitive **"receive1(cid)"** or **"receive2(cid)."** The primitives "receive1(cid)" and "receive2(cid)" are used as "get1" and "get2," respectively.

mark_end_of_stream()
This primitive writes "EOS" into the end of an output stream as an identifier indicating the end of the stream.

check_end_of_stream(cid)
This primitive detects the end ("EOS") of a stream. It returns a logical value "TRUE" if the end of the stream is detected. Otherwise, it returns "FALSE."

2.3 Functional Computation

The amount of data propagated by a single demand is referred to as "granularity." As described in 2.2, granularity is indicated as a property of the channel. When a relational operation is described as a function, three kinds of granularity can be specified as follows:

(1) tuple-level granularity
(2) page-level granularity
(3) relation-level granularity

In tuple-level granularity, although the size of the buffer between the producer and the consumer can be minimized, many demands may be issued to the producer function. This may cause heavy communication overheads [3]. In relation-level granularity, the whole intermediate data, that is the complete intermediate relation, is produced by a single demand. In this granularity, the buffer of a channel is required to store the whole intermediate relation. Furthermore, the stream-oriented parallelism between producer and the consumer functions is not exploited.

In each granularity, it can be specified whether stream-oriented parallel processing is performed between the producer function and the consumer function. When a formal argument corresponding to a stream is encountered more than once in a function body, one of the parameter passing mechanisms ("recomputation" or "caching") is indicated as the property of the channel.

In the following, several function definitions of relational operations are presented. The following programs abstractly show compiled object codes of functions by using the notation of the C language. In our approach, if a sequential processor is used to execute a function, the function is compiled into sequential object codes including basic primitives. The basic primitives are implemented at the architecture level in the processor.

(1) page-level granularity without stream-oriented parallelism

Stream elements (tuples of a relation) can be transferred via the limited size of buffer. The size of buffer is referred to as "granularity" and it is equal to the "page size." When the program of a function "selection" receives a demand from another function, "get1(cid)" issues a demand to the producer function of a stream, and then waits for a page of stream elements. When the page of stream elements is stored in the input buffer, the "selection" function begins to compute the selection operation and stores the resulting elements in the output buffer of the output channel by the primitive "put1(d)" until the output buffer is filled. In this case, according to granularity, several demands are issued to the function producing the stream elements.

Program 1

```
define_function selection(relation, a)
stream relation;
item a;
{
    tuple tu;
    while (!check_end_of_stream(relation)) {
         tu = get1(relation);
         if (selection_test(tu, a))
              put1(tu);
    }
    mark_end_of_stream();
}
c1 = channel(tuples, INPUT_BUFFER_SIZE,
            RECOMPUTATION or CACHING);
/* specification of channel for output stream */
c2 = channel(tuples, OUTPUT_BUFFER_SIZE,
            RECOMPUTATION OR CACHING);
/* specification of channel for input stream */
new(selection, pid, c2, c1, a);
/* creation of function instance */
```

(2) page-level granularity with stream-oriented parallelism

In demand-driven evaluation, a demand is issued to the producer function when the actual argument is encountered. However, in order to perform stream-oriented parallel processing between consumer and the producer functions, the demand must be pre-issued from the consumer function to producer function. It is achieved by using the primitives as shown Program 2. In the function "selection," the primitive "pre-demand" pre-issues a first demand to the producer function of the input stream, and then the execution of the selection operation begins. At the same time, the producer function begins function computation by the pre-issued demand, and stores a resulting page into the output buffer. In function "selection," a demand is pre-issued again to the producer function by the primitive "get2(cid)," and begins computing the selection operation to the page. As the result of the pre-issued demand, the page is stored by the primitive "put2(d)" in the producer function.

In this function, demands are always pre-issued by "pre-demand" or "get2(cid)." This stream-oriented parallelism is exploited by supporting the double buffering mechanism in the buffer of a channel between the consumer function and the producer function of its actual argument.

Program 2

```
define_function selection(relation, a)
stream relation;
item a;
{
    tuple tu;
    pre-demand(relation); /* pre-issuing a demand */
    while (!check_end_of_stream(relation)) {
         tu = get2(relation);
         if (selection_test(tu, a))
              put2(tu);
    }
    mark_end_of_stream();
}
c1 = channel(tuple, INPUT_BUFFER_SIZE,
            RECOMPUTATION or CACHING);
c2 = channel(tuple, OUTPUT_BUFFER_SIZE,
            RECOMPUTATION or CACHING);
new(selection, pid, c2, c1, a);
```

(3) page-level granularity with stream-oriented parallelism (Several references to same argument are encountered.)

In the function "selection," a reference to the input stream is encountered only once. When references to the input stream are encountered more than once in a function body, it is required to reproduce the same stream by recomputation or to retain the whole stream by caching. For example, binary relational operations, such as the join or union operations, require to refer to the same input stream (the stream of the inner relation) more than once [17]. Therefore, recomputation or caching is specified alternatively as the property of a channel. In the case of recomputation, function computation can be performed within the limited memory resource. In the case of caching, the retained stream data may cause memory overflow. The function "join" is shown in Program 3. In this function, tuples in the outer-relation page are sorted on the joining attribute, and then each tuple in the inner-relation page is joined with tuples of the outer-relation page by using binary search algorithm.

Program 3

```
define_function join(relation_1, relation_2,pagesize)
stream relation_1; /* stream of outer-relation */
stream relation_2; /* stream of inner-relation */
int pagesize;      /* size of outer-relation page */
{
   tuple in1[pagesize], in2;
   int i;
   pre-demand(relation_1);
   pre-demand(relation_2);
   while (!check_end_of_stream(relation_1)) {
     for (i = 0; (i < pagesize) &&
           !check_end_of_stream(relation_1); i++)
           in1[i] = get2(relation_1);
      sort(in1); /* sorting of outer-relation page */
          while (!check_end_of_stream(relation_2)) {
           in2 = get2(relation_2);
           if (!check_end_of_stream(relation_1) &&
               check_end_of_stream(relation_2))
              /* pre-issuing a demand to request
              re-reference to the stream of
              inner-relation */
             pre-demand(relation_2);
                if (binary_search(in1, in2))
                 put2(concatenate(in1, in2));
                /* comparing an inner-relation tuple
                   (in2) with outer-relation tuples
                   (in1) by binary search algorithm */
                /* concatenating tuples if joining
                   condition is satisfied */
          }
       }
       mark_end_of_stream();
}

c1 = channel(tuple, BUFFER_SIZE_1,
               RECOMPUTATION or CACHING);
/* channel for input stream of outer-relation */
c2 = channel(tuple, BUFFER_SIZE_2,
            RECOMPUTATION or CACHING);
/* channel for input stream of inner-relation */
c3 = channel(tuple, OUTPUT_BUFFER_SIZE,
            RECOMPUTATION or CACHING);
/* channnel for output stream */
new(join, pid, c3, c1, c2, BUFFER_SIZE_1);
```

(4) relation-level granularity

When the producer function is required to be reduced completely by a single demand, "complete_reduction" is specified as granularity. In this granularity, operand source relations or intermediate relations may overflow the limited size of buffer. Furthermore, stream-oriented parallelism is not exploited between producer and consumer functions. To realize relation-level granularity, the function "selection" shown in Program 2 is used, and "complete_reduction" is specified as granularity in the primitive "channel" as follows:

```
c1 = channel(tuple, COMPLETE_REDUCTION, RECOMPUTATION
or CACHING);
c2 = channel(tuple, COMPLETE_REDUCTION, RECOMPUTATION
or CACHING);
new(selection, pid, c2, c1, a);
```

3. Relational Operations and Inference Operations

Functional programming concepts can be applied to various kinds of applications. In this section, we discuss an approach to processing relational and inference operations in knowledge base systems. A basic algorithm of stream-oriented parallel processing for relational operations is discussed in [17] in detail. In this section, the algorithm for relational operations is briefly reviewed, and the stream-oriented algorithm for inference operations is presented.

3.1 Relational Operations

When a function of a relational operation receives a demand from the consumer function, it begins accessing tuples in its input buffer, then executes the relational operation until it completes the production of one resulting page of tuples in the output buffer. The output buffer is then regarded as the input buffer for the consumer function. An individual function instance of a relational operation is not required to create a whole intermediate relation by a single demand. Each function instance creates only one page of tuples specified as granularity. Individual buffers do not require the capacity to store an entire intermediate relation. In this algorithm, if a producer function and a consumer function are allocated to different processors, it is assumed that the double buffering mechanism is supported in every buffer. That is, while the consumer function gets tuples in one area of its input buffer, the producer function can store tuples in the other area of the same buffer at the same time.

Just before a consumer function begins accessing tuples in one of two areas of its input buffer, it pre-issues a demand to the producer function to make the other area of the input buffer refilled with the subsequent page.

As a result, stream-oriented parallel processing is performed between the producer and consumer functions. By using demand-driven evaluation, unary relational operations (the selection, restriction and projection operations), and binary relational operations (the join, union, intersection, difference and Cartesian-product operations) can be concurrently executed within a limited memory resource environment. In particular, this algorithm shows attractive advantages in executing the join, union and Cartesian-product operations, which are the most time-consuming and the most memory-consuming operations.

3.2 Inference Operations

The combination of logic programming concepts with relational databases often appears as a promising approach to knowledge base processing [6],[18]. In our approach, sets of Horn clauses are combined with relational databases. Inference operations based on unification are executed within the framework of functional programming concepts.

Three ways to combine logic programming concepts with relational databases can be considered as follows:

(1) A logic program is used as a high-level query for a relational database system. In this way, a query written in a logic program is translated into a sequence of relational operations (relational database operations), and the sequence of relational operations are executed by a relational database operation system.

(2) Relations in a relational data base are regarded as sets of Horn clauses. An inference system operates unification for relations. That is, the relational database is regarded as a part of sets of Horn clauses, and inference operations are executed in relational databases. Such a database is generally referred to as a deductive database [6]. A relation is regarded not only as a set of tuples but also as a set of fact clauses. The relational database is regarded as the collection of relations and the collection is defined as the extensional database [6]. Relations are used to store fact clauses, and rule clauses are referred to as the intensional database [6]. That is, fact clauses are distinguished from rule clauses, and they are stored as relations. That is, elementary information is classified into rules and facts. Sets of facts are represented explicitly as a relational database.

(3) As in (2), a relational database is also used to represent logic programs. However, not only a fact clause but also a rule clause are represented as a tuple [22]. Inference operations are executed in a relational database with the rule and fact clauses. This way is effective when the rule base is huge. This is because the rule base can be managed within the framework of the relational database management.

We realize two ways of (1) and (2). That of (3) is not supported. Usually, the size of a fact base determines the number of concurrent activities that can be carried out by the parallel machine. Hence, ways (1) and (2) are oriented toward knowledge base management applications.

In way (1), a query described in a logic program is translated into a sequence of relational operations. The sequence is executed by using the parallel processing scheme based on the functional programming concepts as described in Subsection 3.1.

To support way (2), the mechanism for executing unification of a goal clause with fact clauses is also implemented on the basis of functional programming concepts. The resolutions for rule clauses are performed by a rule reduction system. A query represented as a goal is reduced into AND-literals. Each AND-literal is referred to as a subgoal. A goal is reduced into AND-literals in the rule reduction system until each subgoal requires to be unified with fact clauses. A set of fact clauses are represented as a relation. A predicate name of a fact clause corresponds to a relation name. The facts which have the same predicate name are represented as a set of tuples in a relation. AND-literals reduced from the goal clause are unified with fact clauses in relations. AND-literals can be processed independently as long as no variables are shared among literals. As a result, AND-parallelism is exploited. Between AND-literals sharing free variables, binding environments of variable/value are transferred in the form of a stream.

In our approach, each AND-literal inputs a binding environment and returns a new binding environments as presented in [7] and [8]. In [7] and [8], binding environments are propagated within the framework of the data-driven evaluation. In [7], the eager and lazy evaluations are used to control the stream of binding environments. In our approach, demand-driven evaluation is used for processing fact clauses. The process of solving each literal in AND-literals is regarded as a function that returns a stream of binding environments. Each binding environment represents an alternative solution to the goal clause. Like relational operations, inference operations for a relational database are also performed on the basis of functional programming concepts. If AND-literals are sharing free variables, the input and output of binding environments among those AND-literals are performed using the stream-oriented scheme based on demand-driven evaluation.

For example, a goal clause " <- L1,L2,L3 " is constructed as three function nodes. The output of a function node is served as the input to another. It is clear that the activation of a literal does not have to wait for complete intermediate solutions to be generated by execution of the preceding function node. Therefore, stream-oriented parallel processing can be performed among these functions. In each function node, a literal is unified with fact clauses in a corresponding relation. Each function node is executed on the basis of demand-driven evaluation by using the following algorithm.

(1) When an inference function node receives a

demand from the consumer node, it repeatedly executes (2) and (3) until one page of new binding environments is created and stored in the output buffer.

(2) A single page of binding environments is accessed in one area of the input buffer. At this time, the other area of the input buffer becomes available and a demand is pre-issued to the producer node to refill the area with the subsequent page. As a result, stream-oriented parallel processing is performed between this node and the producer node.

(3) The search and unification to fact clauses represented in the operand relation are executed using the binding environments in the page that has just been accessed in (2), and new binding environments are created and stored in the output buffer. If the output buffer is filled with a page of new binding environments, execution is suspended and the node waits for the next demand from the consumer node, otherwise, (2) and (3) are executed repeatedly. If the page being manipulated is the last one of the binding environments served by the producer node, the execution of this inference node is terminated.

4. System Architecture

In this section, a relational database machine architecture based on functional programming concepts is presented. In this architecture, a sequential processor is used for computing an individual function. The overview of the multiprocessor architecture is shown in Fig. 1. Each processor (RKU or QRU) supports the basic primitives defined in Section 2. The processor includes an internal memory, and the processors are connected via a high-speed interconnection network. The relations in the database are distributedly stored in disk devices. The disk devices are connected to the processors via Staging Buffers. The three-level memory hierarchy employed in this architecture. As discussed in [9], [12], [13], [15] or [19], the three-level memory hierarchy is effective in manipulating relational operations.

Although a sequential processor is currently used to compute an individual function, advanced processors as discussed in [14] and [15] can be attached to each processor in future.

The total internal memory of each processor is not large enough to store a whole source relation or a whole intermediate relation in general. In this architecture, an operand source relation is staged up into a staging buffer as a stream of tuples, using the demand-driven evaluation.

The system consists of the following components:

QRU : QRU decomposes a query into a sequence of relational operations or into a goal clause which consists of AND-literals. QRU is regarded as a rule reduction system. In manipulating a query as a goal clause, the query is reduced into AND-literals until each literal requires to be unified with fact clauses. A query is reduced by unifying with rule clauses in the rule base. QRU allocates each relational or inference operation as a function instance to one of RKU's, and specifies the channels between function instances.

RKU : RKU's carry out relational operations and inference operations on the basis of functional programming concepts. Each RKU supports the basic primitives for realizing demand-driven evaluation and function application, and executes relational or inference operations as functions discussed in Sections 2 and 3. One or more operations can be allocated to each RKU by QRU. If several operations are allocated to a single RKU, they are executed as coroutines within the framework of demand-driven evaluation. This allocation enables the machine to execute a query within the limited number of processors. The execution sequence of the operations allocated to a single RKU is controlled by a scheduler. The scheduler gives control to one of the operations, when the operation has already received a demand and its operand pages have already been prepared in the internal memory. The function node to which control is given executes the operation to the data in the input buffer until the output buffer is filled (put-wait) or the operation to the data stored in the input buffer completes (get-wait). Then, the node suspends execution of function computation and returns control to the scheduler.

Communication Network : Stream data (the sequence of tuples or the sequence of binding environments) and demands are transferred via the Communication Network. Information for allocating function instances and for specifying the channels among function instances is also transferred between QRU and RKU's. The transfers of stream data and demands between processors are managed by Communication Processing Units connected to every processor.

Data Staging Network : Source relations are transferred from disk devices to Staging Buffers via the Data Staging Network. A source relation is stored to

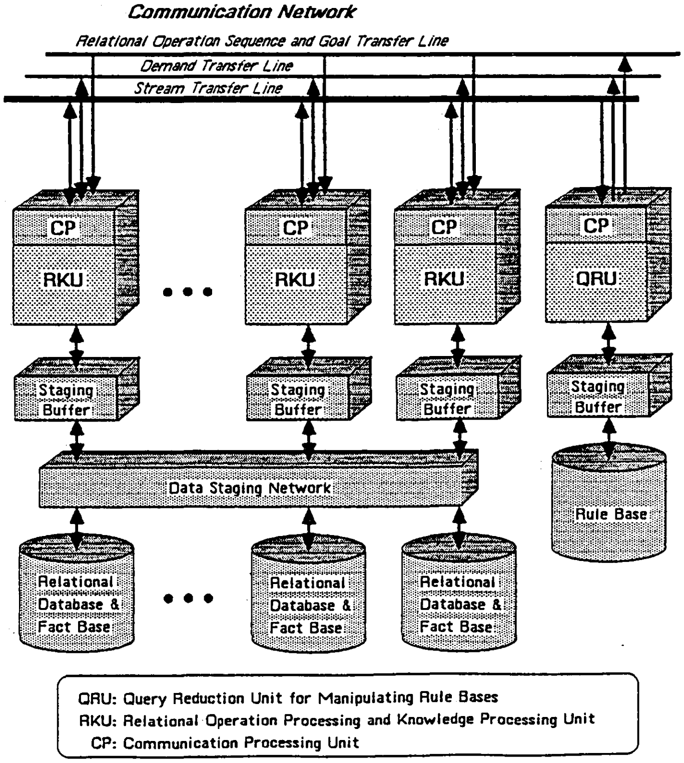

Communication Network
Relational Operation Sequence and Goal Transfer Line
Demand Transfer Line
Stream Transfer Line
CP
RKU
CP
RKU
CP
RKU
CP
QRU
Staging Buffer
Staging Buffer
Staging Buffer
Staging Buffer
Data Staging Network
Rule Base
Relational Database & Fact Base
Relational Database & Fact Base
Relational Database & Fact Base
QRU: Query Reduction Unit for Manipulating Rule Bases
RKU: Relational Operation Processing and Knowledge Processing Unit
CP: Communication Processing Unit

Fig. 1 System architecture

one of the Staging Buffers which is connected to a single RKU. This network supports the allocation of source operand relations to the staging buffer.

5. Experimental Implementation

For simulation experiments of the relational database machine, we have implemented the basic primitives in software, and have developed a relational operation system as described in Sections 2 and 3 on the Sun-2 workstation [23]. Although this system is currently running on a single processor, it can simulate parallel processing environments based on demand-driven evaluation. Each relational operation is defined as a function and executed within the framework of demand-driven evaluation. In this section, experimental results of query execution are shown. The relational operation system can realize various environments of demand-driven evaluation.

5.1 Environments of the Experiment

The query chosen for presenting several environments of demand-driven evaluation is shown in Fig. 2. The query includes four selection operations, three join operations and a projection operation. The cardinality (the number of tuples) of each relation, the tuple length, selection selectivity factors and join selectivity factors are set as shown in Table 1. It is assumed that five processors (RKU's) are used to execute the query. Relational operations are allocated to processors as shown in Fig 2.

The following environments are assumed in the experiments.

(1) The transfer of stream elements is exclusively performed between two processors. While two processors are communicating via the Communication Network, no other processors can use it. The transfer of demands can be simultaneously performed among every processors. The data transfer rate is set to 16.6 milliseconds for a 2k-byte data.

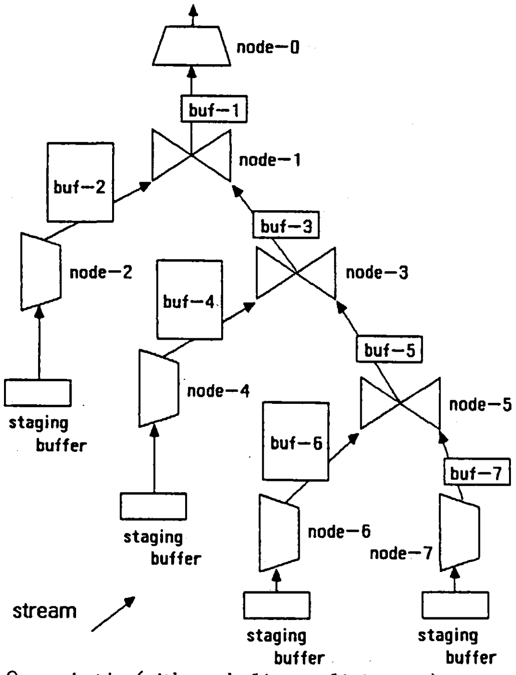
(2) The elements of a stream corresponding to each source operand relation have been already stored in Staging Buffers. Tuples of a source relation are staged up to the Staging Buffer as a stream. Each operand relation is stored in the Staging Buffer connected to the processor which manipulates the operand relation.

5.2 Experimental Results

The query was executed in four environments shown in Table 2. These environments are realized by using basic primitives and by specifying the properties of channels as discussed in Section 2. Experimental results of Environment-1, Environment-2, Environment-3 and Environment-4 are shown in the time charts in Fig. 3, Fig. 4, Fig. 5, and Fig.6, respectively.

In Environment-1, stream-oriented parallelism is not exploited. On the other hand, in Environment-2, parallelism is exploited and performance of query processing is improved. In Environment-3, recomputation is performed in node-7 of the selection operation. In this environment, although recomputation causes heavy overhead, it enables the relational operation system to perform a query even in a case where the whole relation to be processed is not stored in the internal memory of the processor. In comparing Environment-2 with Environment-4, parallelism in Environment-2 is higher than that in Environment-4. In Environment-4, since the relation-level granularity is employed, the stream-oriented parallelism is not exploited. However, the parallelism is exploited by evaluating formal arguments of a function in parallel.

Response times were measured by varying granularity settings in Environment-2 as shown in Fig. 7. In this experiment, only the sizes of inner relation pages in executing the join operations are varied, and the granularity for streams of outer relations is set to the relation-level granularity. The granularity is set to the same value in every join operation in a query. For small granularity, response time is long. This is because demands are issued many times and the number of communication times of stream data increases in the small granularity case. For large granularity, the response time is long.



node-0: projection(without duplicate elimination)
node-1,node-3,node-5: join
node-2,node-4,node-6,node-7: selection
processor-0: node-0 buf-1: operand page of projection
processor-1: node-1,node-2 buf-2,4,6: outer-relation page of join
processor-2: node-3,node-4
processor-3: node-5,node-6 buf-3,5,7: inner-relation page of join
processor-4: node-7

Fig. 2 Query

Table 1 Parameter settings

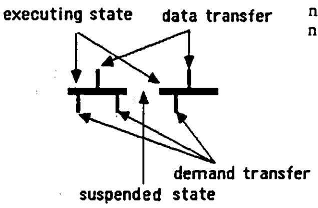
tuple size	64 bytes
operand attribute (integer value)	4 bytes
cardinality (number of tuples) of each source relation	10000 tuples
cardinality of each intermediate relation	1000 tuples
selection selectivity factor(ssf)	0.1
join selectivity factor(jsf)	0.001

(cardinality of result relation):
ssf*(cardinality of source relation) in selection,
jsf*(cardinality of outer relation)
*(cardinality of inner relation) in join.

Table 2 Query processing environments

granularity (buffer size) for input stream: (number of tuples)

Experimental Environment	selection operations				join operations					
	node-2 (staging buffer)	node-4 (staging buffer)	node-6 (staging buffer)	node-7 (staging buffer)	node-1		node-3		node-5	
					outer-relation (buf-2)	inner-relation (buf-3)	outer-relation (buf-4)	inner-relation (buf-5)	outer-relation (buf-6)	inner-relation (buf-7)
Environment-1	page-level (1000)	page-level (1000)	page-level (1000)	page-level (1000)	relation -level (1000)	page-level without stream parallelism (100)	relation -level (1000)	page-level without stream parallelism (100)	relation -level (1000)	page-level without stream parallelism (100)
Environment-2	page-level (1000)	page-level (1000)	page-level (1000)	page-level (1000)	relation -level (1000)	page-level with stream parallelism (100)	relation -level (1000)	page-level with stream parallelism (100)	relation -level (1000)	page-level with stream parallelism (100)
Environment-3	page-level (1000)	page-level (1000)	page-level (1000)	page-level (1000)	relation -level (1000)	page-level with stream parallelism (100)	relation -level (1000)	page-level with stream parallelism (100)	page-level with stream parallelism (500)	page-level with stream parallelism (recomputation) (100)
Environment-4	page-level (1000)	page-level (1000)	page-level (1000)	page-level (1000)	relation -level (1000)	relation -level (1000)	relarion -level (1000)	relation -level (1000)	relation -level (1000)	relation -level (1000)



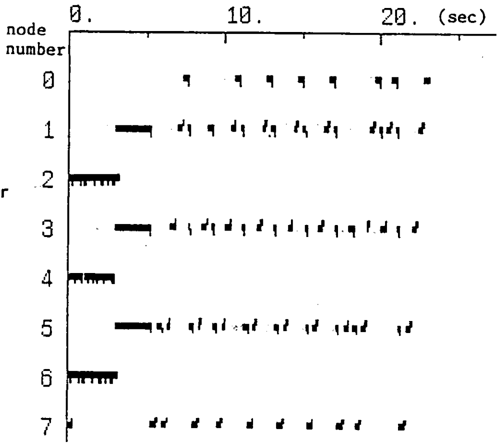


Fig. 3 Time chart (Environment-1)

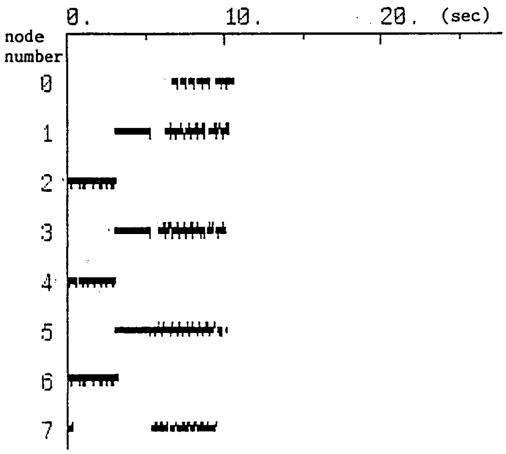


Fig. 4 Time chart (Environment-2)

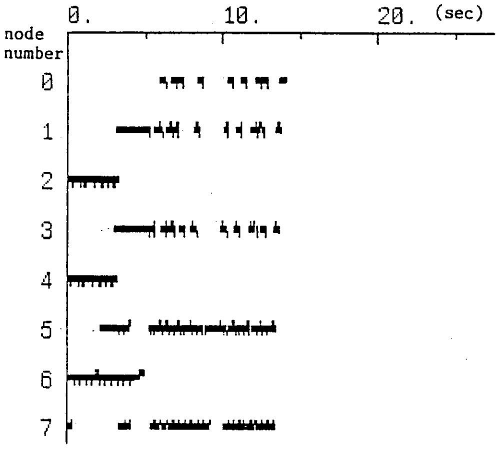


Fig. 5 Time chart (Environment-3)

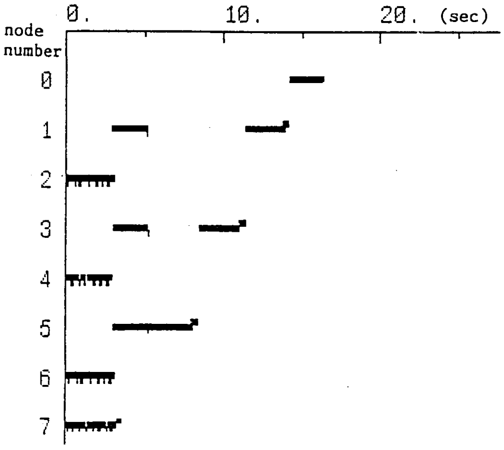


Fig. 6 Time chart (Environment-4)

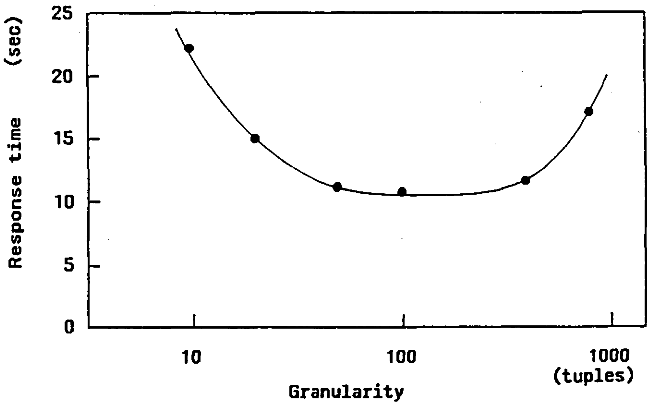


Fig. 7 Response time in various granularity settings

This is because stream-oriented parallelism is not exploited. In the case of large granularity, larger memory space is required in order to store a larger page. Optimal granularity is dependent not only on hardware performance, such as communication speed of the Communication Network, the size of available memory space or processing power of each processor, but also on contents of manipulated data. The proper page-level granularity exploits the highly parallelism within a limited memory resource environment.

5. Conclusions

We have presented a novel approach to a relational database machine for processing knowledge bases. The principle purpose of the approach is to concurrently perform both relational operations and inference operations within the limited resource environment in knowledge base systems. This approach is based on functional programming concepts in order to manage computer resources with the theoretical neatness of functional computation. By using demand-driven evaluation as a driving method of the functional computation, parallelism can be exploited in knowledge base processing. In this paper, we have defined the basic primitives which are used to implement demand-driven evaluation and function application. We have also presented a basic algorithm and a system architecture for executing relational and inference operations by using a demand-driven evaluation mechanism.

In this paper, we have also discussed how the relational database concepts are combined with logic programming concepts. For processing a set of fact clauses, we have presented an algorithm based on stream-oriented parallel processing.

We have developed the relational operation system based on the proposed approach. Currently, we are designing the inference operation system for fact clauses, and also designing the architecture in more detail.

Acknowledgements

The authors would like to thank Dr. Makoto Amamiya and Ryuzo Hasegawa (NTT Electrical Communication Laboratories) for their valuable discussions and suggestions.

References

[1] M. Amamiya and R. Hasegawa, "Dataflow computing and eager and lazy evaluations," New Generation Computing, vol. 2, no. 2, pp. 105-129, 1984.

[2] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra its algebra programs," Comm. ACM, vol. 21, no. 8, pp. 613-641, Aug. 1978.

[3] H. Boral and D. J. DeWitt, "Design considerations for data-flow database machines," in Proc. ACM SIGMOD 1980 Int. Conf. Management of Data, pp. 94-104, May 1980.

[4] E. F. Codd, "A relational model of data for large shared data banks," Comm. ACM, vol. 13, no. 6, pp. 377-387, June 1970.

[5] D. P. Friedman and D. S. Wise, "Aspects of applicative programming for parallel processing," IEEE Trans. Comput., vol. C-27, pp. 289-296, Apr. 1978.

[6] H. Gallaire, J. Minker and J. M. Nicolas, "Logic and databases: a deductive approach," ACM Computing Surveys, vol. 16, no. 2, Jun 1984.

[7] Z. Halim and I. Watson, "An or-parallel data-driven model for logic programs," in Proc. Int. Workshop High-level Computer Architecture, pp. 1.26-1.36, May 1984.

[8] R. Hasegawa and M. Amamiya, "Parallel execution of logic programs based on dataflow concept," in Proc. 1984 Int. Conf. Fifth Generation Computer Systems, pp. 507-516, 1984.

[9] P. B. Hawthoron and D. J. DeWitt, "Performance analysis of alternative database machine architectures," IEEE Trans. Softw. Eng., vol. SE-8, no. 1, pp. 61-76, 1982.

[10] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," in Information Processing 77: Proceedings of IFIP Congress 77, pp. 993-998, Aug. 1977.

[11] R. M. Keller and M. R. Sleep, "Applicative caching," in Proc. ACM Conf. Functional Programming Lang. Comput. Arch., pp. 131-140, 1981.

[12] N. Kamibayashi and K. Seo,"SPIRIT-III : an advanced relational database machine introducing a novel data-staging architecture with tuple stream filters to preprocess relational algebra," AFIP Proc. NCC 1982, 1982.

[13] M. Kitsuregawa, H. Tanaka and T. Moto-oka, Application of hash to data base machine and its architecture," New Generation Computing, vol. 1,pp.63-74, 1983.

[14] Y. Kiyoki, K. Tanaka, N. Kamibayashi and H. Aiso, "Design and evaluation of a relational database machine employing advanced data structures and algorithms," in Proc. 8th Int. Symp. Computer Architecture, pp. 407-428, May 1981.

[15] Y. Kiyoki, M. Isoda, K. Kojima, K. Tanaka, A. Minematsu and H. Aiso, "Performance analysis for parallel processing schemes of relational operations and a relational database machine architecture with optimal scheme selection mechanism," in Proc. 3rd Int. Conf. Distributed Computing Systems, Oct. 1982.

[16] Y. Kiyoki, R. Hasegawa and M. Amamiya, "An execution scheme for relational database operations with eager and lazy evaluations," Trans. IPSJ, vol. 26, no. 4, pp. 685-695, July 1985 (in Japanese).

[17] Y. Kiyoki, R. Hasegawa and M. Amamiya, "A stream-oriented parallel processing scheme for relational database operations," to appear in Proc. 1986 Int. Conf. Parallel Processing, 1986.

[18] D. Li, "A Prolog Database System," Research Studies Press., 1984.

[19] K. Seo, N. Kamibayashi, A. Minematsu and H. Aiso, "A look-ahead data staging architecture for relational database machines," Proc. 8th Int. Symp. Computer Architecture, pp.389-406, May 1981.

[20] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins, "Data-driven and demand-driven computer architecture," ACM Computing Surveys, vol. 14, no. 1, Mar. 1982.

[21] S. R. Vegdahl, "A survey of proposed architectures for the execution of functional languages," IEEE Trans. Comput., vol. C-33, no. 12, pp. 1050-1071, Dec. 1984.

[22] H. Yokota and H. Itoh, "A model and an architecture for a relational knowledge base," Proc 13th Int. Symp. Computer Architecture, June 1986.

[23] Programmers Reference Manual for the Sun Workstation, Sun Micro System, Inc., 1982.

KNOWLEDGE-BASED EXPERT SYSTEM FOR HARDWARE LOGIC DESIGN

Tamio Mano, Fumihiro Maruyama, Kazushi Hayashi, Taeko Kakuda,
Nobuaki Kawato, and Takao Uehara

FUJITSU LIMITED
1015 Kamikodanaka, Nakahara-ku
Kawasaki 211, Japan

ABSTRACT

We have developed a knowledge-based expert system for hardware logic design that designs CMOS circuits from a concurrent algorithm written in the occam *high-level programming language. This work was done as part of the activities of the Fifth Generation Computer Systems (FGCS) Project of Japan. Our system aims at supporting the entire design process from a specification to completed CMOS circuits by incorporating the designers' expertise into the computer and utilizing it effectively. Prolog was selected as the implementation/knowledge-representation language. This paper gives examples of the knowledge provided for this system, emphasizing the functional design phase which is heavily dependent on the designers' expertise. This paper also describes how Prolog was used to express this knowledge, and how the inference engine was created. While constructing this system, we also evaluated the effectiveness of Prolog as the implementation language for a new generation of CAD systems.*

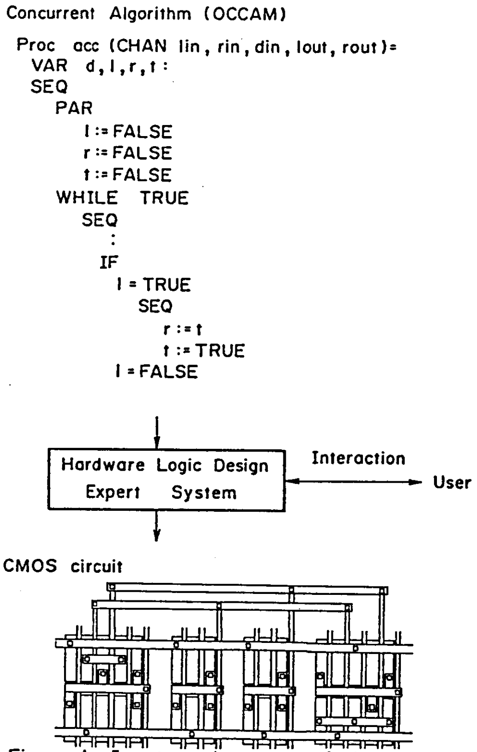
1. INTRODUCTION

The *FGCS* Project has been concerned with research into knowledge-based systems. We chose hardware logic design as an application for the following three reasons. First, the application must be an area in which human expertise plays a significant role. At present, reliable and efficient hardware logic design can only be done by skilled designers. Therefore, this area is a suitable testing ground for the use of expert knowledge. Secondly, because hardware logic design is synthesis-oriented, which is quite unlike any successfully developed analysis-oriented system such as MYCIN, there are many unknown factors to investigate. Lastly, since much must be considered in hardware logic design, a wide range of knowledge is required.

The implementation language is Prolog, which is also used as the underlying knowledge-representation language. Knowledge representation is an important issue. Hardware logic design employs various areas of knowledge, and design data must be represented as well. In addition, human designers switch from one representation to another in the course of their work. For these reasons, we have not adopted any particular existing tool for knowledge representation.

2. SYSTEM OVERVIEW

Our system aims at supporting the entire design process from a specification to completed CMOS circuits by incorporating the designers' expertise into the computer [1][2][3]. This system designs CMOS circuits that conform to the input specification through the interaction with the user, as shown in Figure 1. The input specification is a concurrent algorithm described in occam[4] which is a programming language characterized by its treatment of concurrency. The input specification is based on software concepts, such as variables and procedure calls, but not hardware concepts, such as registers and clocks. Therefore, with occam, a designer can create a concurrent algorithm without being familiar with hardware details.



Concurrent Algorithm (OCCAM)

```
Proc acc (CHAN lin, rin, din, lout, rout)=
  VAR d,l,r,t :
  SEQ
    PAR
      l := FALSE
      r := FALSE
      t := FALSE
    WHILE TRUE
      SEQ
       :
        IF
          l = TRUE
            SEQ
              r := t
              t := TRUE
          l = FALSE
```

Figure 1. Input and output of the system

CH2345-7/86/0000/0979$01.00 © 1986 IEEE

Given a concurrent algorithm, the system performs functional design using designer's expertise and interacting with the user to determine the hardware behavior. The hardware concepts, such as registers and clocks, first emerge in this intermediate design stage. Next, the system designs CMOS circuits to achieve the hardware behavior determined by the functional design process. The system outputs the CMOS cells in the library and CMOS functional cells, and their connections. Another program should be used for placement and routing of these cells on a chip.

Our system consists of a functional design phase and a circuit design phase, as shown in Figure 2. Because the system is divided into these two phases, the system is able to flexibly cope with different semiconductor technologies. The functional design phase, the first half of the design process, determines the application of hardware concepts in the implementation of a concurrent algorithm in **occam**, and produces the finite-state machine description in DDL[5] using designers' expertise related to the functional design for hardware. Although hardware behavior is determined at this step, it is independent of the semiconductor technology being employed. The circuit design phase, the second half of the design process, transforms the finite-state description into the desired CMOS circuits. To achieve this, the translator subsystem extracts and rearranges information required for the circuit design from DDL. The functional cell design subsystem designs combinational circuits, and the functional block design subsystem allocates cells in the library to functional blocks such as registers and memories. In the circuit design phase, technology-dependent knowledge related to circuit synthesis is used.

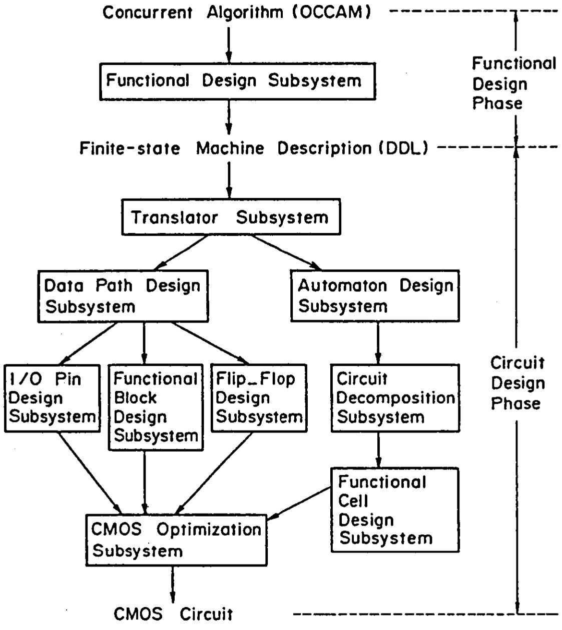

Concurrent Algorithm (OCCAM)
Functional Design Subsystem
Functional Design Phase
Finite-state Machine Description (DDL)
Translator Subsystem
Data Path Design Subsystem
Automaton Design Subsystem
I/O Pin Design Subsystem
Functional Block Design Subsystem
Flip_Flop Design Subsystem
Circuit Decomposition Subsystem
Circuit Design Phase
Functional Cell Design Subsystem
CMOS Optimization Subsystem
CMOS Circuit

Figure 2. System configuration

3. DESIGN EXAMPLE

This section discusses how the system works, taking a pattern matcher proposed by M.J.Foster and H.T.Kung[6], as an example. Figures 3 and 4 show the outline of the pattern matcher. This pattern matcher checks whether a given pattern, which is a fixed length vector of characters, is embedded in a given text string, which is an endless string of characters, as shown in Figure 3.

Let us denote the input string stream as $s_0s_1s_2\ldots$, the input finite pattern stream as $p_0p_1p_2\ldots p_k$, and the output result stream as $r_0r_1r_2\ldots$. Characters in the two input streams may be compared for equality, with the wild card character X matching any character in an input stream. The output bit r_i is to be set to 1 if the substring $s_{i-k}s_{i-k+1}\ldots s_i$ matches the pattern, and 0 otherwise. For example, in Figure 4 the pattern AXC matches substrings $s_0s_1s_2$, $s_3s_4s_5$, and $s_4s_5s_6$ (ABC, AAC, and ACC, respectively).

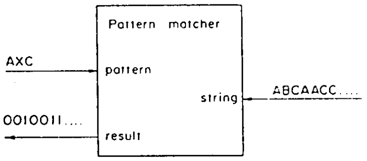


Figure 3. Data to and from the pattern matcher

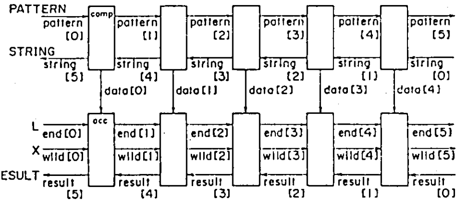


Figure 4. Dataflow of the pattern matcher

3.1 Design Specification

The concurrent algorithm of the pattern matcher, which is input to this system, is described in **occam**, as shown in Figure 6.

The **occam** program consists of three declaration parts, (A),(B), and (C), and a description of parallel processes, (D). The declaration parts are as follows.

(A) declares channel vectors. A channel vector is a set of channels, these being used for communication between concurrent processes. For example, pattern[6] means that there are six channels named pattern, and they are numbered from 0 to 5, as shown in Figure 4.

(B) declares a single comparator process. PROC gives the name *comp* to this process, and identifies five formal parameters, the internal channels, *pin*, *sin*, *pout*, *sout*, and *dout*, as shown in Figure 5. When the named process is substituted for the subsequent process (D), the formal parameters are replaced by the actual parameters. Process *comp* is a sequential process, which consists of two processes. One is the initialization (B-1), and the other is an endless

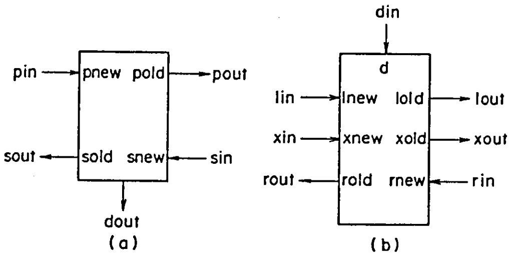


Figure 5. Formal parameters and variables
(a) Comparator (b) Accumulator

```
CHAN  pattern [6]:                                        ┐
CHAN  string [6]:                                         │
CHAN  data [5]:                                           │ (A)
CHAN  end [6]:                                            │
CHAN  wild [6]:                                           │
CHAN  result [6]:                                         ┘
PROC  comp (CHAN pin, sin, pout, sout, dout)=             ┐
 VAR  pnew, pold, snew, sold:                             │
 SEQ                                                      │
   PAR                                                    │
     pold:=0                                ┐ (B-1)       │
     sold:=0                                ┘             │
   WHILE  TRUE                                            │
     SEQ                                                  │ (B)
       PAR                                  ┐             │
         pin  ?  pnew                       │             │
         pout  !  pold                      │ (B-2)       │
         sin  ?  snew                       │             │
         sout  !  sold                      ┘             │
     PAR                                                  │
         pold := pnew                       ┐             │
         sold := snew                       │ (B-3)       │
         dout  !  pnew = snew:              ┘             ┘
PROC acc (CHAN xin, lin, rin, din, xout, lout, rout)=    ┐
 VAR  d, xnew, xold, lnew, lold, rnew, rold, t:           │
 SEQ                                                      │
   PAR                                      ┐             │
     xold := FALSE                          │             │
     lold := FALSE                          │ (C-1)       │
     rold := FALSE                          │             │
     t    := TRUE                           ┘             │
   WHILE  TRUE                                            │
     SEQ                                                  │
       PAR                                  ┐             │
         xin  ?  xnew                       │             │
         xout  !  xold                      │             │
         lin  ?  lnew                       │ (C-2)       │ (C)
         lout  !  lold                      │             │
         rin  ?  rnew                       │             │
         rout  !  rold                      │             │
         din  ?  d                          ┘             │
       PAR                                  ┐             │
         xold := xnew                       │             │
         lold := lnew                       │             │
         IF                                 │             │
          lnew = TRUE                       │             │
            SEQ                             │ (C-3)       │
              rold := t                     │             │
              t = TRUE                      │             │
          lnew = FALSE                      │             │
            PAR                             │             │
              rold := rnew                  │             │
              t := t /\ (xnew \/ d):        ┘             ┘
PAR  i = [1 FOR 5]                                        ┐
 PAR                                                      │
   comp (pattern [i-1], string [5-i], pattern [i],        │ (D)
                       string [6-i], data [i-1])          │
   acc (wild [i-1], end [i-1], result [5-1], data [i-1],  │
                      wild [i], end [i], result [6-i])    ┘
```

Figure 6. Algorithm for the pattern matcher in OCCAM

iterative process, WHILE TRUE. The iterative process contains a sequential process, which consists of two processes. The first two are output and input processes (B-2). The last process compares the characters in the pattern with those in the text string and outputs a Boolean value, TRUE or FALSE (B-3).

(C) declares a single accumulator process. The process named *acc* contains seven formal parameters, as shown in Figure 5. The variable *d* stands for the current comparison result of the comparator, *xnew* and *xold* for the don't care bit, *lnew* and *lold* for end of pattern, *rnew* and *rold* for final result of matching, and *t* for temporary result of matching. Process *acc* is a sequential process, which consists of two processes. One is the initialization of variables (C-1). The other is an iterative process, similar to process *comp*. The iterative process includes a sequential process, which consists of two processes. The first includes output and input processes (C-2). The second is a conditional process (C-3). When the end of pattern is reached, an accumulator uses the value *t* (current temporary result) as the final result, and then resets *t* to TRUE. Otherwise, it maintains a temporary result *t*, which is set by the logical expression $t := t \wedge (xnew \vee d)$. $\wedge$ and $\vee$ stand for boolean AND and OR, respectively. Thus if the current temporary result *t* is TRUE, and *xnew* or the current comparison result *d* is TRUE, then the new temporary result will be set to TRUE.

(D) indicates a 2x5 array of concurrent processes, as shown in Figure 4. The cells at the top are the comparators; the pattern flows from left to right and the string flows from right to left. The bottom cells, accumulators, receive the results of the comparison from above. They maintain partial results, and shift completed results from right to left. Two bits associated with the pattern flow through the accumulators from left to right. One bit is the end of pattern, L . The other is the wild card character, X.

3.2 Functional Design Phase

In the functional design phase, the finite-state machine description in DDL is produced from a given concurrent algorithm. Functional design can be thought of as that phase of the design that determines which hardware concepts to apply to the implementation of the concurrent algorithm and describes how the hardware should behave. This phase is one of the most knowledge-intensive parts, and its performance depends heavily on the knowledge which is used. This phase continues with the following procedures through the interaction with the user, as shown in Figure 7.

(1) Analysis of the structure of the occam specification. Taking the concurrent descriptions into account, it analyzes the occam construct(WHILE, SEQ, IF, etc.) and creates an outline of the hardware control mechanism (state transitions).

(2) Implementation of variables described in occam, using hardware elements(registers, terminals, etc.). Because the occam specification does not provide information about the number of required bits, the system queries the user. However, the system automatically infers that the number of bits for a variable is one if all of its sources turn out to be Boolean values ("TRUE","FALSE", or the evaluation of a logical expression).

(3) Compression of the operation sequences. This is an

```
yes
I ?-functional_design.
 Parsing your specifications in OCCAM....
 Implementing OCCAM variables....
              :
 Should variable rold  have only one bit?y/n
I:y
 How many bits should variable snew have ?
I:8
 Should variable pnew have as many bits as variable snew?y/n
I:y
 Should variable pold have as many bits as variable pnew?y/n
I:y
 Should variable sold  have as many bits as variable snew?y/n
I:y
              :
 Compressing a sequence of operations....
 Implementing inter-process communication.....
              :
 Optimizing....
 Enter the name of the register for rnew and rold.
I:r
              :
 Generating partial DDL descriptions....
 Constructing the final DDL code from partial DDL descriptions.
yes
I?-
```

Figure 7. Interaction with the user

```
<system> pm.
 <time>  clk.
 <entrance> pin(8), sin(8), xin, lin, rin, din.
 <exit> pout(8), sout(8), dout, xout, lout, rout.
 <terminal> sendl.
 <automaton> comp:clk:
  <register> p(8), s(8).
  <states>
    init: p←0, s←0, →idle.
    idle: pout=p, sout=s,
           p←pin, s←sin, →state2.
    state2: sendl=1, dout=(p:=s),→idle.
  <end>.
 <end> comp.
 <automaton> acc:clk:
  <register> d, x, l, r, t.
  <states>
    init: x←0, l←0, r←0, t←1,→idle.
    idle: sendl: xout=x, lout=l, rout=r,
                 x←xin, l←lin, r←rin,
                 d←din, →statel.
    statel: |* l *|[r←t, t←1]
                  ; t←(t & (x | d)).,→idle.
  <end>.
 <end> acc.
<end> pm.
```

Figure 8. Final DDL description

attempt to transform some occam sequential processes into DDL register transfer operations executed in parallel, which improves the performance of the generated hardware.

(4) Implementation of the communication between processes described in occam("?" for inputting a value from a channel, "!" for outputting a value to a channel). Basically the communication is implemented by hand shaking. Signal lines through which data is transferred and signal lines used for synchronization are provided.

After implementing occam primitive operations as DDL hardware operations and generating partial DDL descriptions, the system puts these partial descriptions together to complete the final DDL description, as shown in Figure 8.

3.3 Circuit Design Phase

The translator subsystem transforms the DDL finite-state machine description into information to be used by the circuit design process. It gathers and edits conditions for terminal connection, register transfer, and state transition operations. Then the translator subsystem organizes the data in frame-like structures, classified into twelve categories: system, clock, automaton, input signal, output signal, terminal, memory, register, state, decoder, arithmetic expression, and logical expression.

The DDL code shown in Figure 8 contains three register transfer operations of register "r" in automaton "acc". These operations provide the following information:

1) $| * \ acc\_init \ * | \quad r \leftarrow 0.$

2) $| * \ acc\_idle \ \& \ send1 \ * | \quad r \leftarrow rin.$

3) $| * \ acc\_state1 \ \& \ l \ * | \quad r \leftarrow t.$

where the states whose identifiers are modified by "acc" belong to the automaton "acc". "| * * | ..." stands for an if-clause. Based on the above information, the input circuit of register "r" is designed, as shown in Figure 9.

The automaton design subsystem implements automata having the appropriate states using flip-flops. It designs a control circuit around these flip-flops according to information on state transitions provided by the translator subsystem. Figure 10 shows the resulting circuit design for a pattern matcher that consists of a pair of the comparator and the accumulator.

The size of a combinational circuit that can be created using a single CMOS cell is limited by CMOS technology. Signal delay depends mainly on the number of FETs inserted in series between the power supply line and output line. Therefore, the circuit decomposition subsystem decomposes the combinational circuits so that the number of FETs in series does not exceed the limit imposed by signal delay considerations. The combinational circuit around register "r" is decomposed into four parts as shown in Figure 9.

The functional cell design subsystem implements the decomposed combinational circuits as functional CMOS cells. The optimal layouts of the functional CMOS cells are obtained by a heuristic algorithm[7]. However, if the designer is required to use NAND or NOR gates, then the partial combinational circuit (iv) in Figure 9 results in mediocrity as shown in Figure 11(a). In this case, we take advantage of the property of CMOS functional cells that physically adjacent gates can be connected by a diffusion area. As a result, the optimal layout is obtained as shown

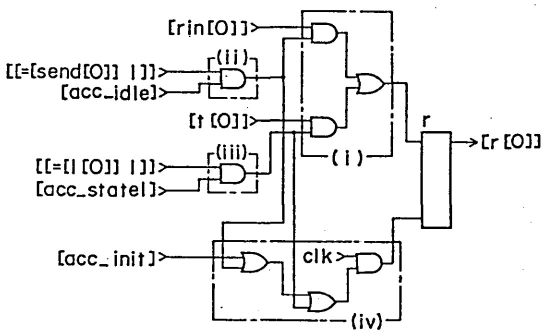


Figure 9. Input circuit of register r

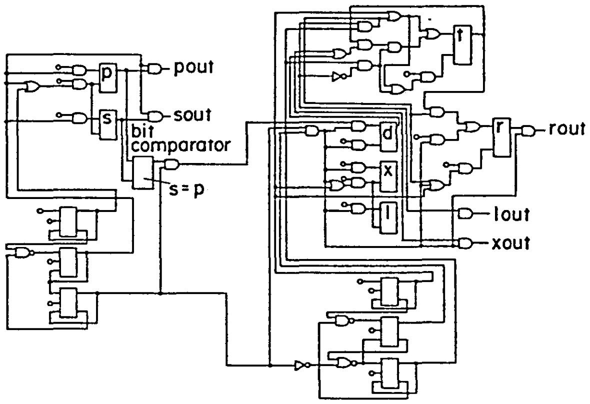
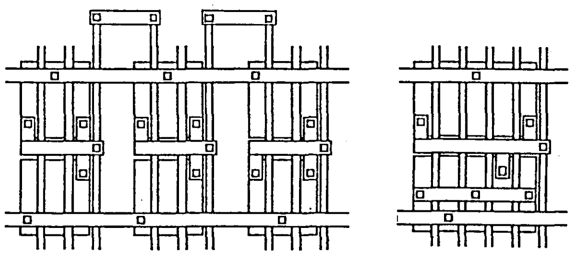


Figure 10. Logic diagram of pattern matcher



(a) NAND gates (b) Functional cell

Figure 11. Implementation of the circuit (iv) in Figure 9.

in Figure 11(b). Note that the optimal array is almost 50% smaller than the basic conventional array.

The functional block subsystem allocates cells in the library to functional blocks, such as registers, memories, decoders, and adders. Since functional cells and cells in the library are of the same height and have the same power connections and standardized connection points, they can be laid out by an existing physical design CAD system.

4. INFERENCE MECHANISM AND KNOWLEDGE

The functional design is heavily dependent on the designers' own expertise. It is difficult to automate the functional design by conventional algorithmic CAD techniques. We automated the functional design by incorporating the designers' expertise into the computer.

This section discusses how the inference mechanisms were created and gives examples of the designer's expertise provided for the functional design phase. We also evaluate the effectiveness of Prolog as an implementation/knowledge-representation language for a knowledge-based logic design system.

4.1 Characteristics of Functional Design

We must consider several points when performing functional design by a knowledge-based approach. The designer gradually refines the design while checking the specifications and clarifying the entire circuit. At each design step, the designer evaluates the design development and makes appropriate decisions. Therefore, it is extremely important in design to always have a clear understanding of the relationship among design objects. The knowledge-representation framework should reflect this fact. A good method must be provided for referring to the relations among design objects.

Another important issue is controlling the order of design jobs. With several similar jobs, which is placed first is significant, because one job may make another job unnecessary or at least make it easier. Therefore, the implementation language should be able to easily express the control of design jobs.

4.2 Inference Mechanism

4.2.1 Forward chaining

Forward chaining is used as the basic inference mechanism. Logic design is a process of incremental refinement. Incremental refinement comes from successive design decisions; every time a decision is made, the current design development is changed. Design development is represented as the information in the working memory. Chainging the current design development is performed by updating the information in the working memory.

Forward chaining seems to be able to simulate the incremental refinement design process. When the conditions set forth by the premise section of a forward chaining rule are satisfied, the conclusion section is activated and the working memory is updated. Then another rule is activated, referring to the updated working memory. Processing continues in this manner. We use the assert(addition of fact) and retract(deletion of fact) predicates of Prolog for updating the working memory. The working memory consists of design information, which is represented by about 40 types of Prolog fact. As the design process continues, the working memory is gradually filled with design information.

Figure 12 shows the general format of the forward chaining rule. When conditions 1, ..., n are satisfied in the premise section, actions 1, ..., m are activated in the conclusion section. The working memory is updated while causing side effects by the assert and/or retract predicates.

```
(Process):-
   (Condition 1),        --------┐
        .                        |
        .                        ├-- Premise  section
                                 |
   (Condition n),        --------┘
   (Action 1),           --------┐
        .                        |
        .                        ├-- Conclusion  secion
                                 |
   (Action  m).          --------┘
```

Figure 12. General format of the forward chaining rule

```
(Condition 0):-     ----------- Conclusion section
     (Condition 1), -------┐
          :                ├-- Premise  section
     (Condition n), -------┘
```

Figure 13. General format of the backward chaining rule

4.2.2 Backward chaining

Backward chaining is used for checking various conditions. In checking the conditions set forth in the premise section of a forward chaining rule, only reference to the working memory is required in some cases. In other cases however, several inference steps are required. When a specified condition is checked, the backward chaining rule whose conclusion section matches the specified condition is fired. Then, the various conditions set forth by the premise section of the fired rule are checked. Backward chaining is already implemented as the execution mechanism of Prolog.

Figure 13 shows the general format of the backward chaining rule. The head of the Prolog clause contains the conclusion, while the body contains the premise section. When conditions 1, ..., n are satisfied in the premise section, condition 0 is concluded. In backward chaining, the invocation of rules does not affect the working memory.

4.3 Knowledge Representation

In this section, specific knowledge examples of the functional design phase are presented to show how Prolog is used to express designer's expertise. Functional design can be regarded as the process that determines hardware concepts according to specifications in occam. Thus, the knowledge used in this process primarily associates specifications described as software with hardware concepts.

4.3.1 Knowledge for implementing a variable with hardware elements

Figure 14 shows the knowledge used to implement a variable used in occam descriptions with hardware elements. This is a forward chaining rule for implementing a occam variable *Var* and states that:

IF:

(1) There exists the input sources to the *Var* from the external processes through the channel, and

(2) input sources are all Boolean values, and

(3) there exists assigned sources to the *Var*, and

(4) assigned sources are all Boolean values, and

(5) other conditions are satisfied,

THEN:

(6) Store in the working memory the information stating that variable *Var* is to be implemented as a 1 bit register.

```
implement_variable (Var,_):-
   input_source (Var, Input_sources),               ----- (1)
   truth_value (Input_sources),                     ----- (2)
   assigned_source (Var, Assigned_sources),         ----- (3)
   truth_value (Assigned_sources),                  ----- (4)
        :
        .                                           ----- (5)
   assert (implementation (Var, 1, register,...)).  ----- (6)
```

Figure 14. Knowledge for implementing a variable with hardware elements (Forward chaining rule)

This predicate is activated with the variable name used in occam as its first argument. If conditions(1)–(5) are all satisfied, fact(6), stating that *Var* must be implemented as a 1 bit register, is written into the working memory. This rule is used to implement, for example, variable t in Figure 6.

input_source(*Var*, *Input_sources*) is a procedure call. What it means becomes clearer, according to the declarative reading of Prolog clauses; it reads "input source of *Var* is *Input_sources*".

4.3.2 Knowledge for compressing sequential operations

Figure 15 shows the knowledge used to check whether the sequential operations described in occam can be transformed into hardware operations executed in parallel, which would improve the performance of the generated hardware. This is a backward chaining rule and states that:

IF:

(1)(2) Both are store-type operations such as assignment processes, and

(3) they are processed one after another, and

(4) the variables into which the sources are to be stored are different, and

(5)(6) both variables are to be implemented as registers, and

(7) the variable in the first operation is not referred to in the source of the second operation,

THEN:

two operations in a sequence are compatible, or can be executed simultaneously.

This predicate is activated with two operations in **occam** as its arguments. Backward chaining is performed as follows. The conditions *store_operation* and *implementation* are resolved only by referring to the working memory, but the condition *referred_to* activates other rules. When conditions (1)–(7) are all satisfied, the compatibility check is successful.

Using this rule, the **occam** sequential process within the dashed box in Figure 6 is compressed into parallel register transfer operations in DDL, shown within the dashed box in Figure 8. Here, it is noted that the variables *rold* and *rnew* are implemented as a single register r, using the knowledge related to merging two registers into one.

The rule "*compatible*" expresses the relationship between two operations, and Prolog predicates provide a natural means of expressing such relations.

```
compatible (Operation 1 , Operation 2):-
 store_operation (Operation 1, Var 1, assign, Source 1 ,....) ---- (1)
 store_operation (Operation 2, Var 2, assign, Source 2,....) ---- (2)
 followed_by (Operation 1, Operation 2),                      ---- (3)
 Var 1  \== Var 2,                                            ---- (4)
 implementation (Var 1 , _, register,....),                   ---- (5)
 implementation (Var 2, _, register,....),                    ---- (6)
 not (refered_to (Var 1, Source 2)).                          ---- (7)
```

Figure 15. Knowledg for compressing sequential operations (Backward chaining rule)

4.3.3 Knowledge involving local job control

Figure 16 illustrates another forward chaining rule for implementing a variable in **occam** with hardware elements. In particular, this rule changes the order of design jobs locally. The idea of the predicate *implement_variable* is as follows; when no clues are provided regarding the number of bits for a variable, and a "similar" variable exists, try to determine the number of bits for the "similar" one first and use the result. Here, a "similar" variable is defined as one that stores the same type of data. To prevent falling into a loop while a decision about a specific variable is postponed, the list of all postponed variables is stored as the second argument of the predicate, *implement_variable*. This rule states that:

IF:

(1) There exists a variable which looks similar to *Var*, and

(2) the implementation of *Another_var* has not been postponed, and

(3) *Var* is added to the list of postponed variables and *Another_var* is implemented first, and

(4) the number of bits of *Another_var*, *Bit_width* is confirmed, and

(5) other conditions are satisfied,

THEN:

(6) Store in the working memory the information stating that the variable *Var* is to be implemented as a *Bit_width* bit register.

```
mplement_variable (Var, Task_list):-
 looks_similar (Var, Another_var ),                            ---- (1)
 not (member ( Another_var, Task_list)),                       ---- (2)
 implement_variable (Another_var,[Var | Task_list]),          ---- (3)
 implementation (Another_var, Bit_width, .... ),               ---- (4)
                 .
                                                               ---- (5)
                 .
 assert (implementation (Var, Bit_width, register,..)).        ---- (6)
```

Figure. 16. Knowledge involving local control (Forward chaining rule)

For example, this rule is applicable to variables *rold* and *t* which appear in PROC acc in Figure 6. The following assignment processes concern variable *rold* and *t*.

$$rold := t$$

$$t := TRUE$$

$$\vdots$$

$$rold := rnew$$

$$t := t \wedge (xnew \vee d)$$

The system attempts to implement variable *rold* first, but this design job is postponed because there are no clues to the number of bits. Next, the system tries to implement the similar variable *t*. In this case, *t* takes only Boolean values (TRUE or FALSE). Therefore, the system implements variable *t* as a one-bit register. After the implementation of variable *t*, it becomes clear that the variable *rold* should have only one bit, because *rold* has the same number of bits as *t*. In this way, the regular flow of control is altered by changing the order of jobs locally.

4.4 Effectiveness of Prolog

We evaluated the effectiveness of Prolog as an implementation/knowledge-representation language for a new generation of CAD systems.

While constructing this system, we confirmed that the following Prolog characteristics are useful in the construction of a knowledge-based expert system for logic design.

(1) The "predicate represents inter-object relationship" characteristic of Prolog is very efficient for expressing

relationships among hardware concepts, and software concepts.

(2) The characteristic that Prolog code can be declaratively read gives us a better understanding of the rules represented in Prolog.

(3) Controlling the order of design jobs, which requires complex operations for procedural programming languages, is facilitated by using the Prolog execution mechanism.

5. SYSTEM IMPLEMENTATION AND PERFORMANCE

This system was developed using C-Prolog and runs on the VAX-11/780. The C-Prolog interpreter of the VAX-11/780 runs at approximately 1 kLIPS. Table 1 lists the program sizes of various subsystems and the execution times of the two hardware design examples (CPU time of VAX-11/780). Design example 1 applies to the pattern matcher. Design example 2 applies to the simple microprocessor. The input **occam** specifications were about 40 and 50 lines, respectively. The functional design phase took 37 seconds of CPU time to generate the DDL description of the pattern matcher and 13 seconds for the microprocessor. The circuit design phase created the CMOS circuit of the pattern matcher in 11.6 minutes and that of the microprocessor in 13.3 minutes. These circuits correspond to approximately 350 gates for the pattern matcher and 1000 gates for the microprocessor.

6. CONCLUSIONS

We discussed a knowledge-based expert system for hardware logic design implemented in Prolog, with an emphasis on the inference mechanism and knowledge representation of the functional design phase.

Areas requiring further research are as follows.

First, the working memory needs to be appropriately structured. The working memory currently used in this system is a flat-type that consists only of Prolog facts. All procedure calls that deal with the working memory (Prolog's assert and retract predicates are used to update the data) are contained in rules. As a result, the procedure calls makes rule expression too complicated. The use of ESP[8] is a possible solution to the problem of structuring the working memory and separating these procedure calls from rules. ESP provides us with time-dependent states and a frame-like structure while retaining essential logic programming language features.

Secondly, the current system does not have a rule-interpreter on Prolog. As previously mentioned, the local control flow can be modified by changing the local job sequence without a rule-interpreter. However, global job control seems to be required to avoid waste design jobs and to increase the performance of the system. We are planning to introduce a meta-rule interpreter to achieve global control.

Lastly, expert system capability largely depends on the amount of stored knowledge. To improve this, expert knowledge must be repeatedly extracted and stored in the knowledge-base. Therefore, we believe that research into knowledge acquisition is necessary.

ACKNOWLEDGEMENTS

This work is based on the results of the R & D activities of the Fifth Generation Computer Systems Project. The authors would like to thank Dr. K. Furukawa and Mr. Y. Iwashita of ICOT (Institute for New Generation Computer Technology) for their encouragement and support.

Table I. Program size and execution time

	Program size (K step)	Execution time (sec)	
		Example I	Example 2
Functional Design	1.9	37.0	13.0
Circuit Design			
Translator	0.5	4.0	3.4
Automaton Design	2.1	7.4	10.9
Data Path Design	1.4	9.1	11.1
I/O Pin Design	0.1	4.5	0.2
Functional Block Design	0.2	0.1	1.0
Flip_Flop ,Design	0.2	5.4	15.0
Circuit Decomposition	1.2	570.6	611.7
Functional Cell Design	0.8	96.2	142.9
CMOS Optimization	0.1	1.6	0.8
Total	8.5	735.9	810.0

REFERENCES

[1] Maruyama,F., Mano,T., Hayashi,K., Kakuda,T., Kawato,N. and Uehara,T., "Prolog-Based Expert System for Logic Design," International Conference on Fifth Generation computer Systems 1984(FGCS'84), pp.563-571, Nov. 1984.

[2] Mano,T., Maruyama,F., Hayashi,K., Kakuda,T., Kawato,N. and Uehara,T., "**occam** to CMOS Experimental Logic Design Support System," 7th Computer Hardware Description Languages and their Applications (CHDL 85), pp.381-390, Aug. 1985.

[3] Maruyama,F., Mano,T., Hayashi,K., Kakuda,T., Kawato,N. and Uehara,T., "Logic Design: Issues in Building Knowledge-Based Design Systems," Expert Systems(to appear).

[4] Taylor,R. and Wilson,P., "**occam**: Process-oriented language meets demands of distributed processing," Electronics, Nov. 30, 1983.

[5] Dietmeyer,D.L., "Logic Design of Digital Systems," Allyn and Bacon, 1971.

[6] Foster,M.J. and Kung,H.T., "Design of Special-Purpose VLSI Chips: Example and Opinions," CMU-CS-79-147, 1979.

[7] Uehara,T. and vanCleemput,W.M., "Optimal Layout of CMOS Functional Arrays," IEEE Transactions on Computers, vol.C-30, no.5, May 1981.

[8] Chikayama,T., "Unique Features of ESP", International Conference on Fifth Generation Computer Systems 1984(FGCS'84), pp.292-298, Nov. 1984.

RESEARCH ACTIVITIES ON NATURAL LANGUAGE PROCESSING OF THE FGCS PROJECT

Toshio Yokoi, Kuniaki Mukai, Hideo Miyoshi, Yuichi Tanaka

Institute for New Generation Computer Technology(ICOT)
Mitakokusai Building 21F.
1-4-28, Mita, Minato-ku, Tokyo 108, Japan

ABSTRACT

The research activities on natural language processing of the FGCS Project are presented. Linguistic phenomena are formalized in terms of complex structures and constraints on them. The logic programming paradigm is adopted for implementing natural language processing systems because the basic operation for the complex structures is isomorphic with respect to unification. DUALS (Discourse Understanding Aimed at Logic-based System), CIL (Complex Indeterminate Language), and JPSG (Japanese Phrase Structure Grammar) are being developed using the unification-based approach. The large-scale machine readable and understandable dictionaries are also being developed.

1. Introduction

In our daily life, we communicate with one another mainly by means of speech and writing in natural languages. We get a lot of information from books and papers. Communication between human and computer should also be performed in the medium of natural language. The ability of computers to understand natural language will increase their accessibility and flexibility. The Japanese fifth generation computer project aims to develop such intelligent computer systems.

The special characteristic of the Japanese language, i.e., the use of a great many Chinese characters, has made Japanese text input and processing difficult for a long time. Recently, however, Japanese language processing technology has advanced a lot as evidenced by Japanese word processors and commercial machine translation systems. These technologies combined with artificial intelligence are expected to provide new Japanese information processing technology and a new computer culture.

ICOT began research and development of the Fifth Generation Computer Systems (FGCS) in 1982. Natural language processing technology is one of the most important research themes for the FGCS Project because it is a fundamental technology for knowledge information processing and it is used for the research and development of knowledge-base, intelligent-interface and various basic application systems, such as machine translation systems.

The results of research in the initial stage have led us to the conclusion that the logic programming framework is the most suitable for implementing natural language processing systems [6]. The linguistic phenomena in natural language can be formalized in terms of the complex structures of the grammatical features and the constraints on them, as is seen in the feature set of Generalized Phrase Structure Grammar (GPSG) [7], the functional structure of Lexical Functional Grammar (LFG) [11] and the "dags" of PATR-II [20]. The basic mechanism of logic programming is unification in Horn clause logic. Definite Clause Grammar (DCG) [18] is one of the bridges connecting the natural language processing and logic programming. Most of our research activities can be regarded as the improvement and the extension of DCG. GALOP (BUP) [12] is a bottom-up left corner parser which overcomes the drawbacks of top-down parser for DCG. A parallel model of DCG is also being developed [13]. CIL (Complex Indeterminate Language) [15,16] was developed to express and operate the complex features. CIL is an extension of

CH2345-7/86/0000/0987$01.00 © 1986 IEEE

Prolog. The newly introduced "partially specified term" of CIL is suitable for representing the complex features because an extended unification is defined on two partially specified terms. The declarative constraints can be written using the freeze mechanism of CIL. Our approach for semantic analysis, which also deals with pragmatics, is based on situation semantics [1] theory. In this approach, the semantic analysis process corresponds to constructing the relations between situations, and it is implemented as an algebra of events. The merging of events is one of the basic operations and it has an isomorphic structure with respect to unification. Therefore, the basic operations in both syntactic and semantic analyses are isomorphic with respect to those of logic programming, which makes logic programming compatible with natural language processing. DUALS (Discourse Understanding Aimed at Logic-based Systems) is its application system for discourse understanding which reads stories and answers questions on the stories. JPSG (Japanese Phrase Structure Grammar) [9] is a GPSG-based grammar theory for Japanese language whose basic operation is unification. The unification-based parser for JPSG has been developed. We developed some application systems, in order to verify and evaluate this fundamental technology mentioned above. Finally, the processing of large-scale language data is another important aspect of natural language processing. Dictionaries include much information about syntax and semantics which will be utilized for designing the lexicon and the knowledge-base in natural language processing systems. The following three types of machine readable and understandable dictionaries will be developed in the subproject which started in April 1986:

(1) Basic Word Dictionaries:
Four machine readable master dictionaries with 200,000 entries in each dictionary.

(2) Concept Classification Dictionary:
A systematic dictionary for 400,000 concepts including a general thesaurus.

(3) Concept Description Dictionary:
A knowledge database containing semantic descriptions of 400,000 concepts.

These systems mentioned above are implemented on the personal sequential inference machine PSI [17]. The programs are written in its programming language ESP [4]. This paper describes the main research activities and plans for natural language processing of the FGCS Project -- CIL, DUALS, JPSG, and Machine Readable and Understandable Dictionaries.

2. CIL (Complex Indeterminate Language)

2.1 Partially Specified Term

CIL is an extension of Prolog which was designed for the system description language of DUALS. CIL has the freeze predicate, which was originally introduced in Prolog-II [5], as a primitive predicate for realizing various lazy evaluation controls.

CIL introduces a new type of object called "partially specified term" ("partial term" for brief), which is influenced mainly by the notion of assignment developed in the situation theory of [2,3].

CIL = Prolog + Partial Term + Freeze.

We understand partial term as an abstraction from the following data structures, which are widely seen in programming languages, grammar formalisms, etc.:

-- Herbrand term in first order logic.
-- Association list and property list in LISP.
-- Frame and unit in knowledge representation.
-- Record in programming languages.
-- Record in relational data base theory.
-- Assignment in Barwise's situation theory.
-- Category as complex feature in GPSG and functional structure of LFG.

A partial term is written in CIL like this:

{a1/b1, ..., an/bn} n>= 0

where each ai is a ground term and bi is any term, possibly a partial term. The ordinal unification is extended to the partial terms. For example the extended unifier works like this:

```
    unify({a/1, b/2}, {b/X, c/3})
  = {a/1, b/2, c/3},
```

unifying X to be X = 2.

CIL can represent a semantic network even including cycles by using partial terms. For instance, the CIL unifier solves the system of three equations A = B, A = {a/B}, and B = {a/A}, giving A = B = {a/A}, a singleton graph with a self-loop with an edge labelled a. As is easily seen, CIL unification is close to that over infinite trees in Prolog-II. The domain of CIL can be defined formally to be a set of infinite trees.

2.2 Reserved Forms in CIL

The current CIL syntax is an extension of the syntax of DEC-10 Prolog. The following symbols ':', '?', '@', '#', '??', '!' appearing in terms are reserved for the CIL system as follows:

(1) A term of the form X!a is equivalent as a term to the value of the slot of X whose name is a. That is,

```
{al/bl, ..., ai/bi, ..., an/bn}!ai = bi.
```

(2) A term of the form X:C with terms X and C is called a description. C should be an executable form. This term is read "X such that C".

(3) A literal of the form p(...X?...) is equivalent to the literal freeze(X,p(...X...)).

(4) CIL includes convenient forms of term which are defined as follows:

```
@p  <=> X:p(X?), where p is a predicate
        symbol of arity 1 and X is a new
        variable.
V@p <=> V:p(V?), where p is like the
        above.
V#T <=> V:(V=T).
V?? <=> X:(X?=V), where X is a new
        variable.
```

2.3 Situation Semantics in CIL

Although the current CIL is not a full implementation of situation theory yet, it is already useful because of the introduction of partial terms and extended unification over them. Partially specified terms have general and natural descriptive power to represent various data structures of objects necessary for situation theory. The most difficult and basic problem which remains open for CIL, however, is to develop some ideas for designing a control library for constraint description. We think that the problem corresponds directly to the implementation of the constraints of situation theory.

3. DUALS (Discourse Understanding Aimed at Logic-based Systems)

DUALS is an experimental discourse understanding system developed to build a computational model for discourse understanding. The semantic framework is situation semantics in which the sentence meanings are represented as relations between situations. DUALS aims at dealing with the following items within this framework.

1) Primitives for discourse understanding
   (a) Anaphora
   (b) Speech act
   (c) Attitude verb
   (d) Tense
   (e) Quantifier
   (f) Conditions
   (g) Coordination
2) Plan-goal
3) Type description
4) Predicates for manipulating situations

The latest version of DUALS was implemented in CIL. It reads a story written in Japanese language and answers various type of questions about it. The system has the following characteristics:

(1) The semantic structure is constructed with the objects used in situation theory, such as individuals, assignments, relations, locations, conditions, events, parameters, and so on.

(2) Syntax analysis is performed by the parser based on the concurrent process model called SAX (Sequential Analyzer of syntaX and semantics) [13]. There are about 250 grammar rules. Constraints between

situations representing sentence meaning are generated in the form of partially specified terms of CIL by the syntax analysis module.

(3) Anaphora processing algorithm, i.e. the identification of pronouns and zero-pronouns (ellipses) is based on Kameyama's model [10].

(4) Plan-goal-based discourse structures are obtained by the discourse processing module. The rules to construct the discourse structures are described as constraints between events.

(5) The sentence generation module generates the surface sentences from internal meaning structures using grammar rules.

Our technical approach to implementation is to build a package for extended unification in logic programming. An interesting problem, and a more theoretical challenge, is determining what kinds of unification are needed as primitives for implementing situation semantics.

4. JPSG (Japanese Phrase Structure Grammar)

Grammar is an important component of a system for natural language understanding. JPSG is a new Japanese grammar theory for Japanese language based on GPSG. GPSG is suitable for implementation in the logic programming paradigm because it is a natural language syntax theory based on context free grammar (CFG) and its basic computational mechanism is unification. Besides, GPSG has the following features:

(a) Syntactic categories are defined as a complex feature set.

(b) Only phrase structure is used to represent grammatical information.

(c) Metarules for phrase structure rules are introduced.

(d) Constraints on features are described in the syntactic principles, which make phrase structure rules general.

(e) Syntax and semantics are closely related.

Since the Japanese language has a word order variation called "scrambling", GPSG cannot handle it feasibly. In order to handle the "scrambling", the subcategorization feature (SUBCAT) whose value is a set of syntactic categories, is introduced in JPSG. This is an extension of HPSG [19].

Currently, the grammar formalism of JPSG is completed for basic Japanese syntax with the following characteristics:

(1) Syntactic categories are defined as a feature set. Some feature examples are as follows:

PAS -- This indicates the passivizability of a verb. The value is '+' or '-'.

POS -- This indicates a part of speech. The value is one of {V, N, P,...}.

GR -- This indicates a grammatical relation. The value is one of {SBJ, OBJ}.

SUBCAT -- This is the set of syntactic categories which a head category demands as its complements.

SLASH -- This is the set of the missing categories. This feature is used in the same way as in GPSG.

(2) The following phrase structure rule is sufficient for basic Japanese syntax:

(2.1) M --> D H

Rule (2.1) states that mother category (M) dominates one daughter category (D) on the left and one head category (H) on the right. This simplification of the rule is achieved by describing the constraints on the features in syntactic principles.

(3) Since a new SUBCAT feature is introduced, SUBCAT feature principle (SFP) is extended like that of HPSG. SFP describes the inheritance of SUBCAT values as follows:

"The SUBCAT of M is identical to that of H minus D."

For example, if the category of D is N[OBJ] and SUBCAT of H is {N[SBJ], N[OBJ]}, then SUBCAT of M will be {N[SBJ]}. N[OBJ] represents the object noun phrase. The absence of order between the elements of SUBCAT makes it easy to deal with "scrambling" [8].

(4) Most of the syntactic principles used in GPSG are also used in JPSG such as HEAD feature

convention (HFC) and FOOT feature principle (FFP). A set of certain features is called "HEAD features", for example, the POS feature. HFC is a constraint stating that "HEAD features of M are identical to those of H in (2.1)". SUBCAT and SLASH are called FOOT features. FFP is a principle about the inheritance of FOOT features stating that "FOOT feature of M is identical to the union of that of D and H in (2.1)".

(5) A lot of grammatical information is contained in a dictionary.

The basic operation used in JPSG is "unification" because in the syntactic principles mentioned above, the phrase "be identical to" can be replaced by "can be unified to". The parser for JPSG is being developed in CIL. JPSG and CIL are compatible because the syntactic category as feature set corresponds to a partially specified term and syntactic principles correspond to Horn clauses.

5. Machine Readable and Understandable Dictionaries

We presented the unification-based approach for natural language processing and its applications in previous sections. On the other hand, processing of large-scale language data is another important aspect of natural language processing.

This research aims at developing a large-scale database for various natural language processing and speech processing application systems. The language database will be composed primarily of three machine-readable dictionaries: a large-scale basic dictionary as the master dictionary; a concept classification dictionary including a thesaurus; and a concept description dictionary containing descriptions of the meanings of concepts. Application systems utilizing these dictionaries will be developed including machine translation systems and speech recognition systems.

5.1 Basic Word Dictionaries

The term "basic word" means words used in everyday speech, general technical terms, proper nouns, and so on. Machine-readable master dictionaries will be developed containing these basic words. These are the dictionary types:

(1) Japanese
(2) English
(3) Japanese-English
(4) English-Japanese

Each dictionary will include about 200,000 entry words. These dictionaries will be developed in accordance with the specifications already established [14].

5.2 Concept Classification Dictionary

This dictionary will contain specifications of the relations between concepts and indicate exactly how specific concepts are classified in the concept world. Classification bases for the concept world are 'super-sub', 'whole-part', 'composition-element' and other similar relations. The multiple inheritance mechanism will be used as well. The standard thesaurus will form a part of this dictionary. At least 400,000 concepts will be included.

5.3 Concept Description Dictionary

This dictionary will contain the meaning of each individual concept classified in the concept classification dictionary. The combination of the concept classification and the concept description will form the knowledge base for the "general world", and will be utilized in semantic and discourse analysis.

5.4 Application systems

Machine translation systems and speech recognition systems will be developed using these dictionaries.

6. Conclusion

This paper described the research activities on natural language processing within the Japanese Fifth Generation Computer System project. Having finished the initial stage, the project is now at the end of the first year in the four-year intermediate stage. Natural language understanding includes a lot of difficult issues that remain unsolved, especially in discourse understanding. Nevertheless, fruitful results and new ideas have been obtained over the four years of research to date by concentrating on the logic programming framework as discribed in this paper. The last four years has convinced us that the logic programming approach is very promising for implementing natural language processing systems. In the intermediate stage, we will continue this approach to build the subsystems that will be integrated to form the total knowledge information processing system in the final stage.

7. Acknowledgement

This research is being carried out by the Second Research Laboratory of ICOT in very close cooperation with seven manufactures. Many fruitful discussions were held in meetings of the following working groups -- NLS (Natural Language processing System, Head: Prof. Tanaka of Tokyo Institute of Technology), JPS (Japanese Phrase Structure grammar, Head: Prof. Gunji of Osaka Univ.), MRD (Machine Readable Dictionaries, Head: Prof. Ishiwata of Ibaraki Univ.). Finally, the authors would like to thank the director K. Fuchi, ICOT Research Center for providing us with the opportunity for this research.

[References]

[1] Barwise, J. and Perry, J., Situations and Attitudes, MIT Press, 1983.

[2] Barwise, J., The Situation in Logic-III: Situations, Sets and the Axiom of Foundation Center for the Study of Language and Information, CSLI-85-26, 1985.

[3] Barwise, J., Recent Developments in Situation Semantics, Proc. of International Symposium on Language and Artificial Intelligence, 1986.

[4] Chikayama, T., ESP Reference Manual, ICOT TR-044, 1984.

[5] Colmerauer, A., Prolog-II: Reference Manual and Theoretical Model, Internal Report, Groupe Intelligence Artificielle, Universite d'Aix-Marseille II, 1982.

[6] Furukawa, K. and Yokoi, T., Basic Software System, Proc. of FGCS'84, 1984.

[7] Gazdar, G., Klein E., Pullum G., and Sag I., Generalized Phrase Structure Grammar, Oxford, Basil Blackwell, 1985.

[8] Gunji, T., Subcategorization and Word Oder, Proc. of International Symposium on Language and Artificial Intelligence, 1986.

[9] Gunji, T., Japanese Phrase Structure Grammar, D. Reidel Publishing Company, (to appear).

[10] Kameyama, M., Zero Anaphora: The Case of Japanese, Draft of Ph.D Diss., Dept. of Linguistics, Stanford Univ., 1984.

[11] Kaplan, R. and Bresnan, J., Lexical Functional Grammar: A Formal System for Grammatical Representation, in Mental Representation of Grammatical Relations (Bresnan eds.), MIT Press, 1982.

[12] Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H., BUP; A Bottom-Up Parser Embedded in Prolog, New Generation Computing, OHMSHA, LTD. & Springer-Verlag, Vol.1, No.2, 1983.

[13] Matsumoto, Y., A Parallel Parsing System for Natural Language Analysis, Proc. of the 3rd ICLP, 1986.

[14] Miyoshi, H. Tanaka, Y., Yokoi, T., Ishiwata, T., Tanaka, H., Amano, S., Uchida, H. and Ogino, T., Basic Specifications of the Machine-Readable Dictionary, ICOT TR-100, 1985.

[15] Mukai, K., Horn Clause Logic with Parameterized Types for Situation Semantics Programming, ICOT TR-101, 1985.

[16] Mukai, K., Unification over Complex Indeterminates in Prolog, ICOT TR-113, 1985.

[17] Nishikawa, H., Yokota, M., Yamamoto, A., Taki, K., and Uchida, S., The Personal Inference Machine (PSI): Its Design Philosophy and Machine Architecture, ICOT TR-013, 1983.

[18] Pereira, F. and Warren, D.; Definite Clause Grammar for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks, Artificial Intelligence, 13, 231-278, 1980.

[19] Pollard, C., Lecture on HPSG, Unpublished Lecture Notes, Stanford University, 1985.

[20] Shieber, S. M., Using Restriction to Extended Parsing Algorithms for

Complex-Features-Based Formalisms, Proc. of
23rd ACL, 1985.

ARGUS/V : A SYSTEM FOR VERIFICATION OF PROLOG PROGRAMS

Tadashi KANAMORI [†], Hiroshi FUJITA[‡], Hirohisa SEKI[‡], Kenji HORIUCHI[†], Machi MAEJI[†]

[†] Mitsubishi Electric Corporation
Central Research Laboratory
Amagasaki, Hyogo, JAPAN 661

[‡] ICOT Research Center
Institute for New Generation Computer Technology
Mita 1-4-28, Minato-ku, Tokyo, JAPAN 108

Abstract

The verification system Argus/V for proving properties of Prolog programs is outlined by contrasting verification with testing in logic programming. Specifications in Argus/V are given by a class of first order formulas, including goals for normal execution. Contrary to the specifications considered in the usual framework of verification, our specification states a partial property of the program than states what the progrom does as a whole. Though verification in Argus/V does not guarantee the correctness of the program at one stroke, the more properties of the program we prove, the closer the program is to what we intend. Verification in Argus/V is done using inference rules devised for verification, extension of these for usual execution. Execution in Prolog, on which testing is based, is a special case of the inferences in our verification. These two features of Argus/V show that both testing and verification are methods located on a continuous axis to confirm that the program is as we intend it.

Keywords : Verification, Testing, Theorem Proving, Prolog.

Contents

1. Introduction — Testing and Verification —

It is said that Prolog is a higher-level programming language than conventional languages, because operations in Prolog are further distanced from machine structure and closer to purely symbolic manipulation. Though such machine-independent human-oriented characteristics would seem to make programming easier, we still write incorrect Prolog programs quite frequently. For example, we might write the following incorrect program to reverse a list.

```
reverse([ ],[ ]).
reverse([X|L],M) :- reverse(L,N),append(N,X,M).
```

Figure 1. Incorrect Program for Reversing Lists

How can we determine whether the program is what we intend in our mind ? There are two well-known approaches, which give completely different impressions.

One approach is *testing*. Appropriate data is supplied to the program and it is run in order to see whether it shows unexpected responses or behavior. In general,*the more data for the program we test, the better we can determine how close the program is to what we intend it to be.* But testing is considered a rather naive and informal method, and troublesome to the human programmer.

Another approach is *verification*. A specification, that shows *what* the program does as a whole, is given and proved logically with respect to the program, that shows *how* the desired resuls are computed. Verification is considered a rather formal method, to confirm the correctness of the program at one stroke, possibly mechanically.

These approaches seem to be fundamentally separated in conventional programming. How do things stand in logic programming ? In order to answer the question and make the motivation of our verification method understandable, we review what we are doing in testing in Section 2 and the paradigm of logic programming in Section 3. Then an outline of our verification system Argus/V is given in Section 4. In particular, we emphasize the characteristics of specifications and the use of extended execution. In Section 5, we discuss the similarities between testing and our verification with respect to two features, that is, both approaches confirm the correctness of programs gradually and both approaches are based on execution.

2. Testing of Prolog Programs

Let us test the *reverse* program in Figure 1. *Tracer*, one of the testing tools of DEC10 Prolog system [23], responds as follows.

```
| ?- reverse([2,1],M).
   (1) 0 Call : reverse([2,1],_40) ?
   (2) 1 Call : reverse([1],_105) ?
   (3) 2 Call : reverse([ ],_117) ?
   (3) 2 Exit : reverse([ ],[ ])
   (4) 2 Call : append([ ],1,_105) ?
   (4) 2 Call : append([ ],1,1)
   :
```

Figure 2.1. Example of A Testing of Prolog Program

A goal in the trace holds, when the goals (with "Call") below the goal hold. By observing the behavior in the trace and finding the goal contradicting the programmer's intention, the programmer can correct the program in Figure 1 as follows.

```
reverse([ ],[ ]).
reverse([X|L],M) :- reverse(L,N), append(N,[X],M).
```

Figure 2.2. Correct Program for Reversing Lists

After all, *testing is confirming whether there is any difference between the model the programmer has in his/her mind and the actual behavior of the program.*

3. Logic Programming Paradigm Revisited

Now, let us recall the paradigm of logic programming, the central idea of the Japanese Fifth Generation Computer Systems project. *The execution of a Prolog program is the construction of a logical proof* ([1],[9],[10],[15],[20]). For example, the execution of *?-reverse*$([2,1],M)$ is the construction of a proof of $\exists M reverse([2,1],M)$. In general, we have the following inference rule for each definite clause "B :- $B_1, B_2, \ldots, B_m$", where σ is an m.g.u. of A and B. (It is read the formula below the line holds when the formulas above the line hold.)

$$\text{execution} \qquad \frac{\sigma(B_1 \wedge B_2 \wedge \cdots \wedge B_m)}{A}$$

Figure 3.1. Inference Rule for Execution

The proof of a given formula we would like to prove is constructed from the bottom. (See the Figure 3.2 below.) The tracer in Section 2 shows the proof upside-down from left to right.

$$\dfrac{\text{reverse([ ],[ ])}\wedge\text{append([ ],1,1)}}{\qquad\qquad\dfrac{\text{reverse([1],1)}\wedge\text{append(1,2,?)}}{\text{reverse([2,1],?)}}}$$

Figure 3.2. Proof Tree Corresponding to Execution

According to the paradigm, the behavior on which we focus our attention in testing is how the proof of formulas of the form $\exists Y_1, Y_2, \ldots, Y_m\ (A_1 \wedge A_2 \wedge \cdots \wedge A_k)$ is constructed.

4. Verification of Prolog Programs

4.1. Specification of Prolog Programs

The most prominent feature of logic programming languages is, of course, the close relation to logic, the origin of the name. For example, Prolog is very close to, actually a sublogic of, first order logic, long used as one of the most common specification languages. This close relation between programming languages and specification languages prompts reexamination of the nature and the role of specifications and verification.

Specification and verification are still necesary.

Some people think that Prolog programs are formulas of first order logic, rendering independent specification and verification redundant. We agree that in some cases a specification can be a Prolog program as it is, and we cannot write any other simpler specifications of some programs. But, as can be easily seen, Prolog programs are not always specifications. Many computation mechanism have been devised to increase efficiency. Moreover, even if a Prolog program is a description of our intention in logical formulas, it is written from one point of view, which might be erroneous. Confirmation that the program is what we intend must be effected from another point of view by testing or verification [7].

Specifications verified might be partial.

It has often been said that specifications are sometimes as large as programs themselves. The specifications considered so far are usually *total*, that is, they must contain all the information about what the program does as a whole. Such specifications are necessary for program synthesis, because specifications for synthesis usually have to contain all the information for the program to be constructed. The close relation between Prolog and first order logic suggests the possibility of relaxing this restriction on verification. The specifications in our verification system might be *partial*, that is, they are not necessarily the total description of what the program does as a whole, because the program itself may be a part of its own specification. For example, the following property ***reverse-reverse***

$\forall$ X,Y (reverse(X,Y) $\supset$reverse(Y,X))

is also satisfied by the identity relation ***id*** defined by

id(X,X).

Hence, even if we have proved the ***reverse-reverse*** property with respect to the program at hand, we can't conclude that our program is the correct ***reverse***. It is a fortunate situation when we are able to write down the total specification. But we usually content ourselves with a set of partial specifications. *The more properties of the program we verify, the closer the program is to what we intend.* After all, *verification is confirming whether there is any difference between the model the programmer has in his/her mind and the actual properties of the program.* Our verification is much closer to testing in its nature and functioning.

Specification Formulas and Goal Formulas

Now we introduce the class of first order formulas used for specification in Argus/V.

We generalise the distinctions of positive and negative goals. The *positive* and *negative subformulas* of a formula $\mathcal{F}$ are defined as follows (see [24],[21],[22],[25]).

(a) $\mathcal{F}$ is a positive subformula of $\mathcal{F}$.

(b) When $\neg\mathcal{G}$ is a positive (negative) subformula of $\mathcal{F}$, then $\mathcal{G}$ is a negative (positive) subformula of $\mathcal{F}$.

(c) When $\mathcal{G}\wedge\mathcal{H}$ or $\mathcal{G}\vee\mathcal{H}$ is a positive (negative) subformula of $\mathcal{F}$, then $\mathcal{G}$ and $\mathcal{H}$ are positive (negative) subformulas of $\mathcal{F}$.

(d) When $\mathcal{G} \supset \mathcal{H}$ is a positive (negative) subformula of $\mathcal{F}$, then $\mathcal{G}$ is a negative (positive) subformula of $\mathcal{F}$ and $\mathcal{H}$ is a positive (negative) subformula of $\mathcal{F}$.

(e) When $\forall X \mathcal{G}$ or $\exists X \mathcal{G}$ is a positive (negative) subformula of $\mathcal{F}$, then $\mathcal{G}_X(t)$ is a positive (negative) subformula of $\mathcal{F}$.

Example 4.1.1. Let $\mathcal{F}$ be

∀ B,U,A_1,V,A (reverse(B,[U|A_1])∧append(A_1,[V],A) ⊃
∃ A_2 (reverse(B,A_2)∧append(A_2,[V],[U|A])))

Then $\exists A_2(reverse(B, A_2) \wedge append(A_2, [V], [U|A]))$ is a positive subformula of $\mathcal{F}$, while $reverse(B, [U|A_1])$ is a negative subformula of $\mathcal{F}$.

Let $\mathcal{F}$ be a closed first order formula. When $\forall X \mathcal{G}$ is a positive subformula or $\exists X \mathcal{G}$ is a negative subformula of $\mathcal{F}$, X is called a *free variable* of $\mathcal{F}$. When $\forall Y \mathcal{H}$ is a negative subformula or $\exists Y \mathcal{H}$ is a positive subformula of $\mathcal{F}$, Y is called an *undecided variable* of $\mathcal{F}$. In other words, free variables are variables quantified universally, and undecided variables are those quantified existentially when $\mathcal{F}$ is converted to its prenex normal form.

Example 4.1.2. Let $\mathcal{F}$ be

∀ B,U,A_1,V,A (reverse(B,[U|A_1])∧append(A_1,[V],A) ⊃
∃ A_2 (reverse(B,A_2)∧append(A_2,[V],[U|A])))

Then B, U, A_1, V and A are all free variables, while A_2 is an undecided variable.

A closed first order formula S is called a *specification formula* (or *S-formula* for short) when

(a) no free variable in S is quantified in the scope of quantification of an undecided variable in S and

(b) each undecided variable appears only in some positive conjunction of atoms $A_1 \wedge A_2 \wedge \cdots \wedge A_k$ in S.

In other words, S-formulas are formulas convertible to prenex normal form $\forall X_1, X_2, \ldots, X_n \; \exists Y_1, Y_2, \ldots, Y_m \mathcal{F}$ and each Y_i appears only in some positive conjunction of atom in $\mathcal{F}$. Note that S-formulas include both universal formulas $\forall X_1, X_2, \ldots, X_n \mathcal{F}$ and usual execution goals $\exists Y_1, Y_2, \ldots, Y_m \; (A_1 \wedge A_2 \wedge \cdots \wedge A_k)$.

Example 4.1.3. Let S be

∀ B,U,A_1,V,A (reverse(B,[U|A_1])∧append(A_1,[V],A) ⊃
∃ A_2 (reverse(B,A_2)∧append(A_2,[V],[U|A])))

Then S is an S-formula, because free variables B, U, A_1, V and A are quantified outside $\exists A_2$, and A_2 appears only in the positive conjunction $reverse(B, A_2) \wedge append(A_2, [V], [U|A])$. An execution goal

∃ C append([1,2],[3],C)

is also an S-formula.

A formula G obtained from an S-formula S by leaving free variable X as it is, replacing undecided variable Y with $?Y$ and deleting all quantifications is called a *goal formula* of S. Note that S can be uniquely restorable from G. In the following, we use goal formulas instead of original S-formulas. Goal formulas are denoted by F, G, H.

Example 4.1.4. An S-formula

∀ B,U,A_1,V,A (reverse(B,[U|A_1])∧append(A_1,[V],A) ⊃
∃ A_2 (reverse(B,A_2)∧append(A_2,[V],[U|A]))

is represented by a goal formula

reverse(B,[U|A_1])∧append(A_1,[V],A) ⊃
reverse(B,?A_2)∧append(?A_2,[V],[U|A]).

An execution goal

∃ C append([1,2],[3],C)

is represented by a goal formula

append([1,2],[3],?C).

Let S be a specification in an S-formula, M_0 be the minimum Herbrand model [10] of P and P^* be the completion [8] of P. We adopt a formulation as follows : Model-theoretically speaking, verification of S with respect to P is showing $M_0 \models S$. Proof-theoretically speaking, it is proving S from P^* using first order inference and some induction. (Of course, the proof-theoretical formulation is weaker than the model-theoretical formulation. See Section 4.4 for induction.)

4.2. Inference Rules for Verification

Though we have pointed out the similarity between testing and our verification, it is meaningless to just rephrase the definition of verification. The current Prolog interpreter can't execute our S-formulas directly. Now we present the method and the mechanism of our verification (cf. [3],[13],[14],[28]). Our inference rules for verification consist of *extended execution* and *computational induction*. Extended excution is an extension of usual execution and consists of case splittings (∧-deletion,∨-deletion and ⊃-deletion), definite clause inference (DCI), "Negation as Failure" inference (NFI) and simplification. We omit discussion of the case splitting rules in what follows, because they are not used very frequently. See [Kanamori and Seki 1985], [Kanamori 1986] for details.

Using intuitive notations, DCI, NFI and simplification are depicted as inference rules as follows. In the followings, we use $\mathcal{F}_{\mathcal{G}}(\mathcal{H})$ for replacement of *all* occurrences of a formula $\mathcal{G}$ in a formula $\mathcal{F}$ with $\mathcal{H}$ and $\mathcal{F}_{\mathcal{G}}[\mathcal{H}]$ for replacement of an occurrence of a formula $\mathcal{G}$ in a formula $\mathcal{F}$ with $\mathcal{H}$. See the following explanation for the meaning of other notations.

DCI
$$\frac{\sigma(G_A[B_1 \wedge B_2 \wedge \cdots \wedge B_m])}{G_+[A]}$$

NFI
$$\frac{\tau_1(G_A[\wedge_{i=1}^{m_1} B_{1i}]) \quad \cdots \quad \tau_k(G_A[\wedge_{i=1}^{m_k} B_{ki}]) \quad G_A[false]}{G_-[A]}$$

simplification
$$\frac{\sigma(G)_A(true) \quad \sigma(G)_A(false)}{G}$$

Figure 4.2. Main Inference Rules for Verification

4.3. Extended Execution

The execution of positive goals is generalized using polarity.

Definite Clause Inference(DCI)

Let A be a positive atom in a goal formula G and "B :- $B_1, B_2, \ldots, B_m$" be any definite clause in P. When A is unifiable with B by an m.g.u. σ without instantiation of free variables, a new OR-goal $\sigma(G_A[B_1 \wedge B_2 \wedge \cdots \wedge B_m])$ is generated. ($B_1 \wedge B_2 \wedge \cdots \wedge B_m$ is *true* when $m = 0$.) All new variables introduced are treated as fresh undecided variables.

Example 4.3.1. Let S be

∀ A,B,C,U ((reverse(C,B)⊃reverse(B,C))⊃
(reverse(A,[U|B])⊃reverse([U|B],A))).

Then the goal formula of S is

(reverse(C,B) ⊃reverse(B,C)) ⊃
(reverse(A,[U|B]) ⊃reverse([U|B],A))

We can apply DCI to *reverse*$([U|B], A)$ and it is replaced with *reverse*$(A, ?D) \wedge$ *append*$(?D, [U], B)$. Note that the variable in the body is treated as an undecided variable $?D$.

Example 4.3.2. When S is an existential formula of the form $\exists Y_1 Y_2 \cdots Y_m (A_1 \wedge A_2 \wedge \cdots \wedge A_k)$, i.e., of the form of usual execution goals, the goal formula of S is $?\text{-}A_1, A_2, \ldots, A_k$. (The juxtaposition delimited by "," denotes conjunction and $?\text{-}G$ denotes the goal formula obtained by replacing every variable Y in G with $?Y$.) Then usual execution is applied to $?\text{-}A_1, A_2, \ldots, A_k$. The figure below shows an example, where *common* and *reverse* are defined by

common(X,L,M) :- member(X,L),member(X,M).
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).

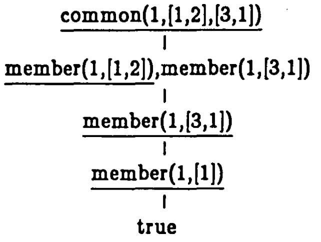


Figure 4.3.1. DCI for Usual Positive Goals

We also generalize the execution of negative goals using polarity.

"Negation as Failure" Inference(NFI)

Let A be a negative atom in a goal formula G. We generate new AND-goals $\tau(G_A[B_1 \wedge B_2 \wedge \cdots \wedge B_m])$ for every definite clause "B :- $B_1, B_2, \ldots, B_m$" in P, whose head B is unifiable with A, and an AND-goal $G_A[\textit{false}]$. ($B_1 \wedge B_2 \wedge \cdots \wedge B_m$ is *true* when $m = 0$.) All new variables introduced are treated as fresh free variables. (Note that A always includes only free variables and τ may be any m.g.u. without restriction.)

Example 4.3.3. Let S be

∀ A,B,C,U ((reverse(A,C)⊃reverse(C,A))⊃
(reverse([U|A],B)⊃reverse(B,[U|A]))).

Then the goal formula of S is

(reverse(A,C) ⊃reverse(C,A)) ⊃
(reverse([U|A],B) ⊃reverse(B,[U|A]))

We can apply NFI to *reverse*$([U|A], B)$. In the first goal,the atom is replaced with *reverse*$(A, D) \wedge$*append*$(D, [U], B)$. Note that the variable in the body is treated as a free variable D. The last goal obtained by replacing the atom with *false* is trivially *true*.

Example 4.3.4. Let S be a specification of the form $\neg A$ where A is a ground atom. Suppose there exist k definite clauses whose heads are unifiable with A by m.g.u.s $\tau_1, \tau_2, \ldots, \tau_k$. When NFI is applied to A, we have $k + 1$ AND-goals

$\neg\tau_1(B_{11} \wedge B_{12} \wedge \cdots \wedge B_{1m_1})$,
$\neg\tau_2(B_{21} \wedge B_{22} \wedge \cdots \wedge B_{2m_2})$,
$\vdots$,
$\neg\tau_k(B_{k1} \wedge B_{k2} \wedge \cdots \wedge B_{km_k})$,
$\neg$*false*.

The last goal formula is trivially *true*. Other goal formulas are of the form $\forall X_1, X_2, \ldots, X_n \neg(A_1 \wedge A_2 \wedge \cdots \wedge A_m)$, because variables introduced from the bodies of the definite clauses are free variables in the generated goal formulas. We can continue applying NFI by selecting atoms in each body of the goal formula. When a selected atom has no unifiable head, the only goal formula generated is the last one, which is always *true*. When all goal formulas are reduced to *true*, $\neg A$ is proved. This is exactly the "Negation as Failure" rule in the usual sense [8],[15]. The figure below shows an example.

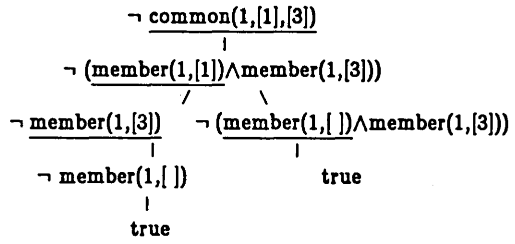

¬ common(1,[1],[3])
¬ (member(1,[1])∧member(1,[3]))
¬ member(1,[3])
¬ (member(1,[ ])∧member(1,[3]))
¬ member(1,[ ])
true
true

Figure 4.3.2. NFI for Usual Negative Goals

We sometimes simplify goal formulas by assuming that some atom is *true* or *false* (cf.[22]).

Simplification

Let G be a goal formula. When $A_1, A_2, \ldots, A_m$ are positive atoms and $A_{m+1}, A_{m+2}, \ldots, A_n$ are negative atoms unifiable to A by an m.g.u. σ without instantiation of free variables ($0 < m < n$), we generate new AND-goals $\sigma(G)_A(\textit{true})$ and $\sigma(G)_A(\textit{false})$.

In the following examples, both σ are $<>$ and undecided variables are not instantiated. For more general simplifications with instantiation of undecided variables, see Figure 4.3.3.

Example 4.3.5. Let G be a goal formula

(add(X,Y,Z) ⊃add(Y,X,Z)) ⊃
(add(X,Y,Z) ⊃add(Y,s(X),s(Z)))

of an S-formula

∀ X,Y,Z ((add(X,Y,Z) ⊃add(Y,X,Z)) ⊃
(add(X,Y,Z) ⊃add(Y,s(X),s(Z)))).

Because $\sigma =<>$ is a substitution without instantiation of free variables and unifies the positive atom *add*(X, Y, Z) and the negative atom *add*(X, Y, Z), we generate new AND-goals

(true ⊃add(Y,X,Z)) ⊃(true ⊃add(Y,s(X),s(Z))) ,
(false ⊃add(Y,X,Z)) ⊃(false ⊃add(Y,s(X),s(Z))) ,

i.e., *add*$(Y, X, Z) \supset$*add*$(Y, s(X), s(Z))$ and *true*. This inference corresponds to generating

$(Y+X)+1=Y+(X+1)$

from

$X+Y=Y+X \supset (X+Y)+1=Y+(X+1)$

in functional programs, i.e., using the equation $X+Y = Y+X$ in the premise and throwing it away. This is called *cross-fertilization* in the Boyer Moore Theorem Prover (BMTP) [5].

Example 4.3.6. Let G be a goal formula

$(\text{reverse}(A,C) \supset \text{reverse}(B,A)) \supset$
$\quad (\text{reverse}(A,C) \wedge \text{append}(C,[U],B) \supset \text{reverse}(B,[U|A]))$

of an S-formula

$\forall\, A,B,C,U\ ((\text{reverse}(A,C) \supset \text{reverse}(C,A)) \supset$
$\quad ((\text{reverse}(A,C) \wedge \text{append}(C,[U],B)) \supset \text{reverse}(B,[U|A])))$.

Because $\sigma = <>$ is a substitution without instantiation of free variables and unifies the positive atom $reverse(A,C)$ and the negative atom $reverse(A,C)$, we generate new AND-goals

$(\text{true} \supset \text{reverse}(C,A)) \supset$
$\quad (\text{true} \wedge \text{append}(C,[U],B) \supset \text{reverse}(B,[U|A]))$,
$(\text{false} \supset \text{reverse}(C,A)) \supset$
$\quad (\text{false} \wedge \text{append}(C,[U],B) \supset \text{reverse}(B,[U|A]))$,

i.e., $reverse(C,A) \supset (append(C,[U],B) \supset reverse(B,[U|A]))$ and *true*. This inference corresponds to generating

$\text{reverse}(C)=A \supset \text{reverse}(\text{append}(C,[U]))=[U|A]$

from

$\text{reverse}(\text{reverse}(A))=A \supset$
$\quad \text{reverse}(\text{append}(\text{reverse}(A),[U]))=[U|A]$

in functional programs, i.e., replacement of the special term $reverse(A)$ with a variable C. This is called *generalization* in BMTP [5].

The Figure 4.3.3. below is one of the sequences of the applications of extended execution. A goal in the sequence holds, when the goals below it hold, like the goals in Figure 2.1.

$\text{reverse}(M,[X|L_1]) \wedge \text{append}(L_1,[Y],L) \supset \underline{\text{reverse}([Y|M],[X|L])}$
$\quad \Downarrow$ DCI with $<>$
$\text{reverse}(M,[X|L_1]) \wedge \text{append}(L_1,[Y],L) \supset$
$\quad \text{reverse}(M,?L_2) \wedge \underline{\text{append}(?L_2,[Y],[X|L])}$
$\quad \Downarrow$ DCI with $<?\overline{L_2 \Leftarrow [X|?L_3]}>$
$\underline{\text{reverse}(M,[X|L_1])} \wedge \text{append}(L_1,[Y],L) \supset$
$\quad \underline{\text{reverse}(M,[X|?L_3])} \wedge \text{append}(?L_3,[Y],L)$
$\quad \Downarrow$ simplification with $<?L_3 \Leftarrow L_1>$
$\underline{\text{append}(L_1,[Y],L)} \supset \underline{\text{append}(L_1,[Y],L)}$
$\quad \Downarrow$ simplification with $<>$
true

Figure 4.3.3. Example of Verification of Prolog Programs

4.4. Computational Induction

Because we have adopted the model-theoretical formulation in 4.1, we need to use some kind of induction in order to make our proof system as strong as possible to approximate the model-theoretic formulation.

The following induction scheme is used for induction on natural numbers [6].

$$\frac{Q(0) \qquad \forall X\ (Q(X) \supset Q(X+1))}{\forall X\ (\text{number}(X) \supset Q(X))}$$

Figure 4.4.1. Induction Scheme for Natural Number

Note that the *number* predicate is defined in Prolog as in Figure 4.4.2 and the induction formulas above the line in Figure 4.4.1 is exactly what are obtained by replacing *number* in Figure 4.4.2 with Q.

```
number(0).
number(X+1) :- number(X).
```

Figure 4.4.2. Prolog Programs Defining Natural Number

Similarly, the induction scheme for *reverse* is obtained by replacing *reverse* in Figure 2.2. with Q. This is de Bakker and Scott's computational induction for Prolog ([2],[9],[11],[12],[29]).

$$\frac{Q([\,],[\,]) \quad \forall L,M,N,X\ (Q(L,N) \wedge \text{append}(N,[X],M) \supset Q([X|L],M))}{\forall L,M\ (\text{reverse}(L,M) \supset Q(L,M))}$$

Figure 4.4.3. Induction Scheme for *reverse*

Example 4.4. When the induction scheme above is applied to *reverse-reverse*, i.e.,

$\text{reverse}(L,M) \supset \text{reverse}(M,L)$

the following two induction formulas are generated.

$\text{reverse}([\,],[\,])$
$\text{reverse}(N,L) \wedge \text{append}(N,[X],M) \supset \text{reverse}(M,[X|L])$.

In general, the goals we are going to prove are not necessarily of the form $p(X_1,X_2,\ldots,X_n) \supset Q(X_1,X_2,\ldots,X_n)$. Moreover, more than two induction schemes might be suggested. In order to manage such situations, we have a device to generate and manipulate induction schemes based on an equivalence-preserving program transformation [27]. See [Kanamori and Fujita 1986] for details.

4.5. Argus/V Verification System

The Argus/V verification system has the module structure depicted below. A given specification is first converted to its goal formula, then passed through the four modules in Figure 4.5.

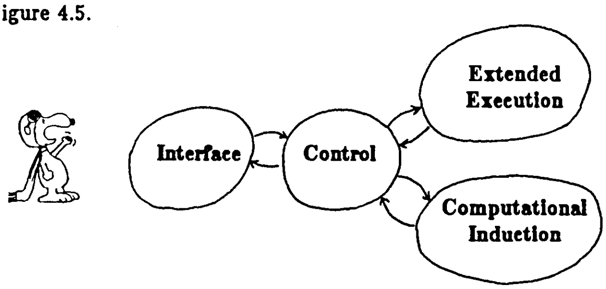


Figure 4.5. Argus/V Module Structure

Because we have additional and generalized inference rules for verification, we need a procedure to select the inference rules to be applied. When and how extended execution and computational induction are applied is controlled by many BMTP-like heuristics [5]. This can be considered a kind of meta-inference [4],[26]. See [Kanamori and Horiuchi 1985] for the use of type inference in Argus/V.

5. Discussion — Testing and Verification —

As shown in Section 4, verification in Argus/V is very close to testing not only in its nature and the role it plays but also in the methodology and its mechanism. In fact, testing is a special case of verification in Argus/V in which the specifications are ground goals or existentially quantified atoms. The difference is that verification in Argus/V is more general than testing. For example, testing can confirm *reverse-reverse* for only lists with specific lengths, while verification in Argus/V proves it for lists of any length. In a sense, testing deals with the superficial observable *directly*, while verification penetrate the interior observable only *indirectly*. In other words, "our *prover* is a *prober*."

6. Conclusions

We have given an outline of the Argus/V verification system for proving properties of Prolog programs by contrasting verification with testing in logic programming. The first version of Argus/V was developed between April 1984 and March 1985. It consists of about 7000 lines in DEC-10 Prolog and takes about 9.5 seconds (CPU time of DEC2060 with 384 kw main memory) to prove *reverse-reverse* automatically. More than 50 theorems have already been proved automatically and the number is increasing.

Acknowledgements

The Argus/V verification system is a subproject of the Fifth Generation Computer System(FGCS) "Intelligent Programming System". The authors would like to thank Dr.K.Fuchi (Director of ICOT) for the opportunity of doing this research and Dr.K.Furukawa (Chief of ICOT 1st Laboratory) and Dr.T.Yokoi (Chief of ICOT 2nd Laboratory) for their advice and encouragement.

References

[1] Apt,K.R. and M.H.van Emden, "Contribution to the Theory of Logic Programming", J.ACM, Vol.29, No.3, pp.841-862, 1982.

[2] de Bakker,J.W. and D.Scott, "A Theory of Programs", Unpublished Notes, IBM Seminar, Vienna, 1969.

[3] Bowen,K.A., "Programming with Full First-Order Logic", Machine Intelligence 10 (J.E.Hayes, D.Michie and Y-H.Pao Eds), pp.421-440, 1982.

[4] Bowen,K.A. and R.A.Kowalski, "Amalgamating Language and Metalanguage in Logic Programming", in Logic Programming (K.L.Clark and S-Å.Tärnlund Eds), Academic Press, 1980.

[5] Boyer,R.S. and J.S.Moore, "Computational Logic", Academic Press, 1979.

[6] Burstall,R., "Proving Properties of Programs by Structural Induction", Comput.J., Vol.12, No.1., pp.41-48, 1969.

[7] Clark,K.L. and S-Å.Tärnlund, "A First Order Theory of Data and Programs", in Information Processing 77 (B.Gilchrist Ed), pp.939-944, 1977.

[8] Clark,K.L., "Negation as Failure", in Logic and Database (H.Gallaire and J.Minker Eds),pp.293-302,1978.

[9] Clark,K.L., "Predicate Logic as a Computational Formalism", Chap.4, Research Monograph : 79/59, TOC, Imperial College, 1979.

[10] van Emden,M.H. and R.A.Kowalski, "The Semantics of Predicate Logic as a Programing Language", J.ACM, Vol.23, No.4, pp.733-742, 1976.

[11] Gordon,M.J.,A.J.Milner and C.P.Wadsworth, "Edinburgh LCF — A Mechanized Logic of Computation", Lecture Notes in Computer Science 78, Springer, 1979.

[12] Hagiya,M. and T.Sakurai, "Foundation of Logic Programming Based on Inductive Definition", New Generation Computing, Vol.2, pp.59-77, 1984.

[13] Hansson,A. and S-Å.Tärnlund, "A Natural Programming Calculus", Proc.of 6th International Joint Conference on Artificial Intelligence, pp.348-355,1979.

[14] Haridi,S. and D.Sahlin, "Evaluation of Logic Programs Based on Natural Deduction", Proc.of 2nd Workshop on Logic Programming, 1983.

[15] Jaffar,J.,J-L.Lassez and J.Lloyd, "Completeness of the Negation as Failure Rule", Proc.of 8th International Joint Conference on Artificial Intelligence, Vol.1, pp.500-506, 1983.

[16] Kanamori,T.and H.Seki, "Verification of Prolog Programs Using An Extension of Execution", ICOT Technical Report, TR-096, 1984. Also Proc.of of 3rd International Conference on Logic Programming, 1986.

[17] Kanamori,T.and H.Fujita, "Formulation of Induction Formulas in Verification of Prolog Programs", ICOT Technical Report, TR-094, 1984. Also Proc.of Conference on Automated Deduction, 1986.

[18] Kanamori,T.and K.Horiuchi, "Type Inference in Prolog and Its Applications", ICOT Technical Report, TR-095, 1984. Also Proc.of 9th International Joint Conference on Artificial Intelligence, pp.704-707, Los Angeles, 1985.

[19] Kanamori,T., "Soundness and Completeness of Extended Execution for Proving Properties of Prolog Programs", ICOT Technical Report, to appear, 1986.

[20] Kowalski,R.A., "Logic for Problem Solving", Chap.10-12, North Holland, 1980.

[21] Manna,Z.and R.Waldinger, "A Deductive Approach to Program Synthesis", ACM Trans. on Programming Languages and Systems, Vol.2, No.1, pp.90-121, 1980.

[22] Murray,N.V., "Completely Non-Clausal Theorem Proving" Artificial Intelligence, Vol.18, pp.67-85, 1982.

[23] Pereira,L.M.,F.C.N.Pereira and D.H.D.Warren, "User's Guide to DECsystem-10 Prolog", Occasional Paper 15, Dept.of Artificial Intelligence, Edinburgh,1979.

[24] Prawitz,D.,"Natural Deduction,A Proof Theoretical Study" Almqvist & Wiksell, Stockholm, 1965.

[25] Schütte,K., "Proof Theory", (translated by J.N.Crossley), Springer Verlag, 1977.

[26] Stering,L. and A.Bundy,"Meta-Level Inference and Program Verification", in 6th Automated Deduction (W.Bibel Ed), Lecture Notes in Computer Science 138, pp.144-150, 1982.

[27] Tamaki,H. and T.Sato, "Unfold/Fold Transformation of Logic Programs", Proc.of 2nd International Logic Programming Conference, pp.127-138, 1984.

[28] Tärnlund,S-Å., "Logic Programming Language Based on A Natural Deduction System", UPMAIL Technical Report, No.6, 1981.

[29] Weyrauch,R.W. and R.Milner, "Program Correctness in A Mechanized Logic", Proc.of 1st USA-Japan Computer Conference, 1972.

A 32-BIT CMOS MICROPROCESSOR WITH SIX-STAGE PIPELINE STRUCTURE

H.Kaneko, Y.Miki, S.Nohara, K.Koya, and M.Araki

Microcomputer Product Div. NEC corp.
1753 Shimo-numabe, Nakahara, Kawasaki City, 211 JAPAN

Abstract

32-bit microprocessors are the key devices which carry high data processing capability, that was obtained by earlier general purpose computer systems and mini-computer systems, in much lower cost. Earlier 32-bit microprocessors were limited to adopt excellent architecture and design using appropriate hardware by number of devices could be fabricated on a chip. Complex functions such as Virtual Memory management and Floating-Point operations had to be obtained by multi-chip configuration in an earlier microcomputer system.

V60 is a 32-bit CMOS microprocessor with the Virtual Memory management facility and Floating-Point arithmetic operations based on IEEE-754 Floating-Point standard on a single chip, and adopts six-stage pipeline structure constructed with six independent pipeline stages.

We obtained a peak performance of 3.5 Million Instructions Per Second (MIPS) execution speed at 16MHz clock rate, and integrated 375,000 transistors on a single chip using a state of the art double-metal layer CMOS process technology based on 1.5um design rule.

1.Introduction

Advantages in VLSI fabrication technology have made it possible to design highly integrated VLSI microprocessors. We are now in a technology region that some hundreds of thousands of circuit devices can be integrated on a tiny chip with reasonable cost[1,2]. The progress, in general, brings three benefits to general-purpose microprocessor designers; multiple 32-bit internal data paths; functionality integration such as pipelined structure; higher performance by short clock cycle.

1.1 32-bit Microprocessors

Microprocessors have been used in a variety of application area, from small control systems to large computing systems, since they were introduced to marketplace in 1971.

The number of devices on a microprocessors have increased about one hundred-times in this 15 years. In addition, the speed of expansion in application range does not seem to be decreased. Also the machine cycle of them have speeded up about 30 times.

The internal data-processing ability has been expanded from 4, 8, and 16 to 32-bit, because of the requests to provide the strong data-processing power and large address space.

For 32-bit microprocessors, the main application area seems to be in data-processing because of their excellent ability that obtained by general-purpose computer and mini-computer systems for information data-processing. Fundamental requests from such application area will include high-level languages support, Operating-Systems support, and application programs support.

To provide high-performance of the 32-bit microprocessors, they operate at shorter operation clock rate (more than 12MHz) than earlier microprocessors.

On the other hand, a large amount of the address space increases the main memory devices and goes to the expense the high-speed main memory systems (more than 1MB) fitted with the machine cycle of the 32-bit microprocessors.

But, the real mission of the 32-bit microprocessors to supply the low-cost information processing systems but the high-ability of their data-processing.

1.2 Design Limitations for Microprocessors

There are two physical limitations to design the microprocessors.

One of those is cost-performance trade-off problem. It decreases the yield of chips on a silicon wafer by throwing much number of the transistors into a chip. In the worst case, the chip designers can not get a properly working chip from a silicon wafer. It is given up to employ a large amount of hardware circuits or high-level hardware techniques for the reasonable chip cost. Consequently this problem limits an architecture designer for the microprocessors to implement his ideal architecture on a chip.

In other words, the functions of the microprocessors are pre-determined and limited in the old-fashioned microprocessor design by number of transistors for employing register set, instruction set, and something related his architecture on a chip.

The other limitation is regarding the operation speed of a fundamental logic element on a silicon wafer. The machine cycle and the timing design are expected by the operation speed of a transistor of silicon chip. The limitations holds down the performance of microprocessors.

1.3 The V60 Microprocessor

A 32-bit microprocessor V60 (uPD70616) is the newest member of NEC's V-series original microprocessors, and is the first implementation of

CH2345-7/86/0000/1000$01.00 © 1986 IEEE

the 32-bit microprocessor family[6].

Philosophy that the processor functions must be included in a single chip as much as possible reflects the V60 architecture. The bottle-neck problems regarding the data transfers between the CPU and external systems can be solved by reducing the number of times to access slow-speed external resources.

General Purpose Registers	32 (32-bit)
Instructions	273 instructions in 119 types
Virtual Address Space	4GB per a space
Real Address Space	16MB
Virtual Memory Management	Demand Paging (4KB/page)
MMU	On-Chip
Translation Look-Aside Buffer	16-entry Full Associative
TLB Hit-Ratio	over 98%
Floating-Point Operations	On-Chip (based on IEEE-754)
Pipeline Stages	6
Simultaneous Execution	4 instructions
Process	Double-metal layer CMOS
Design Rule	1.5uM
Transistors	375,000
Chip Size	13.92mm X 13.80mm
Clock Rate	16MHz (62.5nS)
Power Supply	+5V
Power Dessipation	1.5W
Package	68-pin Pin Grid Array
External Address Bus	24-bit
External Data bus	16-bit
Performance	3.5MIPS

Table-1 Features of V60

V60 has following 4 design goals and features to realize using the advantages of highly integrated CMOS devices[4].

1).Generality

V60 has instruction and register architecture to support the high-level languages. V60 has 273 instructions in 119 types, 21 addressing modes, and thirty-two 32-bit general purpose registers.

Thirty-two 32-bit general purpose registers decrease data transfers of the operand data from/to the main memory system, and reduce the bus-traffic.

2).Application Supports

V60 employs high-level instructions, and various data types such as Character-String, variable length Bit-Field, and Floating-Point data type based on IEEE-754 Floating-Point standard[3].

Several high-level instructions shorten the total instruction code size and reduce the accesses for the instruction codes.

To decrease the loading of the Operating-System for explodingly increased amount of data and memory protection problems, V60 has on-chip Memory Management Unit (MMU) to supply 4GB virtual addressing space with the demand paging method.

On-chip MMU and Floating-Point operations have the advantage to reduce the communications between the CPU and the other external functional chips.

3).Expandability

It is considered to obtain expandability from the past that V60 has Emulation Mode in which it can directly execute instruction set of NEC's 16-bit CMOS microprocessors at object code level. On the other side, V60 has much transparency to expand with the external functional devices such as Floating-Point Co-Processor for the future.

4).High-Performance

V60 has a peak performance of 3.5MIPS by employing six-stage pipeline structure, operating at 16MHz clock rate, and adopting the architecture such as employing high-level instructions that close almost data-processing in a chip.

1.4 Pipeline Structure

One of V60's design goal is to supply high-performance to the system designers. It can be obtained by high-speed execution using high-speed machine cycle. But there are two general reasons to disturb high-speed machine cycle.

One is the limitation in the number of logic gates that are connected in series and operate in the same clock cycle. It is regarding to the propagation delay time of a transistor. It means that operation clock cycle is determined by operation speed of complex or high-level functional hardware circuits like Arithmetic Logic Unit (ALU).

Second is the limitation in the speed of main memory system connected to the CPU. The machine cycle has been accelerated about 30 times, but the access time of main memory devices (Random Access Memory) has speeded up only a couple of times in this 15 years. It takes more expensive cost to employ the main memory devices matched to the microprocessor operating at high-speed machine cycle.

To solve those problems, it is the best solution to employ pipeline structure to the microprocessor using in the sophisticated general-purpose computer systems[5].

There are some problems to employ a pipeline structure; Increasing the registers to arbitrate the operation of each pipeline stage, hardware to solve the Pipeline Hazard problems related to the preceding execution of various instructions, independent hardware such as ALU in each pipeline stage, and complex control, etc. So much and complex hardware is needed to employ pipeline structure, hardware designers abandoned to a real pipeline structure into the earlier microprocessors.

V60 is freed from the limitations concerned to a great number of hardware by fine CMOS fabrication, and can employ a real pipeline structure and some high-level techniques to enhance the performance.

2.Pipeline Structure of V60

We describe implementation and features of the V60's pipeline structure in this section.

V60 obtains well-balanced high-performance for both the simple instructions but frequently used such as the integer operations, and the complex instructions but infrequently used such as the Floating-Point arithmetic operations. V60 adopts six-stage pipeline structure to obtain high-performance for the simple instructions.

2.1 Pipeline Stages

The pipeline structure of V60 is constructed with six independent pipeline stages; Prefetch Unit (PFU), Instruction Decode Unit (IDU), Effective Address Generator (EAG), Memory Management Unit (MMU), Bus Control Unit (BCU), and Execution Unit

(EXU). An Unit is the hardware that corresponds to each pipeline stage. Each pipeline stage operates asynchronously and up to four instructions are processed in various pipeline stages. In following paragraphs, we describe constructions and functions for each Unit.

Fig.1 shows the configuration of functional Units in V60.

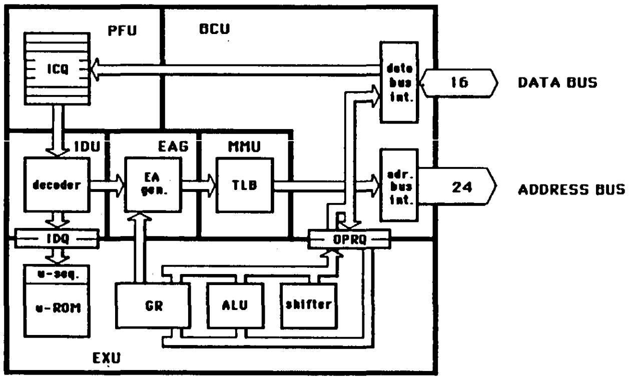


Fig.1 V60 Functional Unit Diagram

2.1.1 PFU (Prefetch Unit)

PFU is the Unit takes charge of instruction code pre-fetch stage preceding to instruction decode and execution stages. PFU has 16B Instruction Code Queue registers (ICQ) and Instruction Code Aligner (ICA). ICQ stores instruction codes transferred from BCU through 16-bit data bus (DBUS).

PFU requires a data transfer request to BCU when PFU detects that more than 4B free area exist in ICQ.

ICA separates instruction code in ICQ to the operation code field, the addressing field, and the displacement field, which are packed into an instruction code.

Fig.2 shows the functional block diagram of PFU.

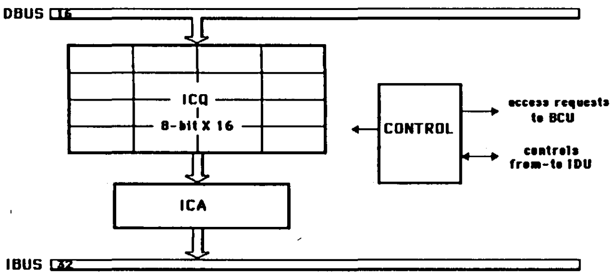


Fig.2 PFU Functional Block Diagram

2.1.2 IDU (Instruction Decode Unit)

IDU is the Unit takes charge of the instruction decode stage. IDU decods the instruction code stored in PFU, and transfers results of the decoding to EAG and EXU. IDU also controls the basic sequence for the pipeline operation.

The instruction decode operations are simultaneously performed for both operation code field and addressing field by the independent decoders.

The results of the decoding transferred to EAG as about fifty output signals that include register, scaling and operand specifying information used for operand Effective Address (EA) calculation. When IDU detects that an instruction needs the EA(s), IDU requests the operand EA calculation to EAG.

In other side, the results of the decoding transferred through Instruction Decode Queue registers (IDQ) to EXU as about fifty output signals that include operand information, arithmetic operation information, and information specifying a microprogram executed for an instruction. When IDU detects that EXU terminates the actual instruction execution, it requests to start of executing of a next microprogram.

Fig.3 shows the functional block diagram of IDU.

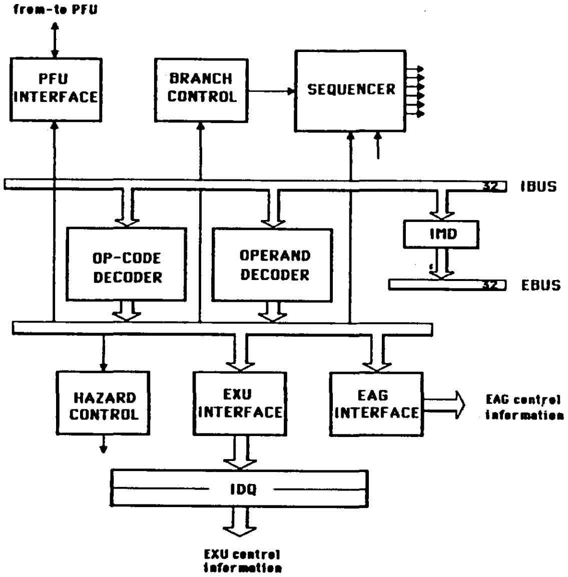


Fig.3 IDU Functional Block Diagram

2.1.3 EAG (Effective Address Generator)

EAG is the Unit takes charge of operand Effective Address (EA) calculation stage. To obtain high-speed EA calculation of 21 addressing modes, EAG adds 4 data stored in EA Temporary registers (EATR0-3) at the same time; base value, index value, displacement, and up-to-date data for pre-decrement or post-increment addressing mode. An adding operation uses high-speed four-inputs 32-bit adder combined with Carry Save Adder (CSA) and Carry Propagation Adder (CPA).

The base and index value of EA stored into the General Purpose registers in EXU are asynchronously transferred through 32-bit data bus (EBUS) for the operation of EXU.

EAG also has Program Counters (PC) to hold the leading address of the instructions corresponding to each pipeline stage. These PCs are up-to-dated at every termination of operation in IDU and EXU, and the contents of PCs are used for EA calculation at PC relative addressing mode.

If a request to calculate EA is transferred by IDU with issuing an external bus cycle, EAG requires the virtual-to-real address translation request to MMU and the access request to BCU. EAG has facility to detect that an access request extends over 2 memory pages (a page is 4KB region), and to calculate new page address and inform it to BCU automatically (Page Boundary processing).

Fig.4 shows the functional block diagram of EAG.

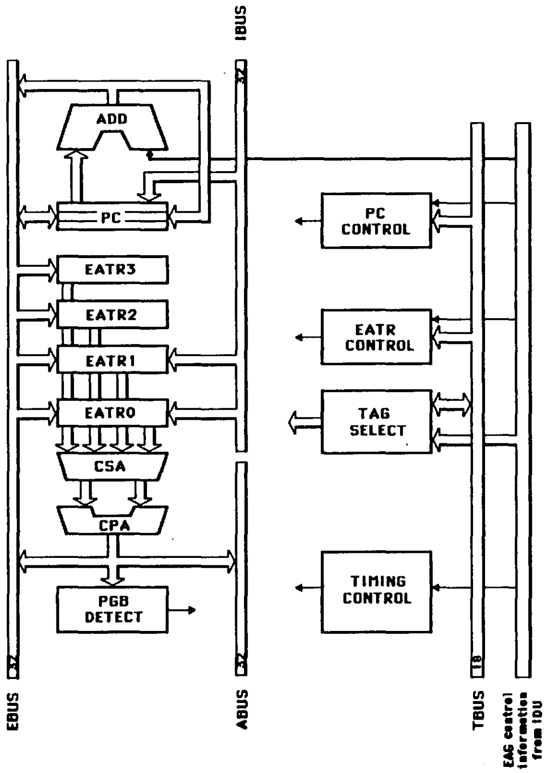


Fig.4 EAG Functional Block Diagram

2.1.4 MMU (Memory Management Unit)

MMU is the Unit takes charge of address translation stage from Virtual Address (VA) to Real Address (RA).

A 4GB VA space is constructed with 3 level hierarchies; Section, Area, and Page. Section is a hierarchy to define basic structure of VA space at construction of multiple-VA spaces, and a VA space has four 1GB-Sections. Area is hierarchy to have common address region privately between VA spaces, and a Section has 1,024 1MB-Areas. Page is hierarchy to realize the Virtual Memory mechanism with Demand Paging method, and an Area has 256 4KB-Pages.

Upper 20-bit of VA (it is called Virtual Page Number) transferred from EAG is translated to 12-bit of upper RA (it is called Real Page number).

This address translation is provided by the Full Associative type Translation Look-aside Buffer (TLB) with 16-entries for high-speed translation. Each TLB entry has 21-bit associative Memory part and 24-bit Data Memory part. 9-bit of the Data Memory part are used as memory protection information (3-bit are used for Page level protection and 6-bit are used for Area level protection), and access rights are checked at the translation.

If no matching entry exists for the translation in TLB, MMU requests to start of a microprogram to replace one of the TLB entries. Pseudo Least Recently Used (LRU) algorithm is used for TLB entry replacement. The TLB achieves about 10^{-2} miss hit ratio (or equivalently 99% hit ratio) in tracing data by computer simulations.

Fig.5 shows the functional block diagram of MMU.

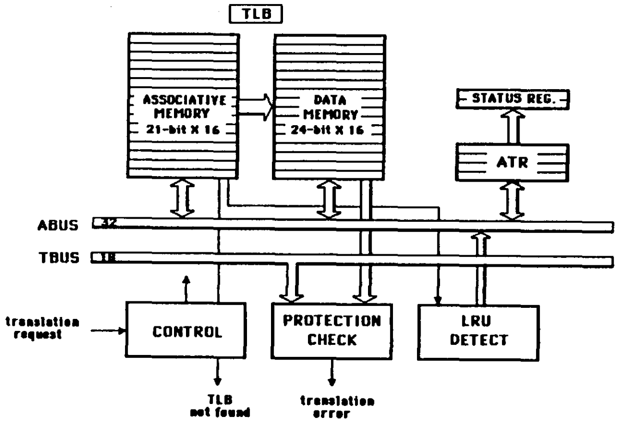


Fig.5 MMU Functional Block Diagram

2.1.5 BCU (Bus Control Unit)

BCU is the Unit takes charge of the external bus access stage for operand read/write and instruction code pre-fetch. BCU takes the interface between V60 and external memory systems or I/O devices regarding the data transfers and pin functions. The requests to issue an external bus cycle from PFU, EAG and EXU are arbitrated by BCU according to specifying of the bus cycle. A request for branch operation has the highest priority, a request for operand read or write operation has next priority, and the lowest priority gives to a request for instruction code pre-fetch operation.

For one operand read/write operation, BCU automatically issue one-to-three external bus cycles according to specified data type and Real Address; 2 bus cycles are issued at a half-word data (16-bit) access for odd address and a word data (32-bit) access for even address, and 3 bus cycles are issued at a word data access for odd address.

Operand Aligners (OWA and ORA) fitly arrange operand data from-and-to EAU. So, EAU can transfer an operand data without any consideration about data type and Real Address.

Two Operand Read registers (OPR) hold data transferred from external data bus until EXU reads out them to reduce idle time in BCU.

EXU transfers the data from OPRs through one of two 32-bit data bus (MBUS and SBUS), operand write to Operand Write register (OPW) through MBUS.

BCU also has the Address Trap facility which is one of the strong software debugging facilities. The Address Trap facility is the mechanism that generates a trap and transfers the control from user's program to a debugging routine when BCU detects that the Virtual Address of an external bus cycle matches pre-determined values. It can select to specify operand read/write and execution (pre-fetch) bus cycle, and two-sets of target address for the Address Trap facility.

Fig.6 shows the functional block diagram of BCU.

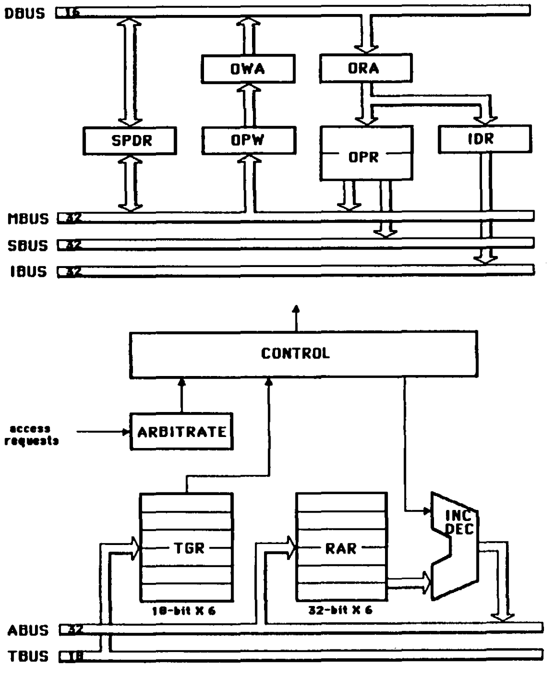


Fig.6 BCU Functional Block Diagram,

2.1.6 EXU (Execution Unit)

EXU is the Unit takes charge of actual execution stage. EXU has thirty-two 32-bit General Purpose registers (GR), sixteen 32-bit Scratch-Pad registers (SPR; parts of them are used for some privilege registers), 32-bit full functional Arithmetic Logic Unit (ALU) with 4 Temporary registers (TEMP), 64-bit Barrel shifter, and some registers for the data processing including Processor Status Word (PSW).

GRs are connected to three sets of 32-bit internal data bus (MBUS, SBUS, and EBUS), and those data buses can be read out the contents of various GRs simultaneously. MBUS and SBUS are used for data paths to transfer two data for ALU operation at the same time. This structure is well known as Dual Bus structure. EBUS is used for Effective Address calculation by EAG. The strong Carry Look Ahead (CLA) circuits are added to ALU to guarantee high-speed operation.

EXU itself is a microprogram controlled machine. A vertical type microinstruction has 32-bit width, and can control 4 control points simultaneously. Microcode is stored in Microcode ROM (MROM) which has 5,184 steps capacity (it equals to 198,108-bit). MROM is addressed by one of Micro-Address registers (MAR). Four 16-bit MARs provide 2^{16} words of addressing space and 4 level nesting structure of Micro-Subroutines.

EXU starts to execute of the microprogram specified by IDU, when IDU terminates the decoding of next instruction. Also a microprogram is activated when EXU detects that an Exception or an Interrupt is requested.

When the execution of a microprogram reaches to the microinstruction for terminating a microprogram, EXU waits its operations until IDU terminates the instruction decoding of next instruction.

Fig.7 shows the functional block diagram of EXU.

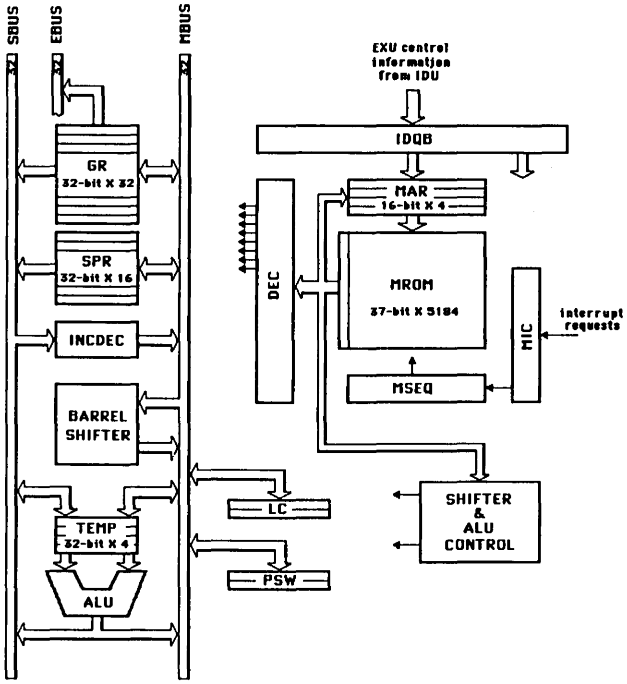


Fig.7 EXU Functional Block Diagram

2.2 Parallel Operation

The basic execution of an instruction in V60 is divided to 7-steps as shown in Fig.8. Fig.8 also shows the Unit corresponding to each step.

In those steps, step-1,5, and 7 are performed by BCU, and each step is sequentially operated. Because V60 reduces the access to slow-speed external resources by employing the General Purpose register architecture and several high-level instructions, the loading of BCU is not so heavy. So, it seldom occurs that BCU is requested to operate for step-1,5, and 7 at the same time.

Also the address translation by MMU in step-4 is always performed following the operand EA calculation by EAG in step-3. In V60 it can assume that the operation time of MMU is not concerned to

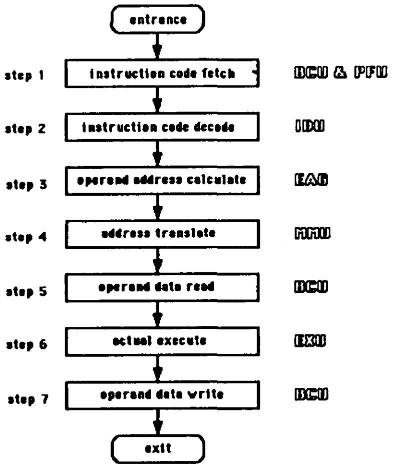


Fig.8 Instruction Execution Flow in Pipeline

the total execution time by employing the high-speed translation mechanism by TLB.

At the some point of the time, step-2,3,6, and one of step-1,5, or 7 are simultaneously performed. So V60 can execute 4 instructions at the same time as shown in Fig.9. Each instruction is executed at every 2 operation clock cycles in the best case. Actually operation time in each pipeline stage changes depend on the instruction to be executed and its addressing mode.

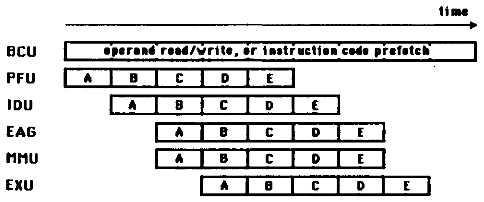


Fig.9 Basic Pipeline Operation

2.3 Queueing Operation

V60 employs the asynchronous pipeline structure considering confliction for the operation time of each pipeline stage. There are some instructions such as the Character-String manipulate instructions which access many operand data in the external memory. In other side, there are some instructions such as multiplication and division instructions which spend much time for their actual execution. In both cases, each pipeline stage might not operate equally; BCU has the heavy loading in the former case, and EXU has the heavy loading in the latter case. V60 avoids the problem that a pipeline stage can not operate depend on the state of other pipeline stages.

To solve above problem, each pipeline stage has the Queueing facility for the next pipeline stage. Under circumstances that each pipeline stage needs different time to operate, V60 avoids the disturbances regarding to other pipeline stages by employing such Queueing facilities.

We describe the Queueing facilities in this paragraph.

1).Instruction Code Queue

PFU has 16B instruction Code Queue registers (ICQ). ICQ stores the instruction code transferred from BCU, when IDU is busy for the instruction decoding. So instruction code pre-fetch operations perform 8 times independent on the state of IDU.

2).Instruction Decode Queue

IDU has the Queueing facility for EXU. Instruction Decode Queue registers (IDQ) stores the results of the instruction decode operation to be transferred for EXU. IDQ is constructed with two sets of the queueing register with about fifty-bit width. So IDU can decode 3 instructions even though EXU is in busy state for actual execution of an instruction.

3).Operand Read Queue

BCU also has the Queueing facility for EXU. Operand Read registers (OPR) are two sets of 32-bit register that store the operand data transferred from the external data bus at an external operand read bus cycle. During actual execution in EXU without an operand read data, BCU can store two 32-bit operand data to OPRs.

2.4 Hazard Detection and Avoidance

The Pipeline Hazard is the obstacle that processing for an instruction depends on the result of execution for preceding instructions in the computer systems with pipeline structure. Detection and avoidance for some types of the Pipeline Hazard regarding that plural instructions are simultaneously executed in various pipeline stages, are general and important problems in the pipeline structure.

The Pipeline Hazard problems that may occur in V60 and the solutions for them are described as follows.

1).Register Hazard

The Register Hazard occurs when preceding instruction uses the contents of General Purpose registers (GR), which are changed by the actual execution of an instruction in EXU, for operand Effective Address (EA) calculations in EAG as base or index value.

Register Hazard detector (RHD) in IDU detects Register Hazard and controls each pipeline stage to avoid it.

2).Flag Hazard

The Flag Hazard occurs when preceding Conditional Branch instruction refers the Conditional flag(s), which is changed by actual execution in EXU, of PSW.

Flag Hazard detector (FHD) in IDU detects Flag Hazard and controls each pipeline stage to avoid it.

3).Memory Hazard

The Memory Hazard occurs when an operand read data or part of it is referred by preceding instruction is changed as the operand write data by actual execution in EXU. Memory Hazard detector (MHD) in BCU checks that the EA for the requested operand read operation has overlapped area with the EA of the expected operand write operation. When MHD detects existing of the overlapped area, BCU cancels the request for the external operand read bus cycle.

Also BCU directly transfer the operand write data in Operand Write register (OPW) to Operand Read register (OPR) as the operand read data using Short-Path, after EXU performs the actual operand write operation. Then an extra operand read bus cycles are not issued.

4).I/O Hazard

The I/O devices are usually connected to V60. They change their internal status through the reading operation by the CPU. In these case, read operations can not retry. If preceding operand read operation (I/O read bus cycle) occurs before the actual execution of an Input instruction (including the instruction which accesses the memory-mapped I/O) starts, an incovenience that it can not perform to retry for the I/O devices may happen. It is that even through an Input instruction is cancelled by requests of some Interrupts or occurring of some Exceptions, the internal status of an I/O device is already changed by the preceding operand read operations.

I/O Hazard detector (IOHD) in EAG always check that the operand read operation for the I/O device is requested. If IOHD detects the request for the I/O devices, EAG instructs to hold the following requests for the operand read bus cycle until EXU starts to execute an Input instruction.

2.5 Exception and Interrupt Handling

The Exception and Interrupt handling are another general problems in the computer systems with pipeline structure and Virtual Memory mechanism.

In the view point of pipeline structure, these are the problems what Unit detect them and when accept them. Basicly an Exception/Interrupt that is detected in earlier pipeline stage than actual execution stage is hold to accept until the actual execution begins. An Exception/Interrupt in the middle of actual execution is accepted only when the internal state of each pipeline stage are guaranteed to hold during execution of Exception/Interrupt processing.

In the view point of Virtual Memory mechanism, these are the problems also what Unit detect them, when accept them, and how restart the instruction. An Exception regarding the Virtual Memory mechanism like Page-Fault is detected in MMU preceding actual execution. Actual execution for the instruction with the Exception is not performed. So, V60 can restart this instruction from top of it after execution of Exception processing.

1).Abort for Pipeline Processing

Basicly an Interrupt request is accepted at the end of actual execution in EXU for an instruction. The Exceptions regarding an instruction, which are detected preceding the actual execution, are accepted just before the actual execution begins. In both cases, an instruction is not aborted in the middle of actual execution except the instructions with continual operand accesses such as String instructions.

EXU checks Exception/Interrupt requests at the end of actual execution for an instruction, and activates a microprogram for them independ on the status of other Units.

2).Suspend for Exceptions

The Exceptions regarding an instruction, which are detected in the preceding pipeline stage, are suspended to accept until the instruction is guaranteed for the actual execution. Page-Fault at the pre-fetch operation and Memory Management Exceptions at operand read operation are good examples for them.

Instruction Decode Queue (IDQ) has some information regarding the Exceptions detected in the preceding pipeline stages. EXU checks the exist of pre-detected Exceptions before the start of actual execution for an instruction.

3).Abort and Restart for TLB not-found

If no match entry exists for the address translation of a memory operand (read/write) in TLB, actual execution sequence is basicly aborted by same manner as Exception/Interrupt. For some instructions such as String instructions, which follow the address translations for memory operands during the actual execution, the execution is aborted in the middle of actual execution.

For both cases, the operations in each pipeline stage are frozen and can be restarted after the processing of the TLB entry replacement from original status.

4).Abort for Executing Instruction

There are two cases that an instruction is aborted in the middle of actual execution. First case is serious Exception like Reset. In this case, the state of each pipeline stage is not kept.

The other case is an Interrupt during the execution for a complex instruction such as String instruction with longer actual execution time than that of basic instructions. In this case the aborted instruction can restart after the interrupt processing. EXU checks an interrupt request at some points of actual execution.

2.6 Other Features

V60 has some other features regarding the pipeline structure to obtain the high-performance.

1).Branch Instruction Processing

The execution speed of a branch instruction is the most effectively for the computer with the pipeline structure, because such instruction avoids the advantages of preceding executions. Because a pipelined machine that executes a branch instruction seems as non-pipelined machine. In general, branch instruction must be detected and executed in earlier pipeline stage than the actual execution stage.

V60 executes branch instructions in IDU. IDU instructs PFU to cancel the validity of pre-fetched instruction codes in ICQ, EAG to calculate the target branch address and transfer the value to a PC in EAG, and BCU to restart the instruction code fetch operation, when IDU detects a branch instruction; EXU operates for the termination of the actual execution of a branch instruction.

2).Burst String Read Operation

To increase the external bus traffic more effectively, BCU has a facility for the preceded operand reading. For such data structures located in memory space continuously and linearly as the Character-String data type, the next addressed data has high probability to be accessed as a data is accessed.

BCU automatically issue the operand read bus cycle in the Burst String mode, as no data is in OPRs independently on the request from EXU. BCU also up-to-dates the address in a RAR for the next operand.

EXU can get the next operand read data during the data processing for the current operand read data.

3).Floating-Point Operation Support

To obtain the excellent performance for Floating-Point arithmetic operations, several hardware supports are provided in EXU.

The 64-bit Barrel shifter does not only shift and rotate a 32-bit data in one operation clock cycle (including logical-shift, arithmetic-shift, and extended rotate operation), but extracts a 32-bit data started from any bit-position in a 64-bit data. The Barrel shifter is used for the scaling operation of a Floating-Point data processing.

The Leading-One detector finds the bit-position, which is the first bit has logical-one value leading from LSB or MSB in a 32-bit data, with high-speed. This Leading-One detector is used for the normalizing operation of a Floating-Point data processing.

The Second-Order Booth's algorithm is implemented for the high-speed multiplication operation. Then the multiplication operation for 32-bit integer is executed in 16 operation clock cycles.

3.Conclusion

In this paper, we described the internal structure of the 32-bit CMOS microprocessor V60, focusing on the pipeline structure and related issues.

Integration of Memory Management Unit and Floating-Point operation on a single chip decreased unnecessary overheads that were typical by multi-chip configuration in earlier generation of 32-bit microprocessors. The pipeline structure is an asynchronous type, i.e., execution time of each pipeline stage may vary, depending on instruction set function. Smooth operation in the pipeline was achieved by designing queueing interfaces among pipeline stages.

The structure was realized by an advantaged VLSI technology, based on 1.5um design rule, double-metal layer CMOS process. It provided not only high-integration density, but higher clock speed due to the shortened propagation delay. Table-2 shows the number of transistors for each pipeline Unit, and Table-3 shows the operation speed of typical circuits.

Unit	Transistors (X1000)	
PFU	4	
IDU	35	
EAG	15	
MMU	17	
BCU	39	
EXU	65	(except u-code ROM)
	200	(u-code ROM)
total	375	

Table-2 Transistors in Each Unit

functional circuit	operation speed (typical)
ALU (32-bit)	15nS
barrel shifter (64→32-bit)	13nS
microcode ROM (37-bit, 5184 steps)	24nS
TLB (16 entries)	36nS

Table-3 Circuit Operation Speed

The V60 chip integrates 375,000 transistors (or equivalent to 45,000 logic gates) on 13.92mmX13.80mm die size. It operates at 16MHz clock rate.

Acknowledgements

The limited space does not allow the authors to list all of the project members to design and implement V60. Instead, they would like to thank M.Suzuki, J.Iwasaki, Y.Sato, T.Iwata, and Y.Yano for their helpful suggestions. They are also indebted to H.Sasaki, K.Kani, J.Takashima, and K.Shimamura for their support throughout the project.

References

[1] C.H.Sequin, "Managing VLSI Complexity: An Outlook", Proceedings of IEEE, Vol.71, No.1, Jan.1983.

[2] C.A.Mead and L.A.Conway, "Introduction to the VLSI Systems", Addison-Welsy, Reading MA, 1980.

[3] "IEEE Standard for Binary Floating-Point Arithmetic", IEEE, Aug.1985.

[4] N.Weste and K.Eshraghian, "Principles of CMOS VLSI Design", Addison-Welsy, Reading MA, 1985.

[5] P.M.Kogge, "The Architecture of Pipelined Computers", McGrow-Hill Advanced Computer Series, 1981.

[6] Y.Yano, et.al., "A 32-bit Microprosessor with On-Chip Virtual Memory Management", ISSCC Digest of techinical papers, p.36-37; Feb.1986.

ADVANCED SUPER INTEGRATION

Tomotaka SAITO*, Tetsuya YAMAMOTO*, Tomohisa SHIGEMATSU*,
Ken-ichi NAGAO**, Sumio TAKEDA* and Yasoji SUZUKI*

* Integrated Circuit Division, Toshiba Corp.
** Toshiba Microcomputer Engineering Corp.

Abstract

A new ASICs (Application Specific ICs) design methodology, called Advanced Super Integration, is presented which allows short-time, one-chip integration of user-built systems created utilizing general-purpose LSI products as cores. Advanced Super Integration uses a hierarchical automatic layout system, a real chip simulator as a system verification method, advanced double-level aluminum CMOS technology, and an enriched CMOS cell library.

[1] Introduction

Continuing progress in device miniaturization is driving LSI packing density to a true "system-on-a-chip" level, where customization for a particular application becomes crucial in order to realize a maximum cost-performance ratio.

To meet the emerging demands for Application Specific ICs (ASICs) in a way that is easily accessible to the system designer, the authors developed a kind of core-based design method which was called Super Integration[1]. Super Integration (SI) has been aimed at reducing design cycle times, costs and risks. Double-level aluminum CMOS technology and a series of CAD software units are the major driving forces for SI implementation. CAD software includes TAPLAS[2],[3] (Toshiba Automatic Pattern Layout and Analysis System) as an automatic layout system, EMAP[4] (Extended Mask Analysis Program) as DRC (Design Rule Checker), and MACLOS[5],[6] (Mask Check Logic Simulator) as a logic simulator.

Furthermore, a new design methodology, called Advanced Super Integration, has been developed which is more progressive than conventional SI. Advanced SI is achieved owing to the introduction of the following new technology factors:

(i) Hierarchical aTAPLAS (aTAPLAS-II: advanced Toshiba Automatic Pattern Layout and Analysis System II), which can be used as an automatic tool to generate the layout of a whole chip.

(ii) Real chip simulator, which enables a designer to simulate a whole chip, including large-scale function blocks, such as an 8-bit microprocessor (MPU).

(iii) More advanced double-level aluminum CMOS process, which realizes high packing density and high speed performance.

Advanced Super Integration, whose scheme is shown in Fig. 1, has the following key features.

(1) Standard products, such as microprocessors, peripheral LSIs, RAMs, and A/D converters can be utilized as super macrocells.

(2) The super macrocells and custom circuit blocks, which provide a customer with design flexibility, are designed hierarchically with aTAPLAS-II, using a super macrocell library and a standard logic library.

(3) They are incorporated on the same chip and interconnected by double-level aluminum technology.

An example of an Advanced SI chip is given in Fig. 2. The chip functions as a hard disk con-

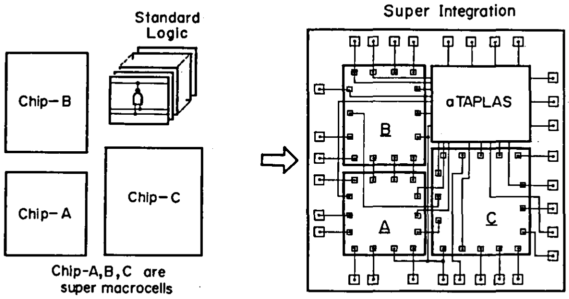


Figure 1 Super Integration Scheme

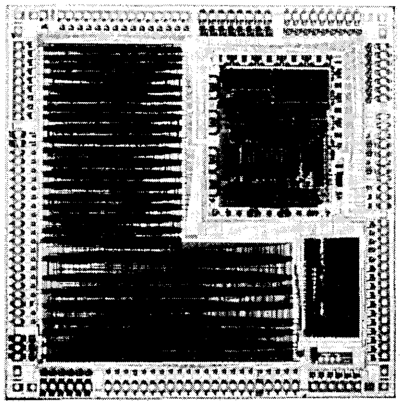


Figure 2 Hard Disk Controller Microphotograph

CH2345-7/86/0000/1008$01.00 © 1986 IEEE

troller (HDC) and includes the authors' CMOS version of an 8-bit MPU, 128-Byte Timing RAM (TRAM), and 7 k gates for standard logic. It was fabricated with double-level aluminum technology and the 2.0 μm design rule CMOS process. Die size is 95.1 mm^2.

Only 3 man-weeks were spent in designing this chip, from the circuit design to the mask data generation. This fact proves the advantage of using Advanced SI technology.

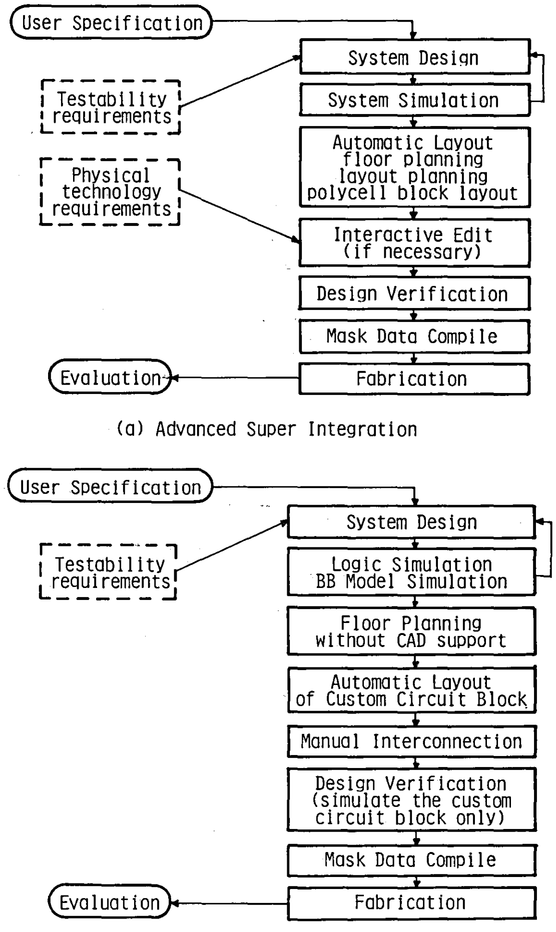
[2] Design Methodology

The design procedure for an Advanced SI chip is described in this section. Figure 3 shows the general flow in Advanced SI, in comparison with a conventional SI.

2.1 System Design

Advanced SI allows a designer to define a VLSI chip in a hierarchical net-list form of predefined CMOS cells. There are two kinds of cells in Advanced SI. The first category, called macrocells or polycells, implements SSI/MSI-level function. The second category, called super macrocells, is a cell for a large-scale function. Super macrocells implement a high degree of function, such as microprocessors and other peripheral LSI parts.

The designer can design an Advanced SI chip on an engineering workstation (EWS) and create a TDL (TEGAS Design Language) file. This file is uploaded to the mainframe. Figure 4 shows an example of a system-level schematic, designed on an EWS.


User Specification
System Design
Testability requirements
System Simulation
Automatic Layout floor planning layout planning polycell block layout
Physical technology requirements
Interactive Edit (if necessary)
Design Verification
Mask Data Compile
Evaluation
Fabrication

(a) Advanced Super Integration

User Specification
System Design
Testability requirements
Logic Simulation BB Model Simulation
Floor Planning without CAD support
Automatic Layout of Custom Circuit Block
Manual Interconnection
Design Verification (simulate the custom circuit block only)
Mask Data Compile
Evaluation
Fabrication

(b) Conventional Super Integration

Figure 3 Design Method Overview

2.2 System Simulation

When developing a conventional LSI, logic simulation is carried out for the entire system. However, since a Super Integration chip includes super macrocells as large-scale function blocks, a simple simulation method is not applicable to the SI chip.

One possible method is the gate-level software modeling which works so well for super macrocells embedded in the SI chip. The gate-level modeling normally becomes unsatisfactory for the SI chip, mainly because of the computing time needed to simulate the functions of microprocessors and other peripheral LSIs.

The solution for simulating the entire SI chip is to let super macrocells model themselves, which is called a real chip simulator. This real chip simulator is usable on several kinds of EWSs. Advanced SI allows a designer to implement the system verification with the real chip simulator. The real chip simulator does not need detailed gate-level modeling for super macrocells, but requires only their pin descriptions. Since the designer needs to see only the pin information for

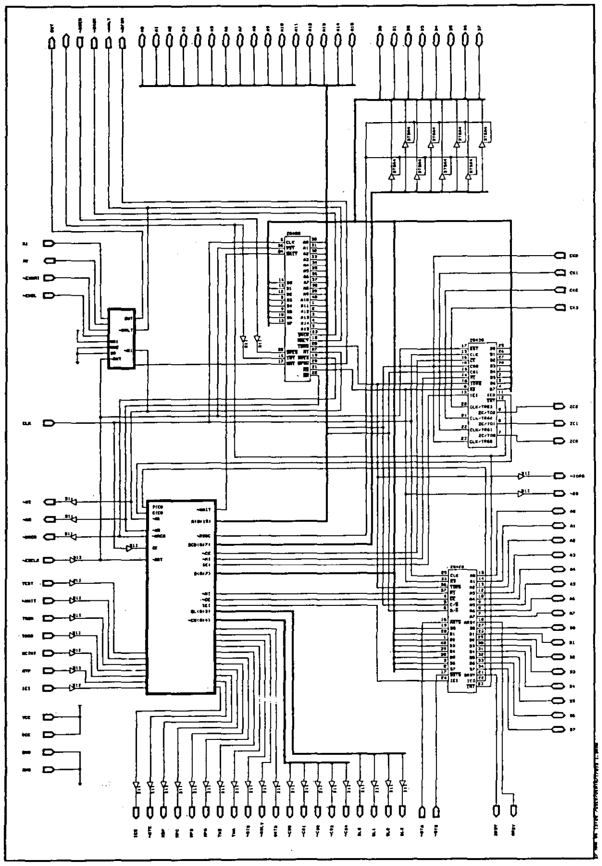


Figure 4 An Example Showing System-level Schematic on an EWS

super macrocells, as shown in Fig. 4, the access level is limited only to the information displayed on the EWS.

Figure 5 shows an example of a simulation output result obtained by the real chip simulator, corresponding to the waveforms from an 8-bit microprocessor.

2.3 Automatic Layout

In a conventional SI, the custom circuit block layout is generated automatically with TAPLAS. The generated custom circuit block can be treated as one of several super macrocells. Interconnection among super macrocells, including the custom circuit block, is performed manually using the graphic editing system.

In the advanced SI, the whole chip layout design is achieved hierarchically using a hierarchical automatic layout system, aTAPLAS-II[7]. This layout system can be divided into three parts: floor planning, automatic layout planning, and polycell block layout.

In floor planning, with the support of the mainframe computer, an engineer can pre-define the following items: 1) the method to partition the system into blocks, defined as clusters of polycells, 2) the global placement of blocks, and 3) the wiring routes for principal signals (e.g. power lines).

In automatic layout planning,[7] the polycell block dimensions can be estimated. Also, the global router attempts to minimize the die size, both by optimally assigning interconnection requirements to channels and by allocating block terminals not only on the block's boundary, but also in its internal area.

Polycell block layout proceeds after the determination of terminal positions, in the same manner as in TAPLAS. Then detailed inter-block routes proceeds. Finally, the detailed over-block routes are also laid out by aTAPLAS-II.

As shown in Fig. 6, layout for a HDC chip is performed automatically, after which part of the layout is modified manually. Some pin locations and related wiring were changed, due to the physical technology and area saving requirements. The result of these modifications was about 5 % reduction in die size.

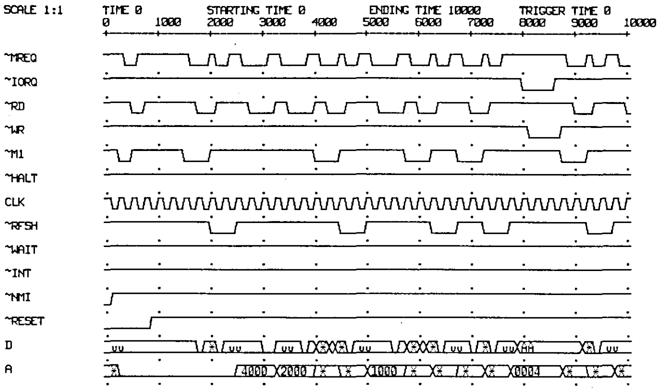


Figure 5 An Example Showing Real Chip Simulation Results

2.4 Design Verification

Design Verification need not be performed in case all the databases of an Advanced Super Integration chip are generated automatically.

However, if parts of the laout are carried out on the mask layout level, as in the case of the HDC chip, or if interconnections are altered manually, design verification tools have to be run. These tools include DRC (Design Rule Checker), ERC (Electrical Rule Checker), and a logic simulator.

For logic simulation, MACLOS (Mask Check Logic Simulator), unit delay switch level simulator, is applied to the transistor network extracted from the final mask database by EMAP (Extended Mask Analysis Program). In the near future, Toshiba is going to introduce an after-layout simulation using estimated routing capacitance and loading.

[3] Cell Library

Since Advanced Super Integration is generally used in a large-scale system, it is essential that a cell library be designed with CMOS technology, both to prevent too much of an increase in power consumption and to promote device miniaturization. The two kinds of cell libraries advocated, macrocell library and super macrocell library, use C^2MOS (Clocked CMOS) circuitry[8]. Thus each cell library has electrical characteristics of low power consumption, high-speed performance, and high packing density.

Each standard product is designed according to the same drawing rule. Its mask data is to be linearly shrunk on the drawn-level data in order to obtain reasonable die size and high performance. Utilizing the drawn-level data for a standard product as drawn-level data for a super macrocell in Advanced SI, the global and most updated shrinkage rate can be selected; even if the actual shrinkage rate for the standard product is different from others.

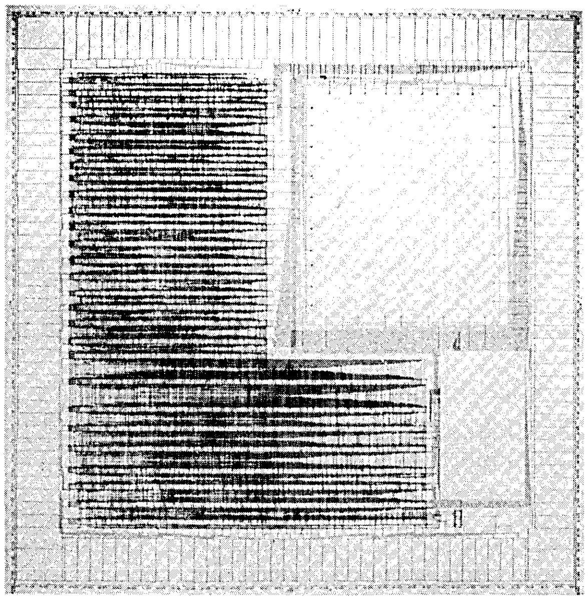


Figure 6 HDC Chip Layout Result

The authors macrocell library includes SC library and SI library. The two are functionally equivalent, but not electrically. In regard to gate propagation delay time, the SI library is inferior by at worst 20 % to SC library according to SPICE simulation results. However, in regard to packing density, the SI library is about 1.5 times as much as the SC library. The reason is that the SI library has smaller dimensions than the SC library and allows over-block routes, as shown in Fig. 7. These libraries allow a designer to select and use either of the two, considering a trade-off between die size and AC performance.

In the above real chip simulation, the performance data for each macrocell, which includes $T_{PHL}$and T_{PLH} equation with variables for fan-out and capacitive load, is needed in order to perform timing verification. Figure 8 shows an overview of the procedure to extract performance data for macrocells automatically from the mask level layout. Transistor network and load information are generated from the layout data and then transformed into SPICE format, based upon the process parameters. SPICE simulation results are compiled in the LDDL (Library Data Description Language) format. The LDDL file is uploaded to the mainframe computer and the EWS. This file is used as performance data for logic simulator and timing verifier.

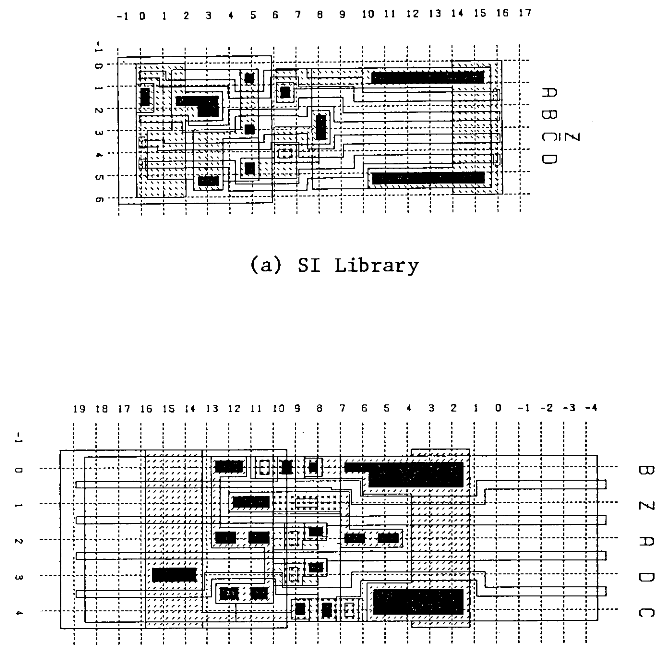
The super macrocell library includes the complete 8-bit MPU family, DMA controller, UART, I/O controller, VDP, FDC, CRT controller and other peripheral controllers. In addition, 16k/8k/2k/1k-bit RAMs and 8-bit A/D converters are included in the library. 28 super macrocells are available at 1.5 µm / 2.0 µm design rules. These super macrocells are functionally compatible with the corresponding standard LSIs. They have a large AC margin to specification and conform very well with Advanced SI approach. The authors have been designing these super macrocells, without stripping away the bonding pads and large output buffers, considering compatibility with existing standard LSI parts.

[4] Testability Design

In an Advanced SI chip comprising several super macrocells, testability, especially for embedded super macrocells, is one of the major concerns. As shown in reference [1], a conventional SI can gain direct access to each of embedded super macrocells with "cutset circuitry", which includes multiplexers, transceiver logic and other circuits modifying the peripheral circuits around the terminals. In testing a conventional SI chip, entire chip tests follow testing individual super macrocell by a developed test program and testing custom circuit block by a newly-developed test program.

The new test method described in the following has been considered and adopted. This new cutset logic includes additional circuits, which let embedded input signals to the embedded MPU lead to the outside. Connecting the led signals to the corresponding terminals of an existing standard MPU allows a user to emulate the embedded MPU using the standard MPU.

Test efficiency for both user and vendor is achieved with this cutset logic.



(a) SI Library

(b) SC Library

Figure 7 Complex Gate Polycell Layout

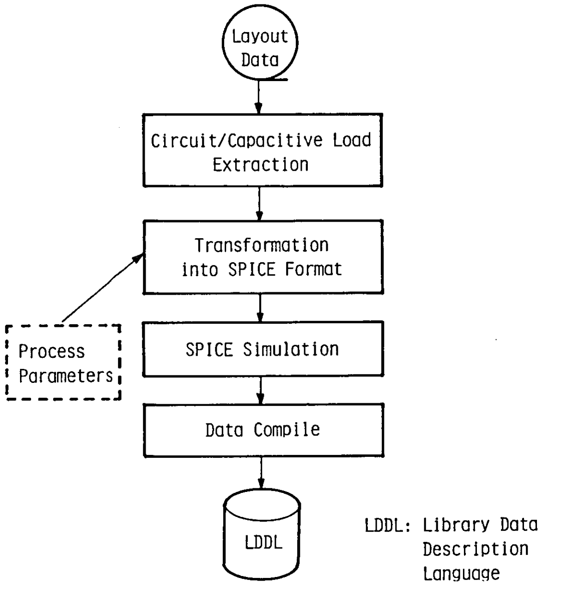

Layout Data
Circuit/Capacitive Load Extraction
Transformation into SPICE Format
Process Parameters
SPICE Simulation
Data Compile
LDDL
LDDL: Library Data Description Language

Figure 8 Procedure to Extract Macrocell Library Performance Data

[5] Advanced Super Integration Performance

As described in Section 3, the cells and all the standard products for logic LSIs are designed according to the same drawing rule, each of which has a different design rule in the fabrication process. So, it is possible for all standard procucts to be used as super macrocells. The standard products are released after they are evaluated in the same fabrication process as Advanced SI.

Since an Advanced SI chip normally uses databases whose design has been finished and whose performance data has been guaranteed, the Advanced SI chip performance greatly depends on that of the super macrocells. Generally, the LSI performance is determined by both its fabrication process and design level. Since the design level for super macrocells has been fixed, the Advanced SI chip performance mainly depends on its fabrication process.

The relationship between performance and fabrication process will be explained in the following. The progress in the fabrication process has covered the range from 3.5 μm to 1.5 μm design rule. The authors' consistent scaling philosophy allows these processes to be applied to all the super macrocells and Advanced SI chips. Figure 9 illustrates the progress by an example of an 8-bit MPU which is the authors' CMOS version. In 2.5 μm, 2.0 μm and 1.5 μm design rule processes, respectively, the maximum operation frequency is obtained at around 1.3, 2.8, 3.4 times as many as in 3.5 μm process. The power supply current is reduced by around 30 %, 45 %, 50 %, compared with that in the 3.5 μm process.

Individual 2.5 μm, 2.0 μm and 1.5 μm design rule processes allow developing an SI chip embedding 4 MHz, 6 MHz, and 8 MHz version, respectively, of the MPU.

[6] Estimation Results

A design procedure for Advanced Super Integration has satisfied the purpose of reducing the design cycle times, costs and risks, through design automation. As an example of an HDC chip design, the results of an Advanced SI chip design are discussed in the following.

(1) Design cycle time

As shown in Fig. 10, it took only 3 man-weeks to complete the design, from starting the circuit design to making up the EBMT (Electron Beam Magnetic Tape) for mask fabrication. As mentioned before, the layout on the chip corner was accomplished manually at the mask layout level. Also, parts of the interconnections were manually altered, so that an extra 4 man-days were expended on interactive editing.

It is estimated to take nearly one year, for three or four talented engineers, when design is performed by conventional hand-craft method. The approach by SC (Standard Cell) or SOG (Sea of Gate) seems close to the same time as in the Advanced SI approach. However, since it is impossible to describe super macrocells, e.g. and 8-bit MPU, by gate-level modeling, neither the SC nor the SOG approach is generally practical in this case. Advanced SI design method and a conventional SI design method are a practical way to realize a chip which include existing standard LSIs. In the conventional SI, it took 4 man-weeks to complete the design, according to reference [1]. Advanced SI realized 25 % reduction in design cycle time.

(2) Die Size

Automatic layout on the HDC chip took around 1.4 CPU-Hours (on a 7-MIPS computer). The die size obtained by it was 5 % larger than the final die size, which was 95.1 mm^2, as before. Figure 11 shows the die size prediction for the HDC chip with several design methods at both the 2.0 μm and 1.5 μm design rule levels, where SOG is available for the HDC chip only in the 1.5 μm design rule process. It is clear, in Fig. 11, that Advanced Super Integration is superior to SC and SOG, although it is inferior to the hand-craft design in the die-size point of view. This high density comes from usage of super macrocells, which had been manually designed, in order to minimize the number of transistors and to integrate with high packing density.

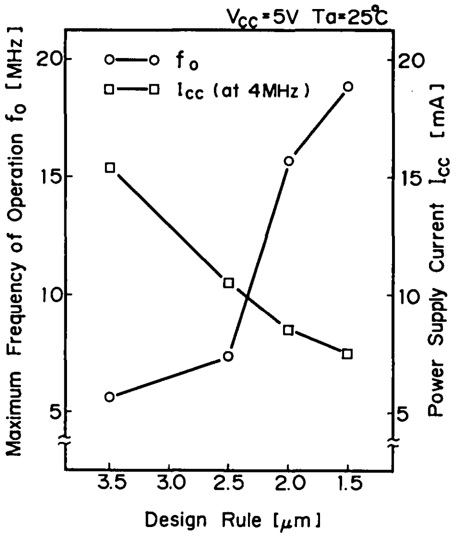


Figure 9 An 8-bit Microprocessor Performance

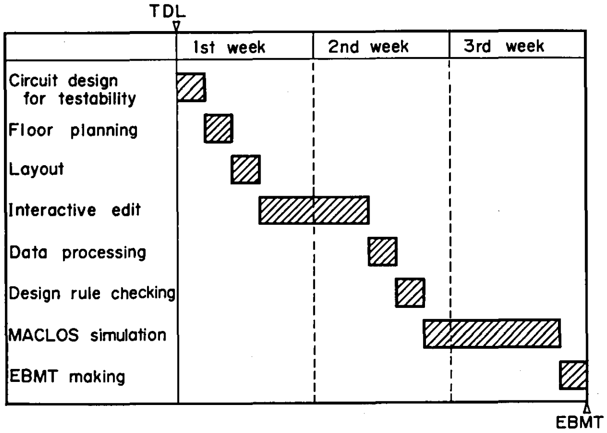


Figure 10 HDC Chip Design Cycle Time

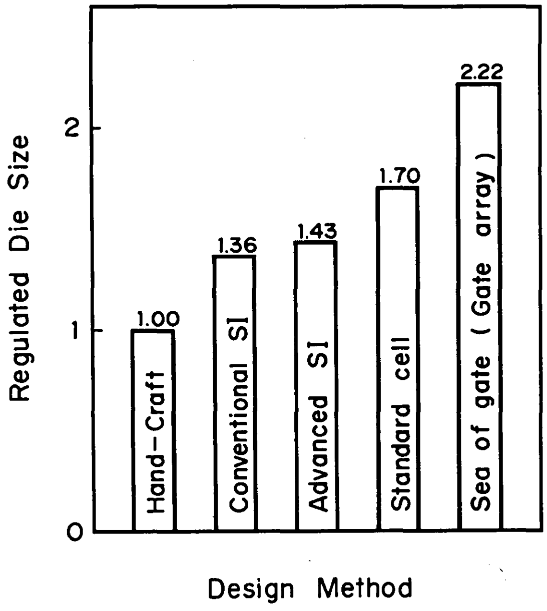


Figure 11 Die Size Comparison

Conclusion

A design method, called Advanced Super Integration, that utilizes high performance microcomputers as cores and that is particularly suitable for ASICs, is proposed and discussed. Advanced SI advantages are discussed in comparison with a Conventional SI. Some of them, especially in regard to the design methodology, are a whole chip simulation on an EWS and an automatic layout by hierarchical aTAPLAS, which realizes both very short design cycle time and high packing density. High speed performance is also achieved, using 1.5 µm double-level aluminum CMOS technology. This design method seems to provide a new field for VLSIs.

Acknowledgement

The authors would like to acknowledge the continuing guidance and encouragement of H. Egawa, Dr. T. Tarui, K. Suzuki, S. Takisawa, K. Yoshida, M. Ishibashi and M. Hirasawa.

References

[1] K. Nagao et al. "Super Integration," IEEE 1985 CICC, pp. 267-271.

[2] Y. Shiotari et al. "New TAPLAS for Full Custom C^2MOS LSI Design," IEEE 1982 CICC, pp. 111-114.

[3] Y. Nakada et al. "High Packing Density LSI with Interactive Facilities," ISSCC Digest of Tech. Papers, pp. 41-46, 1974.

[4] T. Mitsuhashi et al. "An Integrated Mask Artwork Analysis System," Proc. of 17th DA Conf., pp. 277-284, June, 1980.

[5] M. Takashima et al. "Programs for Verifying Circuit Connectivity of MOS/LSI Mask Artwork," Proc. of 19th DA Conf., pp. 544-550, June, 1982.

[6] M. Kawamura et al. "Logical Verification of LSI Mask Artwork by Mixed Level Simulation," Proc. of IEEE International Symp. on Circuit and System, pp. 1021-1024, May, 1982.

[7] M. Yamada et al. "A Multi-layer Router for Standard Cell LSIs," Proc. of ISCAS 85, pp. 191-194, 1985.

[8] Y. Suzuki et al. "Clocked CMOS Calculator Circuitry," ISSCC Digest of Tech. Papers, pp. 58-59, 1973.

A 16-BIT MICROPROCESSOR WITH MULTI-REGISTER BANK ARCHITECTURE

Hideo MAEJIMA*, Hiroyuki KIDA*, Tan WATANABE**,
Shiro BABA*** and Keiichi KURAKAZU***

* Hitachi Research Laboratory, Hitachi,Ltd.
4026 Kuji-cho, Hitachi-shi, Ibaraki 319-12, Japan
** Systems Development Laboratory, Hitachi,Ltd.
*** Musashi Works, Hitachi,Ltd.

ABSTRACT

This paper describes a newly develped microprocessor architecture, which we call a multi-register bank architecture. To accelerate a task switching speed, a microprocessor includes multi-register banks, each of which consists of sixteen general purpose registers.

In the multi-register banks, there are two types of banks. One is a global register bank assigned to an individual task and the other is a ring register bank asigned to a task called by a procedure call like subroutines.

The microprocessor incorporates an orthogonal instruction set with complex and reduced formats to achieve high code efficiency for high-level languages.

INTRODUCTION

Rapid progress in semiconductor technology, in particular, MOS technology, has made it possible to fabricate highly integrated microcomputers that exceed 100 thousand MOS transistors. Microcomputers with many peripheral functions are appearing on the market which reduce system cost and increase system performance. Presently, 8-bit machines are the most popular because of their excellent cost/performance ratio.

However, many applications are beginning to require new generation microprocessors with higher performance which advanced semiconductor technology helps to provide. Several 16-bit microprocessors have been utilized in high-end applications for controllers, and development focus is moving to 16-bit machines.

A real time processing capability is vital for microprocessors serving as controllers. In the real time processing, tasks are switched frequently according to events, e.g., interruptions from I/O devices and software interruption. In microprocessors, not only execution speed of instructions, but the speed of task switching must be improved to achieve high performance at the system level. Additionally, software productivity must be improved. To do this, application programs should be described using popular, high level languages and microprocessor architecture should help in their effective execution.

In this paper, descriptions are given of an architectural concept, and an architecture for real time processing and for a high level language support of a 16-bit microprocessor.

ARCHITECTURAL CONCEPT

HIGH SPEED TASK SWITCHES

One of the main jobs of the conroller is control for I/O devices. Interruptions from I/O devices drive the corresponding tasks. They occur frequently and new tasks are then driven each time. There are many cases in which switching time of the tasks dominates system performance, even though instructions are executed at high speed. Thus, breakthroughs in microprocessor archiecture are required.

SOFTWARE PRODUCTIVITY

Another consideration in microprocessor architecture is software producivity. In the past, mainly assembling languages have been used to describe programs in control applications, as only they made it possible to achieve the maximum performance through their machine dependent functions. As control applications have now gone on to a higher level, software productivity becomes more important. Instead of the assembling languages, high level languages are becoming popular among many software designers.

PROCESSOR ARCHITECTURE

BASIC ARCHITECTURE

Our 16-bit microprocessor is based on a general register machine with a multi-register bank architecture. No distinction between address registers and data registers exists in the general purpose registers. A set of general registers composes a register bank, and plural banks exist in the microprocessor. The addressable memory space of the microprocessor is linear and currently 16M bytes with 4G bytes anticipated in the future. Byte-oriented instructions based on orthogonal and variable length concepts include complex and reduced formats, in order to achieve both high level functions and simple operations. Memory to memory operation is available in most of the instructions.

REGISTERS

The microprocessor registers are shown in Fig.1. It consists of sixteen 32-bit general purpose registers and six 32-bit special purpose registers, a 16-bit status register and three 8-bit special purpose registers. The 32-bit special purpose registers include a program counter (PC), a supervisor stack pointer (SSP), an exception vector base register (EVBR), a RAM relocation base register (RRBR), a bank stack pointer (BSP) and a current bank number register (CBNR). The 8-bit special purpose registers include a bank mode register (BMR), a global bank number register (GBNR) and a valid bank number register (VBNR).

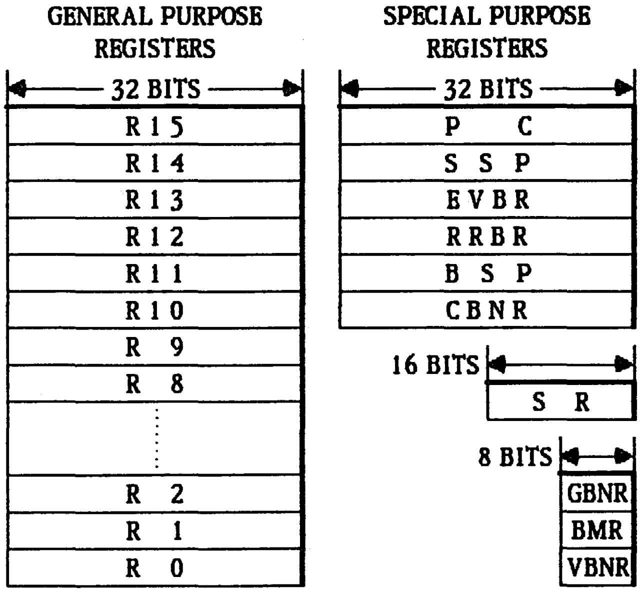


Fig.1: Microprocessor Registers

MICROCHIP ARCHITECTURE

The microprocessor is a microcoded processor and its general purpose registers are implemented in a centralized RAM, which we call a RAM-based architecture. The microprocessor (Fig.2) consists of a memory access processor (MAP), an instruction execution processor (IEP) and a RAM. Each processor consists of an execution unit and a control unit that includes a microprogram memory (ROM). The internal buses are 32 bits wide to ensure high performance.

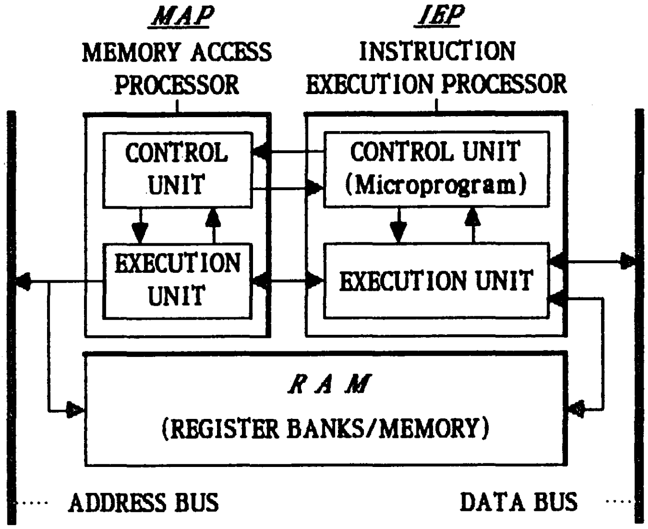


Fig.2: Microprocessor Architecture

MULTI-REGISTER BANK ARCHITECTURE

CONCEPT OF NEW ARCHITECTURE

To achieve high speed task switching, one problem encountered is overhead time for saving contents of the general purpose registers in a main memory and resuming them. In a microprocessor with a single register bank architecture (a conventional architecture), the sequence shown in Fig.3 is performed when one task i switches to another task j. In this case, the microprocessor

--- Saves the contents of the general purpose registers for task i in the main memory.
--- Loads new data from the main memory to the general purpose registers for task j.
--- Executes task j.
--- Resumes the data for task i, which were saved in the memory before task j started to execute, and returns them to the general purpose regis-

ters.
--- Starts task i again.

When the microprocessor executes multi-tasks, task switches occur frequently. These reduce system performance, even if execution speed of instructions is fast. To cut the task switching time, the volume of data transferred between the general purpose registers and the memory should be minimized. To realize this, a multi-register bank architecture has been developed. A register bank, which consists of sixteen 32-bit general purpose registers, is given to each task. Saving and resuming the registers disappear. Thus, the switch between tasks i and j can be performed immediately.

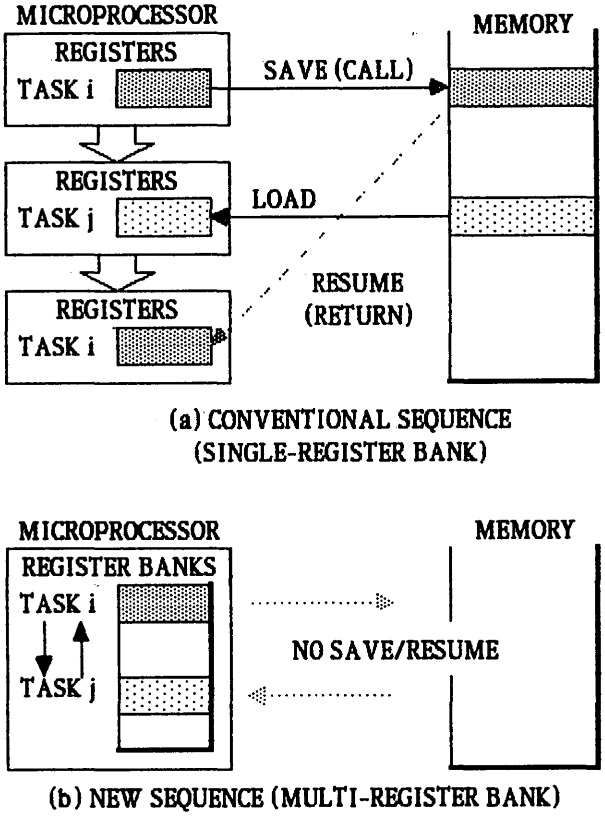


Fig.3: Task Switching Mechanism

PROGRAMMABLE REGISTER BANK

The number of register banks differs with the applications. So, it should be large enough and also programmable by software. The number of register banks can be set to 2, 4, 8, or 16. Therefore, the register banks are implemented in a RAM. When the number is small, for example, in a single task, the rest of the RAM is available to be used as an on-chip high speed memory, as shown in Fig.4.

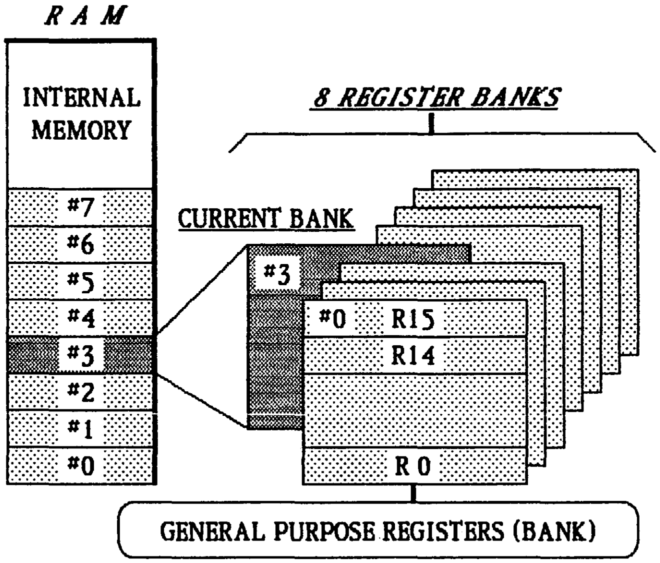


Fig.4: Multi-Register Bank Architecture

GLOBAL REGISTER BANK

One event driven task is independent of others in many cases. One register bank is given to each task. The register bank is designated a Global Register Bank (GRB). In order to switch from one register bank to another, the Global Bank Number Register (GBNR) must be updated by the software (a special instruction).

The register bank switching mechanism is as follows. When a certain event caused by a software interruption (e.g. macro instruction call) or a hardware interruption (e.g. interruption from I/O devices) occurs, an operating system is called and task switching is performed. In Fig.5, GRB #1 is given to the current

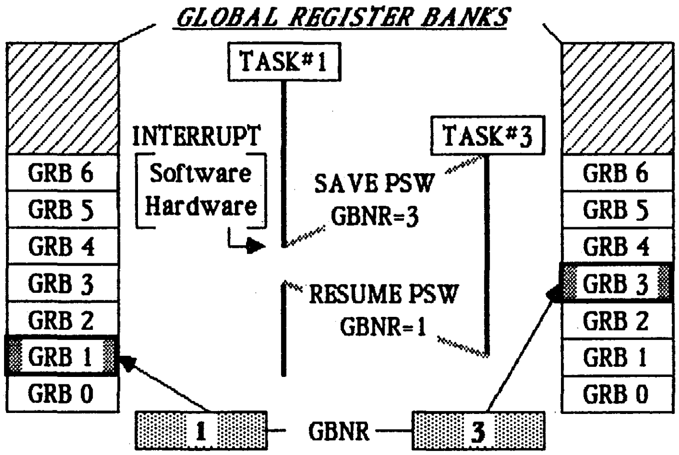


Fig.5: Operation of the Global Register Banks (GRBs)

task #1 and GRB #3 is given to the next task #3. A switch from the current task #1 to the next task #3 is performed by saving the current PSW (Program Status Word; a program counter and a status register) in a predetermined area of the main memory and updating the contents of the GBNR from 1 to 3. It is unnecessary to save the contents of the current GRB, so task switching time is minimized. To return from task #3 to task #1, the contents of the GBNR are updated from 3 to 1 and the old PSW is resumed.

RING REGISTER BANK

In programs described with high level languages, procedure calls and returns often occur. In this case, a stacking mechanism is required. The multi-register bank architecture can be applied to this mechanism, as shown in Fig.6. The top part of the stacks (on the main memory) should be maintained in the multi-register banks, like an on-chip stack memory. These register banks are called Ring Register Banks (RRBs).

When one of the GRBs requires a new register bank via a procedure call, the lowest part of the RRBs is given to the called task. The rest of the RRBs is given to the following tasks. Addressing modes of the microprocessor support three types of register accesses, the GRB, the current RRB and the previous RRB.

The maximum number of RRBs is eight. When an overflow occurs in them, the contents of the least recently used RRB are saved in the main memory and data for the new task are loaded in the open RRB. When an underflow occurs, all of the RRBs are open. In this case, the most recently used data in the memory are loaded in one of the RRBs. In execution of some benchmark programs, the occurrence ratio of overflows or underflows is less than 10%.

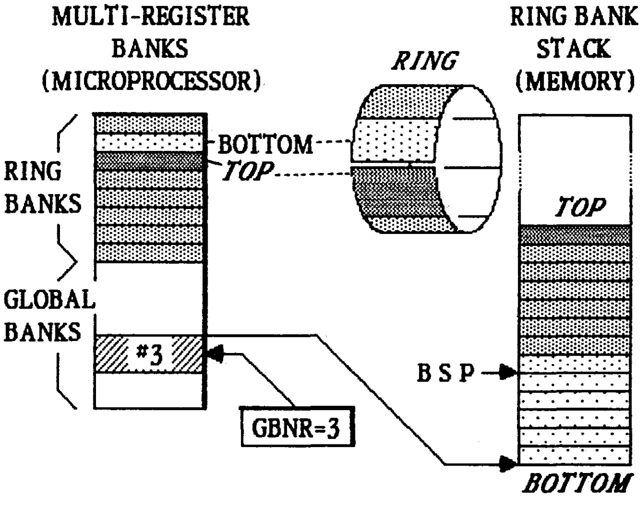


Fig.6: Operation of the Ring Register Banks (RRBs)

ORTHOGONAL INSTRUCTION SET

BYTE ORIENTED INSTRUCTION FORMAT

The instruction format is designed on an orthogonal and variable length concepts. A basic instruction is availble to specify two memory operands. As shown in Fig.7, basic instructions are composed of four fields. The first is an operation code (OP) which specifies the processing function. The second is an operand size (sz) field which specifies the size of the processed operand. The third is an effective address (EA) field which consists of seven bits and specifies the location of the processed operand. An accumulator bit (A) is used to specify an accumulator (R0) as the implied second operand. When this bit is one, the accu- mulator becomes the second operand and the second EA field is omitted.

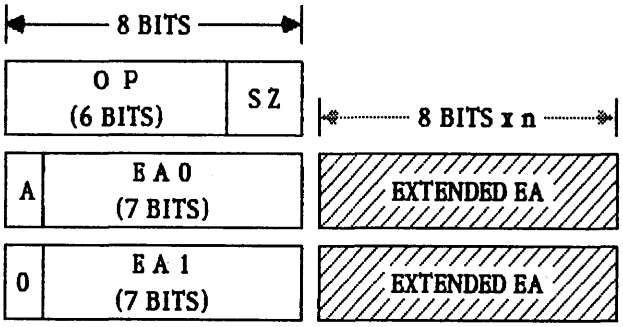


Fig.7: Orthogonal Instruction Format

ADDRESSING MODES

Table 1 shows the thirteen addressing modes of the microprocessor, which include a current bank mode and a previous bank mode for the RRBs. Both of them become prefixes of the following EA fields. The EA field with the prefix makes it possible to specify a register which belongs to the current task or the previous task.

Other addressing modes are for the GRBs. In the addressing modes, sizes of displacement, absolute address and immediate data are defined freely by Sd, Sa and Si fields, respectively. Each field consists of two bits, and one size of 8-bits, 16-bits or 32-bits which is selected by the compilers to optimize object codes.

COMPLEX INSTRUCTIONS

As shown in Fig.8, basic instrucions consist of three formats, 0-operand, 1-

Table 1: Addressing Modes of the Microprocessor

No.	ADDRESSING MODES	EA CODES	EXTENDED EA CODES		
1	REGISTER DIRECT	100 Rn	NOTHING		
2	REGISTER INDIRECT	0Sd Rn	DISPLACEMENT (0,8,16,32)		
3	REGISTER INDIRECT WITH AUTO-INCREMENT	101 Rn	NOTHING		
4	REGISTER INDIRECT WITH AUTO-DECREMENT	110 Rn	NOTHING		
5	IMMEDIATE	11100Si	IMMEDIATE (8,16,32)		
6	ABSOLUTE ADDRESS	11101Sa	ADDRESS (8,16,32)		
7	SCALED REGISTER INDIRECT	11110Sd	Scale Rn	DISP	
8	INDEXED REGISTER INDIRECT	1111100	0LSd Rn	Scale Rn	DISP
9	PROGRAM COUNTER RELATIVE WITH INDEX	1111101	0LSd	Scale Rn	DISP
10	PROGRAM COUNTER RELATIVE	1111101	10Sd Rn	DISP	
11	REGISTER INDIRECT DOUBLE	1111110	SdSd Rn	DISP	DISP
12	CURRENT BANK	1110000	ARBITRARY EA CODES 1-11		
13	PREVIOUS BANK	1110100	ARBITRARY EA CODES 1-11		

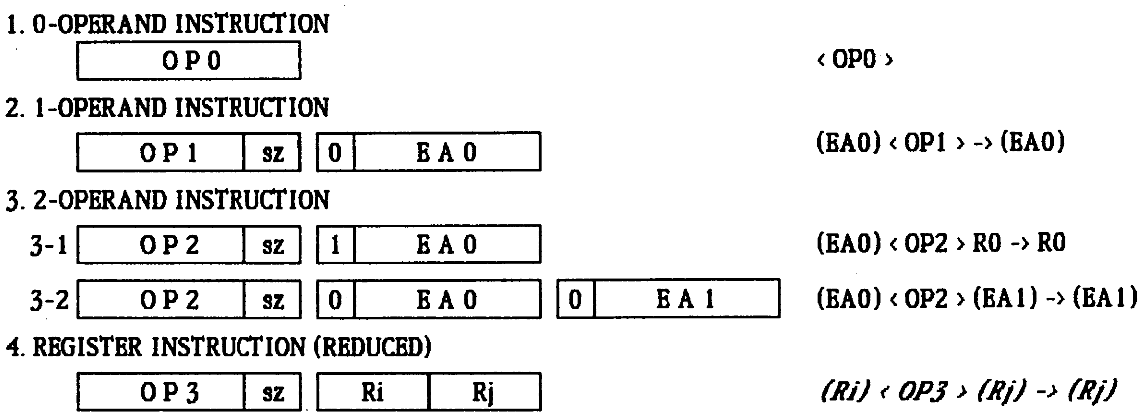


Fig.8: Complex and Reduced Instructions

operand and 2-operand instructions. The 0-operand instruction consists of one byte with no sz field. The 1-operand instruction consists of a minimum of two bytes, one byte for OP-code and sz field and the other byte for an EA field. The 2-operand instruction consists of a minimum of two or three bytes. If the MSB of the first EA field, which is an accumulator bit, is one, the second EA field is not needed, because the acumulator (R0) is implicitly specified. If the MSB of the first EA field is zero, the second EA field is needed.

REDUCED INSTRUCTIONS

Although the basic instructions are orthogonal, the microprocessor has a reduced instruction format as for register-to-register operations. This feature effectively ensures high code efficiency in object codes and also high performance.

CONCLUSION

A newly developed architecture for a 16-bit microprocessor has been described. Its multi-register bank architecture should be effective for executing event driven tasks and procedure calls, by through a combination of global register banks and ring register banks. The microprocessor can be fabricated with advanced CMOS technology to operate at a 16 MHz machine cycle. It should be the nucleus of a 16-bit single-chip or system integrated microcomputer with many peipheral I/O functions included in the same chip.

ACKNOWLEDGEMENTS

The authors would like to extend their sincere thanks to Y. Hatsukano, H. Yasuda, J. Nakajima and I. Masuda for giving them the opportunity to develop this microprocessor and for useful suggestions during the work.

REFERENCES

[1] Dave Bursky, "MOS/digital/Analog mix= low-cost, high performance single chip uCs," Electronic Design, Vol.27, No.13, pp.43-48, Jun. 1979

[2] H. Maejima, K. Katsura, et al., "The VLSI Control Structure of a CMOS Microcomputer," IEEE Micro, Vol.3, No.6, pp.9-16, Dec. 1983

[3] David A.Patterson and Carlo H.Sequin, "A VLSI RISC," IEEE Computer, Vol.15, No.9, pp.8-18, Sept. 1982

SOFTWARE ORIENTED APPROACH FOR SUPERCOMPUTER DESIGN

Kenichi MIURA*, Yoshiyuki TANAKURA**, and Sachio KAMIYA**

* FUJITSU AMERICA INC., Computational Research Dept.,
3055 Orchard Drive, San Jose, California, 95134-2017, USA

** FUJITSU LIMITED, Software Division,
104 Miyamoto, Numazu-shi, Shizuoka, 410-03, JAPAN

ABSTRACT

The most important point in the design of supercomputers that achieve high-speed execution by parallel processing is to make efficient use of the inherent parallelism in the programs. This means the software-oriented approach is very crucial. Fujitsu's vector processors, the FACOM VP systems, are supercomputers that are designed along this line of approach. Our approach to development and its effectiveness is presented in this paper. Future design trends are also discussed.

1. INTRODUCTION

Recently, in the scientific and engineering fields, the demands for faster and easier to use computer systems is rising because of increased computation needs. This demand for higher performance has no ceiling.

General-purpose computers adopt the concept of serial processing. As we are close to the basic limitations in the device and circuit technology, no drastic improvement in performance is expected in this approach alone. It is therefore necessary to incorporate some kind of parallel processing techniques to obtain increased performance. This is the basis for the development of vector processors.

Powerful parallel processing, however, cannot be achieved by hardware alone. Various techniques in the architectural design are required to exploit the maximum amount of parallelism. This indicates that the previous hardware-oriented approach is not sufficient and that a software-oriented approach is also necessary. We believe this trend becomes more manifest as programs contain higher and higher degree of parallelism.

This paper first presents the problems that must be solved to achieve high speed execution by parallel processing. Next, the actual approaches taken in the development of Fujitsu's vector processors, the FACOM VP systems, in order to cope with these problems is introduced as an example. Finally, the future of high-speed execution by parallel processing is briefly discussed together with the results of our experiments.

2. FASTER EXECUTION BY PARALLEL PROCESSING

The two kinds of parallelism that scientific and engineering programs have are: array operations executable in parallel (SIMD type), and different procedures executable in parallel (MIMD type). In addition, there are various levels of operations and procedures.

Execution performance, P, obtained by parallelization is expressed by the following formula:

$$P = \frac{1}{1-\sum_i r_i + \sum_i [r_i/\alpha_i]} \qquad (1)$$

where i indicates each part of a program which can execute in parallel and r_i indicates the ratio of the relative computation time of each part to the total time when parallelization is not applied. α_i indicates the effect of parallelization for each part.

The ideal value of α_i is the ratio of the peak performance inherent in the hardware for parallel processing to that for non-parallel processing. Unfortunately, α_i usually contains various kinds of degrading factors. For example, if an initialization or termination process is involved in parallel execution, α_i contains a factor caused by that process which degrades the execution performance. If data transfer is involved, α_i will also be affected by the side-effect of that data transfer. If each part of a program can do parallel processing in two ways at the same time, α_i will reflect the effect of such parallel processings.

The values of r_i and α_i largely depend on the types of parallel processing as well

CH2345-7/86/0000/1020$01.00 © 1986 IEEE

as the specific features in architecture and software.

As shown in (1), to get high performance by parallel processing, it is necessary to maximize $R=\sum_i r_i$ and each α_i. In vector processors, R is the vectorization ratio, and each α_i is the performance improvement factor in each vectorizable part. In a multiprocessor system, R is the parallelization ratio, and each α_i is the approximate value of the number of processors which do parallel execution. (The actual value is smaller than this because of overhead in task distribution and synchronization.) Performance degradation by other factors such as overhead caused by data transfer may also have to be considered, depending on the hardware configuration of processor interconnection.

The following are necessary to increase R and each α_i:

(a) high speed circuit and packaging technology
(b) incorporation of parallelism in the architecture
(c) system software to extract the maximum amount of parallelism
(d) an efficient algorithm for parallel execution

As far as the architectural design of a supercomputer is concerned, items (b) and (c) are the most important. In other words, to what degree parallelism can be extracted from actual application programs and how efficiently the extracted parallelism can be executed. To maximize these factors, it is necessary to survey actual application programs, then design architecture and soft ware based on the survey. We define this approach as 'software-oriented approach' throughout this paper.

3. DESIGN OF VECTOR PROCESSORS

This section discusses the approach to the design of processors which efficiently execute the parallelism of the SIMD (vector) type, using as an example the method we adopted in designing the FACOM VP systems.

3.1 Actual method

As is evident from (1), in vector processors, it is important

(a) to increase the vectorization ratio as much as possible, and
(b) to design architecture which can execute array type parallelism as fast as possible.

With regard to point (a), it is important to increase the range of operations executable as vector operations, not to mention simple operations such as Ai=Bi+Ci, i=1,..,N. At the same time, software must be able to detect all corresponding vector operations, and to confirm that vectorization has a positive effect on performance.

With regard to point (b), it is important to identify the bottlenecks in actual application programs and to design an architecture which eliminates them.

3.2 Survey results of application programs

We thoroughly examined actual FORTRAN programs from the point of view of vectorization, according to the ideas mentioned above. Over 1000 programs were examined. See Ref. [1] for details of the results.

We will now discuss the results of these analyses from several points of view:

- vectorizable operations
- ratios of operations
- patterns and ratios of memory accesses
- vector length and number of active vectors

a) Vectorizable operations

Besides simple arithmetic operations in a simple DO loop, vectorizable operations include operations that contain complicated control structures like IF statements, and operations expressed in several statements which can be changed into one vector macro operation. Operations containing IF statements appear in about 30% of DO loops. In some cases, these consume as high as 90% of CPU time. This indicates a need to somehow provide an effective means to speed up the conditional operations.

b) Ratios of operations

In arithmetic operations, the number of addition/subtraction, and multiplication operations is about equal. Other operations like division, comparing(logical) operations, vector macro operations, and intrinsic functions occur at almost the same rate. The linked operations are frequently performed for addition/subtraction and multiplication, or in most other cases they can be executed concurrently.

c) Patterns and ratios of memory accesses

The number of data movement operations required per arithmetic operation is about 0.8. In other words, 8 data movement operations are required to execute 10 arithmetic operations. In vector data, there are contiguous data (73%), constantly strided data (19%), and data located at random positions (8%, which is also significant).

d) Vector length and number of active vectors

The vector length widely varied among programs. The number of active vectors at any one time is usually 3 to 4, and seldom exceeds 30. But in very rare cases, it exceeds 70.

3.3 Requirements and the architecture adopted in the FACOM VP systems

The results described in 3.2 clearly indicate certain requirements for the design of efficient vector processors. This section describes the requirements and an outline of the architecture adopted in the FACOM VP systems. For further details, see Ref. [2,3].

a) Types of operations

The basic requirements are to provide all instructions that correspond to recognizable vector operations and to make sure that these instructions operate efficiently.

To realize this, the FACOM VP system provides not only instructions for simple arithmetic operations and vector macros, such as summation operations, but also instructions for manipulating various types of vector data, such as data editing and data retrieving.

In addition, for greater efficiency, the FACOM VP provides mask bits and mask functions which enable or disable the execution of operations on individual vector elements. The total number of vector instructions is 83.

b) Balance of computational devices and parallel execution

The basic requirements are that there be a good balance between addition-related devices and multiplication-related devices, and that these devices can be executed in parallel. It is also essential to employ an architecture which processes multiple vector elements simultaneously at high speed.

Two alternatives meet these requirements: the pipeline type and the processor array type. We have chosen the pipeline type because it has a good affinity with the architecture of general-purpose computers. Four pipelines are provided as computational devices: add/subtract, multiply, divide, and mask. Any two pipelines of arithmetic operations and the mask pipeline are executable concurrently. Since the mask operation is an auxiliary one, it is executed behind other arithmetic operations to obtain higher performance.

c) Access to memory and parallel operation

The Basic requirements are to ensure sufficient data for full operation of arithmetic and logical pipelines, and to allow a wide variety of functions in load/store instructions for the various patterns of data accesses that occur in application programs.

In the FACOM VP systems, there are two memory access pipelines that continuously feed data to two arithmetic/logical pipelines that can be executed in parallel. These two memory access pipelines are also executable in parallel. When these pipelines are executed in parallel, the order of access to memory data must be consistent with the scalar processing. To realize this, the FACOM VP systems have two serialization functions:

- pipeline identification (pipeline ID)
- POST/WAIT instruction insertion

The pipeline ID function assures the order of vector instruction execution by assigning the same ID to the instructions whose execution order must be maintained. Instructions without an ID are executable in parallel, except for those restricted by hardware register reservation mechanism. This drastically lessens the overhead caused by serialization. The POST/WAIT instruction insertion has the same function as in an operating system. It is mainly used for synchronization between the scalar unit and the vector unit.

The software can achieve the optimal serialization by using these functions.

d) Vector register capacity and configuration

It is desirable to have as large a register capacity as possible and as many registers as possible. Of course, there are limitations in actual hardware implementations.

The vector registers in the FACOM VP systems are dynamically configurable, although total capacity stays the same. The vector register may be reconfigured when vector length is changed, that is, for each DO loop execution. The VP-200, for example, takes the following configurations: 32 (length of a vector register in a 64-bitword) x 256 (number of vector registers), 64 x 128,.., 1024 x 8. This reconfigurable feature of the vector registers enables efficient execution of programs for various vector lengths.

The software automatically selects the optimum vector register configuration for each DO loop.

3.4 Effectiveness of the FACOM VP systems

The previous section discussed several architectural features in the FACOM VP systems. This section discusses the effect of this architecture on actual application programs.

Table 1 shows the results of our measurements. The numbers in each row are the execution times, measured in seconds, for three different levels of parallel processing. These three cases are described in Table 2. The numbers in parentheses are the performance ratios of the execution with full parallel processing to that without parallel processing, and the performance ratios of execution with full parallel processing to that with restricted parallel processing, respectively. All the measurements were made on the VP-200. The vector peak performance of the VP-200 is 570 MFLOPS, and the average scalar performance is 8 to 9 MFLOPS.

From Table 1, it is clear that the techniques used in the architecture have great effect. Case 3 is faster than Case 2 by the factor of 1.2 to 3.7. For details concerning the effect of each function, refer to Ref. [4]. Refs. [5] and [6] evaluate performance of other actual application programs.

Table 1: Effectiveness of various vector processing techniques

No	Program fields	Case 1 T1	Case 2 T2	Case 3 T3 (T1/T3)	Effect (T2/T3)	Primary effective architecture
1	Fluid model	855.56	68.72	35.93 (23.8)	(1.9)	IF
2	Fluid model	607.25	189.53	112.26 (5.4)	(1.7)	Vector register
3	High energy	209.48	33.83	26.83 (7.8)	(1.3)	IF, Vector register
4	High energy	31.22	4.12	2.41 (13.0)	(1.7)	IF
5	Particle in cell	168.04	37.69	32.32 (5.2)	(1.2)	IF, Indirect access
6	Plasma	101.79	70.40	42.93 (2.4)	(1.6)	Indirect access
7	Plasma	96.15	96.60	25.81 (3.7)	(3.7)	IF, Vector register
8	Weather forecast	413.80	192.65	109.52 (3.8)	(1.8)	IF
9	Nuclear	25.71	4.74	3.53 (7.3)	(1.3)	Macro operation
10	Energy transform	76.13	6.31	3.82 (19.9)	(1.7)	IF

Unit=seconds, ( )=ratio

Table 2: Description of three different levels of parallel processing

Case 1	Execution time without parallel processing
Case 2	Execution time when the following functions are restricted - Vectorization of DO loops containing complex control structures such as IF statements - Parallel execution of vector computational pipelines - Parallel execution of memory access-related pipelines - Dynamic register reconfiguration (fixed at 32 registers of 256 elements each)
Case 3	Execution time with full parallel processing

4. FUTURE OF HIGH-SPEED EXECUTION BY PARALLEL PROCESSING

As stated in Section 2, application programs inherently contain MIMD type parallelism as well as SIMD type parallelism.

The MIMD type parallelism is not easily detectable with the techniques for the current system software, because they are mainly based on the data dependence analysis of the programs at the local level. However, detection of the MIMD type parallelism is essential for higher speed execution. It is natural to enlarge the target range from the parallelism of the SIMD type to that of the MIMD type.

4.1 Requirements for MIMD-oriented high-speed execution

It is obvious that multiple processors are required for the execution of the programs with MIMD type parallelism. The problem is how to configure the multiple processors and how to allocate the data among them. The latter is especially related to programming languages. That is, data allocation differs greatly depending on the problems to be solved over all spheres of the scientific and engineering fields. For example, a shared-type memory accessible from multiple processors will be best for a program which involves frequent access to all array data. In contrast, small pieces of local memory will be most effective for a program containing many small procedures. There are also many programs like the structure analysis which combine both frequent access to the array data as well as many small procedures.

The optimal configuration and allocation is determined by the specific program. This means the software-oriented approach, again, is very essential.

From the software point of view, there is a problem of granularity. This depends on the ratio between the overhead time of procedure (task) distribution/synchronization caused by program fragmentation and the net computation time in the procedures to be executed in parallel. For a given application program, granularity and parallelism are strongly coupled: if parallelism increases, granularity decreases, and vice versa. There is an optimal point where the parallelism and granularity are balanced. This point varies according to the problem characteristics.

4.2 Multitasking with a multiprocessor system

We have experimented the multitasking in October, 1979, using the FACOM M-200 tightly-coupled 4-multiprocessor system, the highest performance general-purpose computer at that time. This was reported in newspapers, nation-wide. An outline of the experiment follows.

a) Problem

The problem is to solve the 2-dimensional Laplace equation as given in (2) and (3) by SOR (Successive Over Relaxation), after approximating (2) by the finite difference scheme (size: 320x320) as given in (4).

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0 \qquad (2)$$

Boundary condition:

$$U(x,y) = \sin(x) * \sinh(y) \qquad (3)$$

Finite difference scheme:

$$U_{i,j}=U_{i,j}+\omega\{(U_{i+1,j}+U_{i-1,j}+U_{i,j+1}+U_{i,j-1})*0.25-U_{i,j}\} \qquad (4)$$

To perform multi-tasking, an array U is divided into four rectangular domains as shown in Figure 1, and each domain is assigned to one of four CPUs for parallel execution. To assure the same execution order as would be executed by one CPU and one task, the tasks are synchronized whenever tnere is an operation on data at a boundary of the four domains.

b) Program for multitasking

Multitask programs are written in FORTRAN. For controlling tasks, the following basic subroutines are provided. Figure 2 outlines the program for multitasking.

CALL FORK	subtask generation
CALL JOIN	subtask termination
CALL QGET CALL QPUT	synchronization of operation on data at boundary (synchronization between subtasks)
CALL POST CALL WAIT	synchronization between task and subtask

To obtain larger granularity, the convergence check was made as follows:

Step one:

Check the convergence every 200 iterations until the value becomes 1.5 times as large as the convergence criterion.

Step two:

Check the convergence every 10 iterations after the first step is completed.

c) Results

In executing our multitask program, we found that convergence took place at the 2150th iteration. We obtained the following timing results:

System	Wall clock	ratio
M200-1CPU	7 min 44 sec	1
M200-4CPU	2 min 6 sec	3.68

4.3 Survey on multiprocessing

We believe that MIMD type parallel processing should be the next target; since it is a more generalized concept. We are conducting a survey of actual application programs from this point of view.

Various kinds of levels being considered as the target for MIMD type parallelization are:

- multiple DO's (DO loop slicing)
- procedure blocks in a subroutine
- subroutines

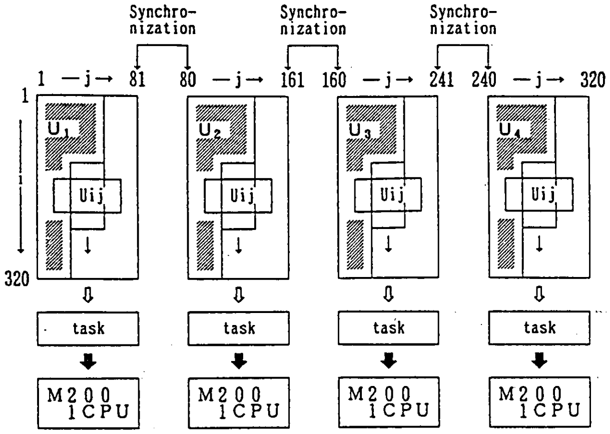


Figure 1: Dividing array U into 4 domains

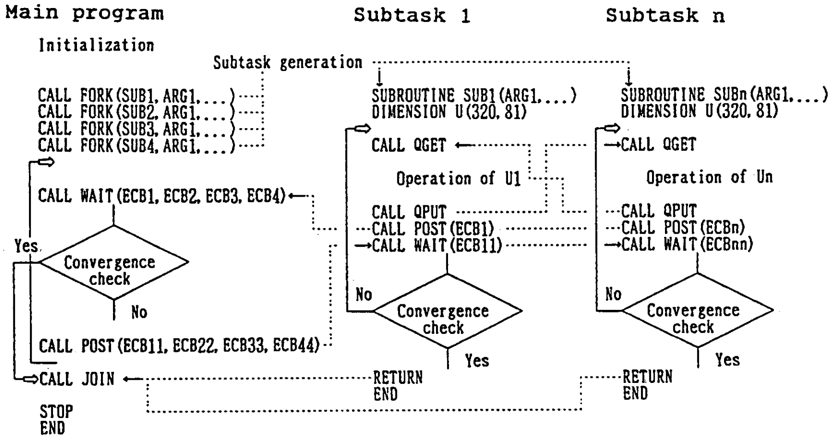


Figure 2: Outline of multitask program

The granularity increases in this order.

Table 3 shows part of the survey of two processor systems. The comments in the column "parallelization level" in Table 3 refer to the combined levels described above. Execution times are the simulated values by hand-parallelization.

4.4 Future of using parallel processing to get high performance

At present, the pipeline architecture for the SIMD type parallelism and the multiprocessor architecture for the MIMD type parallelism are the common combinations. Full utilization of these two types of parallelism is necessary to achieve the maximum performance for wide range of application programs. Any system that targets only one of the two types will soon reach its limit; the best system is a well-balanced one that utilizes both types of parallelism.

As for the ease of use, however, there already exists a vast amount of software in actual use that employs the automatic vectorization technique. Systems for the SIMD type have become very easy to use. Yet, there is little software that exploits the MIMD type parallelism. As indicated in the previous section, implementation of the MIMD type parallelism requires a great deal of program modification. In the previous example, a program with ten steps for a general-purpose computer was recoded into a program with several hundred steps for parallel processing and took three weeks just to run this program. This indicates the challenging nature of developing the parallelizing software. But, because use of the MIMD type parallelism is inevitable to obtain higher execution performance, more software will be developed that aims at applications in MIMD type parallelism.

5. CONCLUSION

The effectiveness of a software-oriented approach to achieve high performance in parallel processing has been discussed in this paper, using the Fujitsu's FACOM VP systems as an example. In addition, the importance of MIMD-type parallel processing as a future trend has been pointed out. For very high-speed execution, however, circuit and packaging technology, programming languages, and parallel processing algorithms are also important and will continue to be so.

"Ease of use" is also a very important factor in a high-performance system.

We will continue our systems development based on the primary goals of higher performance and ease of use.

Table 3: Effectiveness of parallel processing by two processors

Program field	Parallelization ratio	Execution time ratio	Parallelization level
Energy transform	99.9 %	2.0	routine + procedure
Weather forecast	99.9 %	1.9	multiple DO's
Plasma	92.3 %	1.8	multiple DO's + procedure
Plasma	88.9 %	1.7	multiple DO's + procedure
Plasma	46.4 %	1.3	multiple DO's + routine + procedure
Fluid model	33.9 %	1.2	routine + procedure
Nuclear	60.9 %	1.2	routine + procedure
Nuclear	1.0 %	1.0	multiple DO's + procedure

ACKNOWLEDGMENTS

The authors would like to express their thanks to Mr. S. Suzuki, Mr. K. Uchida, Dr. T. Matsuura, Mr. F. Isobe, and many other colleagues at Fujitsu Ltd. for valuable discussions, comments, and contributions on many occasions.

REFERENCES

[1] Kamiya,S., Isobe,F., Takashima,H. and Takiuchi,M.: Practical Vectorization Techniques for the 'FACOM VP', Information Processing, (Edited by Mason, R.F.A), IFIP, 1983, pp.389-394

[2] Miura,K., Fujitsu's Supercomputer: FACOM Vector Processor System, in Supercomputers, (Edited by S. Fernbach), North-Holland, 1986, pp. 137-151

[3] Tamura,H., Kamiya,S. and Ishigai,T., FACOM VP-100/200: Supercomputer with Ease of Use, Parallel Computing 2, North-Holland, 1985, pp.87-107

[4] Matsuura,T., Kamiya,S. and Takiuchi,M.: Design Concept of FACOM VP Based on Extensive Analysis of Applications, in Proc. IEEE Int. Conf. on Computer Design (ICCD '84) (October 1984), pp.232-237

[5] Matsuura,T., Miura,k. and Makino,M., Supervector Performance without Toil -FORTRAN Implemented Vector Algorithm on VP-100/VP-200, Computer Physics Communications 37, North-Holland, 1985, pp.101-107

[6] R.H.Mendez: Supercomputer Benchmarks Give Edge to Fujitsu, SIAM NEWS 3, 1984

"ADVANCED IMPLICIT SOLUTION FUNCTION OF DEQSOL AND ITS EVALUATION"

Chisato Kon'no, Miyuki Saji, Nobutoshi Sagawa, Yukio Umetani

Central Research Laboratory, Hitachi Ltd. Kokubunji, Tokyo 185, Japan

ABSTRACT

DEQSOL is a programming language specially designed to describe PDE problems in a quite natural way for numerical analyses. This language has two design targets. One is to enhance programming productivity by establishing a new architecture-independent language interface between numerical analysts and vector/parallel processors. The other is to automatically generate highly vectorizable FORTRAN codes from DEQSOL descriptions, thus realizing efficient execution. An advanced function has been introduced, so that the descriptive capability of DEQSOL for practical and complex problems has been highly elevated.

1. Introduction

Recently, demand for numerical routines which simulate physical phenomena has been rapidly increasing. The hardware used for such numerical simulation has made remarkable advances through innovations in LSI technology and computer architecture, such as vector and parallel processors. However, the programming itself still remains at a relatively elementary level.

The development of simulation programs by such conventional languages as FORTRAN faces the following problems. A long period of time is necessary to develop even a simple simulator, and special knowledge of numerical analysis methods like discretization is needed. In addition a specialized programming technique is required to exploit the performance of vector/parallel processors. Moreover, such programs are so lengthy and complicated that they are unfeasible to extend.

One approach for coping with these problems is the use of mathematical libraries or software packages designed for the special application fields. However, there are distinct limitations in applicable fields and the adopted numerical algorithms. Additionally, users can not control the numerical scheme in detail, and they frequently can not understand or utilize the complicated functions and interfaces.

To address these limitations, several simulation language systems has been developed, such as SALEM[1], PDEL[2], and ELLPACK[3,4]. ELLPACK especially seems to be a powerful system. ELLPACK has ample problem solving capabilities due to its extensive library, which is still expanding in response to the demands of the interactive and distributed system environment[4]. However, the new solver architecture which has been proposed for ELLPACK is not for enhancing its application to practical complex simulation problems.

On the other hand, the present study proposes a high level programming language system DEQSOL(Differential EQuation SOLver)[5,6] by which the numerical models can be described flexibly and which can automatically generate efficient simulation programs from the high level descriptions. This system has two main targets:

(1)To enhance programming productivity by establishing a new architecture-independent language interface between numerical analyst and computer.

(2)To generate highly vectorizable FORTRAN codes from DEQSOL

CH2345-7/86/0000/1026$01.00 © 1986 IEEE

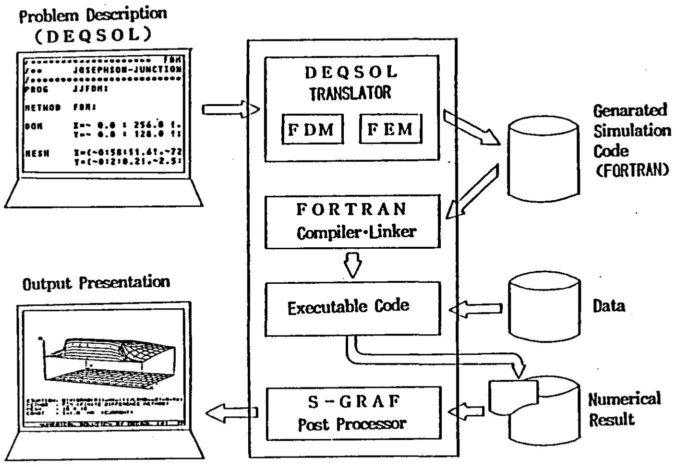
descriptions using its intrinsic parallelism.

The structure of the previously developed DEQSOL system and its processing flow are shown in Fig.1. The DEQSOL description is automatically translated into the FORTRAN simulation program by the DEQSOL translator. The results of several benchmark tests, which indicate that the programming productivity has been improved by almost an order of magnitude over FORTRAN, are shown in Table 1. The table also shows that most of the generated codes have extremely high vectorization ratios(91%-98%) on the Hitachi S-810 vector processor.

Both the Finite Difference Method(FDM) and the Finite Element Method(FEM) are available as discretization method. The key feature of DEQSOL is that users can describe various numerical schemes for a wide range of problems quite naturally by using fundamental DEQSOL statements.

One of the principal functions in DEQSOL is the implicit solution function. Here, this function acts to discretize the objective linear PDE into a system of linear equations. It then obtains a discrete PDE solution by solving this linear equation system. The restriction on the former DEQSOL implicit solution function has been that only one PDE in a rectangular region is permitted for a single implicit solution function. As a result, tightly coupled simultaneous PDEs, or PDEs defined in non-rectangular regions, have not been accepted. Because practical problems are sometimes formulated as described above, this restriction must be eliminated in order to make DEQSOL a practical solver.

In this paper, an advanced implicit solution function is presented for removing these restrictions. Consequently, the tightly coupled simultaneous PDEs defined in regions described by unions or by differences in rectangular subregions can be solved by DEQSOL. In addition, a block-iteration scheme aimed at reducing the region size needed to execute large-scale problems can be developed.



DEQSOL : Differential EQuation SOLver
S-GRAF : Scientific GRAphing Facilities
FDM : Finite Difference Method
FEM : Finite Element Method

Fig. 1 Processing Flow of DEQSOL Program

Tab. 1 Evaluation of DEQSOL (FDM/FEM)
(Productivity & Vectorization Ratio)

Program Name		CVD	PCAD	WEL	JJD	MHD	EBT
Discretization Method (Dim.)		FDM (3)	FDM (2)	FDM (2)	FEM (2)	FEM (3)	FEM (2)
Lines-of-code	DEQSOL	127	79	67	96	904	239
	Generated FORTRAN	1,361	1,312	1,091	1,078	11,558	2,476
	(Ratio)	(10.7)	(16.6)	(16.3)	(11.2)	(12.8)	(10.4)
Vectorization Ratio (S-810) [%]		91.0	96.4	94.1	92.7	93.9	93.7
Acceleration by Vector Processor(S-810)		3.8	8.2	9.4	5.2	7.7	5.5

CVD : Flow Analysis of Chemical Vapour Deposition
PCAD: Impurity Distribution Analysis in LSI Process CAD
WEL : Device Simulation of WEL Layer in LSI
JJD : Magnetic Field Analysis of Josephson Junction Device
MDH : Magnetic Field Analysis of Disc Head
EBT : Field Potential Analysis of Electron Beam Tube

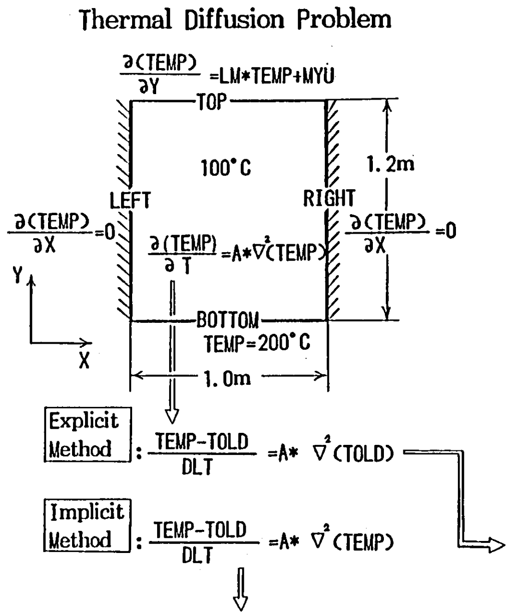
In the following sections, the implicit solution function is first outlined. Then, the method for processing the advanced function is presented. Finally, the results of the benchmark test are shown.

2. DEQSOL scheme description and the implicit solution function

A sample of the DEQSOL scheme description is shown briefly[6] for a simple diffusion problem in Fig.2. Statement (2) indicates the discretization method chosen by the user. Statements (3) to (10) indicate the structure of the numerical model respectively: domain(DOM), time domain(TDOM), FDM meshes(MESH), unknown physical variables(VAR), known physical constants(CONST), subregions(REGION), initial condition(INIT), and boundary conditions(BOUND).

Between 'SCHEME' statement(11) and 'END SCHEME' statement(17), a numerical algorithm is described. To develop a numerical algorithm for this problem, the time differentiation is replaced by the forward difference and the left term is estimated according to either an explicit or implicit method. For the explicit method, statement(14) is used, which assigns the evaluated value of the right terms to the left variable. Statements (12) to (17) represent a numerical algorithm by Euler's explicit method, where the statements between 'ITER'ation and 'END ITER'ation are executed until the specified condition is satisfied. For the implicit method, the 'SOLVE' statement is used, which solves the PDE for the indicated variable. If the implicit method is selected, such as the backward Euler's method, statement(14) should be replaced by 'SOLVE' statement(19). 'SOLVE' statement can be applied not only for the implicit method in transient problems, but also for any PDEs.

The method of translation for the 'SOLVE' statement is described in Fig.3. First, the region is broken down into several subregions according to the given PDE, boundary conditions, and discretization rule, such that each subregion is dominated by a unique equation and discretization rule. For example, the domain in the figure is broken into 6 subregions: 5 boundary and 1 inner point subregions. Each subregion is a unit where the same pattern of linear equations is formed by discretization onto every mesh point. As a result, the generated FORTRAN codes which calculate portions of the total matrix and constant vector can be combined into a single DO-loop.


Thermal Diffusion Problem
∂(TEMP)/∂Y =LM*TEMP+MYU
TOP
100°C
1.2m
LEFT
RIGHT
∂(TEMP)/∂X =0
∂(TEMP)/∂T =A*∇²(TEMP)
∂(TEMP)/∂X =0
Y
X
BOTTOM
TEMP=200°C
1.0m
Explicit Method : (TEMP-TOLD)/DLT =A* ∇²(TOLD)
Implicit Method : (TEMP-TOLD)/DLT =A* ∇²(TEMP)

```
(19) SOLVE  TEMP OF (TEMP-TOLD)/DLT=A*LAPL(TEMP)
                                   BY 'ILUCR';
```

```
(1)  PROG   THERM;
(2)  METHOD  FDM;
(3)  DOM    X= [0:1] ,  Y= [0:1.2] ;
(4)  TDOM   T= [0:1] ;
(5)  MESH   X= [0:1:0.2] ,
            Y= [0:1.2:0.2] ,
            T= [0:1:0.02] ;
(6)  VAR    TEMP, TOLD;
(7)  CONST  A=0.62, LM=0.8, MYU=50;
(8)  REGION RIGHT= [1,*] ,
            LEFT= [0,*] ,
            UP= [*,1.2] ,
            BOTTOM= [*,1] ,
            TOTAL= [*,*] ;
(9)  INIT   TEMP=100 AT TOTAL;
(10) BOUND  DX(TEMP)=0 AT RIGHT+LEFT,
            TEMP=200 AT BOTTOM,
            DY(TEMP)=LM*TEMP+MYU AT TOP
(11) SCHEME;
(12)    ITER  NT  UNTIL NT GT 50;
(13)      TOLD=TEMP;
(14)      TEMP=TOLD+DLT*A*LAPL(TOLD);
(15)      DISP TEMP EVERY 10 TIMES;
(16)    END ITER;
(17) END SCHEME;
(18) END;
```

Fig. 2 A Simple Example of Description by DEQSOL

For each subregion, the DEQSOL translator discretizes the equation according to the discretization rule. It then generates a part of the linear equations, and creates the FORTRAN codes which calculate the corresponding part of the total matrix and constant vector elements of the linear equation system. After the total matrix is completed, it is linked to the efficient matrix solution library which has been prepared beforehand. A portion of the generated FORTRAN codes consisting of the matrix generation and the linkage to the matrix solution library is shown in Fig.4.

The intrinsic parallelism in processing the 'SOLVE' statement is classified into two categories. The first category is the mesh point parallelism which appears when generating a matrix and constant vector from the discretized equations. The other is the algorithm which is dependent on parallelism to solve the linear equations. The first type parallelism is ensured by creating the codes so that the elements of the matrix and constant vector can be uniformly calculated within each subregion by using a single DO-loop, as stated above. The latter parallelism is accomplished by using the efficient matrix solvers in the DEQSOL library.

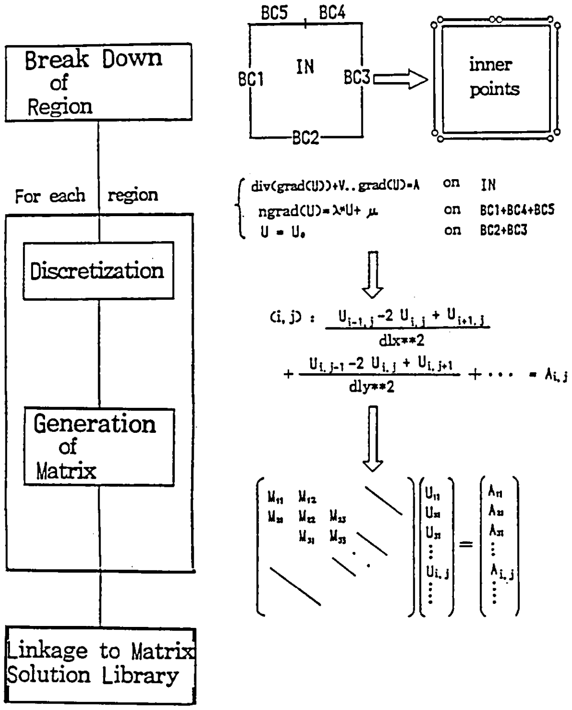
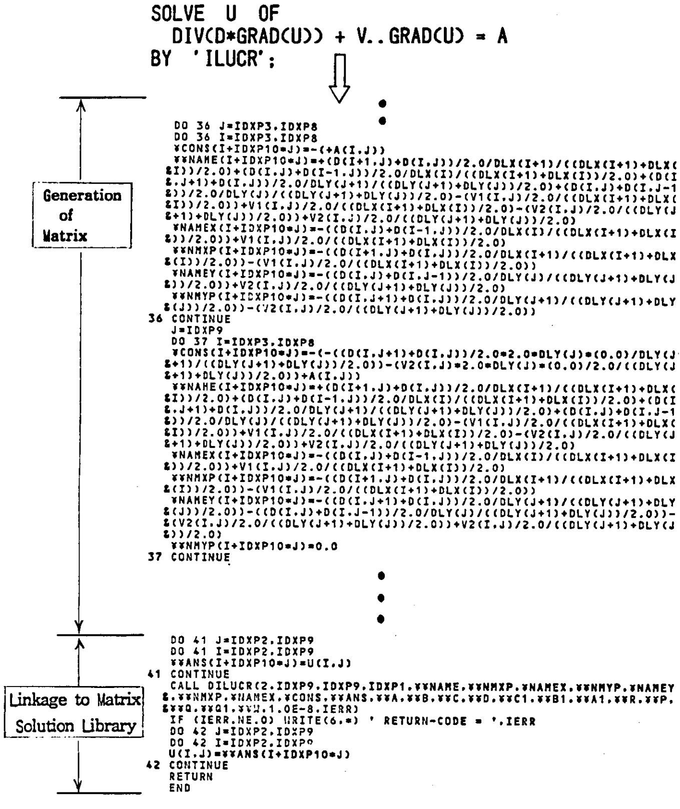

BC5
BC4
Break Down of Region
BC1
IN
BC3
BC2
inner points
For each region
div(grad(U))+V..grad(U)=A on IN
ngrad(U)=λ*U+ μ on BC1+BC4+BC5
U = U₀ on BC2+BC3
Discretization
(i, j) : (U i-1, j -2 U i, j + U i+1, j)/dlx**2 + (U i, j-1 -2 U i, j + U i, j+1)/dly**2 + ··· = A i, j
Generation of Matrix
M11 M12
M21 M22 M23
M31 M33
U11
U21
U31
Ui, j
A11
A21
A31
Ai, j
Linkage to Matrix Solution Library

Fig. 3 Automatic Discretization Method


Generation of Matrix
Linkage to Matrix Solution Library

```
SOLVE  U  OF
  DIV(D*GRAD(U)) + V..GRAD(U) = A
BY  'ILUCR';

      DO 36 J=IDXP3.IDXP8
      DO 36 I=IDXP3.IDXP8
      ¥CONS(I+IDXP10*J)=-(+A(I.J))
      ¥¥NAME(I+IDXP10*J)=+(D(I+1.J)+D(I.J))/2.0/DLX(I+1)/((DLX(I+1)+DLX(
     &I))/2.0)+(D(I.J)+D(I-1.J))/2.0/DLX(I)/((DLX(I+1)+DLX(I))/2.0)+(D(I
     &.J+1)+D(I.J))/2.0/DLY(J+1)/((DLY(J+1)+DLY(J))/2.0)+(D(I.J)+D(I.J-1
     &))/2.0/DLY(J)/((DLY(J+1)+DLY(J))/2.0)-(V1(I.J)/2.0/((DLX(I+1)+DLX(
     &I))/2.0))+V1(I.J)/2.0/((DLX(I+1)+DLX(I))/2.0)-(V2(I.J)/2.0/((DLY(J
     &+1)+DLY(J))/2.0))+V2(I.J)/2.0/((DLY(J+1)+DLY(J))/2.0)
      ¥NAMEX(I+IDXP10*J)=-((D(I.J)+D(I-1.J))/2.0/DLX(I)/((DLX(I+1)+DLX(I
     &))/2.0))+V1(I.J)/2.0/((DLX(I+1)+DLX(I))/2.0)
      ¥¥NMXP(I+IDXP10*J)=-((D(I+1.J)+D(I.J))/2.0/DLX(I+1)/((DLX(I+1)+DLX
     &(I))/2.0))-(V1(I.J)/2.0/((DLX(I+1)+DLX(I))/2.0))
      ¥NAMEY(I+IDXP10*J)=-((D(I.J)+D(I.J-1))/2.0/DLY(J)/((DLY(J+1)+DLY(J
     &))/2.0))+V2(I.J)/2.0/((DLY(J+1)+DLY(J))/2.0)
      ¥¥NMYP(I+IDXP10*J)=-((D(I.J+1)+D(I.J))/2.0/DLY(J+1)/((DLY(J+1)+DLY
     &(J))/2.0))-(V2(I.J)/2.0/((DLY(J+1)+DLY(J))/2.0))
   36 CONTINUE
      J=IDXP9
      DO 37 I=IDXP3.IDXP8
      ¥CONS(I+IDXP10*J)=-(-((D(I.J+1)+D(I.J))/2.0*2.0*DLY(J)*(0.0)/DLY(J
     &+1)/((DLY(J+1)+DLY(J))/2.0))-(V2(I.J)*2.0*DLY(J)*(0.0)/2.0/((DLY(J
     &+1)+DLY(J))/2.0))+A(I.J))
      ¥¥NAME(I+IDXP10*J)=+(D(I+1.J)+D(I.J))/2.0/DLX(I+1)/((DLX(I+1)+DLX(
     &I))/2.0)+(D(I.J)+D(I-1.J))/2.0/DLX(I)/((DLX(I+1)+DLX(I))/2.0)+(D(I
     &.J+1)+D(I.J))/2.0/DLY(J+1)/((DLY(J+1)+DLY(J))/2.0)+(D(I.J)+D(I.J-1
     &))/2.0/DLY(J)/((DLY(J+1)+DLY(J))/2.0)-(V1(I.J)/2.0/((DLX(I+1)+DLX(
     &I))/2.0))+V1(I.J)/2.0/((DLX(I+1)+DLX(I))/2.0)-(V2(I.J)/2.0/((DLY(J
     &+1)+DLY(J))/2.0))+V2(I.J)/2.0/((DLY(J+1)+DLY(J))/2.0)
      ¥NAMEX(I+IDXP10*J)=-((D(I.J)+D(I-1.J))/2.0/DLX(I)/((DLX(I+1)+DLX(I
     &))/2.0))+V1(I.J)/2.0/((DLX(I+1)+DLX(I))/2.0)
      ¥¥NMXP(I+IDXP10*J)=-((D(I+1.J)+D(I.J))/2.0/DLX(I+1)/((DLX(I+1)+DLX
     &(I))/2.0))-(V1(I.J)/2.0/((DLX(I+1)+DLX(I))/2.0))
      ¥NAMEY(I+IDXP10*J)=-((D(I.J+1)+D(I.J))/2.0/DLY(J+1)/((DLY(J+1)+DLY
     &(J))/2.0))-((D(I.J)+D(I.J-1))/2.0/DLY(J)/((DLY(J+1)+DLY(J))/2.0))-
     &(V2(I.J)/2.0/((DLY(J+1)+DLY(J))/2.0))+V2(I.J)/2.0/((DLY(J+1)+DLY(J
     &))/2.0)
      ¥¥NMYP(I+IDXP10*J)=0.0
   37 CONTINUE
                              •
                              •
                              •
      DO 41 J=IDXP2.IDXP9
      DO 41 I=IDXP2.IDXP9
      ¥¥ANS(I+IDXP10*J)=U(I.J)
   41 CONTINUE
      CALL DILUCR(2.IDXP9.IDXP9.IDXP1.¥¥NAME.¥¥NMXP.¥NAMEX.¥¥NMYP.¥NAMEY
     &.¥¥NMXP.¥NAMEX.¥CONS.¥¥ANS.¥¥A.¥¥B.¥¥C.¥¥D.¥¥C1.¥¥B1.¥¥A1.¥¥R.¥¥P.
     &¥¥Q.¥¥Q1.¥¥W.1.0E-8.IERR)
      IF (IERR.NE.0) WRITE(6.*) ' RETURN-CODE = '.IERR
      DO 42 J=IDXP2.IDXP9
      DO 42 I=IDXP2.IDXP9
      U(I.J)=¥¥ANS(I+IDXP10*J)
   42 CONTINUE
      RETURN
      END
```

Fig. 4 Generated FORTRAN Code

3. Advanced implicit solution function

3.1 Implicit solution function for simultaneous PDEs

The numerical scheme for simultaneous PDEs, F(A,B,C) = 0,G(A,B,C) = 0,H(A,B,C) = 0, using the former implicit function, is shown in Fig.5(a). Each equation is solved successively and iteratively until a convergent condition is met. If the coupling relationship among the variables is not tight, this scheme can successfully attain to an appropriate solution within a reasonable period of time. However, when the coupling relationship is tight and the dominating variable in each PDE is not clear, this iteration loop takes a lot of time for convergence, or does not converge at all. To solve such a problem, the implicit solution function can be extended such that any coupled variables and

PDEs are specifiable in a simple 'SOLVE' statement with appropriate boundary conditions specified using the 'UNDER' clause. This is shown in Fig.5(b). By using this clause, the coupled boundary conditions can definitely be assigned to each 'SOLVE' statement.

The extended, advanced implicit solution function discretizes the coupled PDEs, regards them as one system of linear equations for coupled variables, and generates a large matrix for them. In this case, the form of the generated matrix depends on the order in which the linear equations and discrete variables are arranged. For example, suppose F,G,H are PDEs and A,B,C are variables of these PDEs. In addition, Fi,Gi,Hi and Ai,Bi,Ci are discretized equations and variables at point i. The forms of the matrices generated for typical arrangements of Fi,Gi,Hi,Ai,Bi, and Ci are shown in Tab.2. In this table, the solid lines in the matrix indicate the position where non-zero elements may appear. From the viewpoint of solving the linear equations, a band matrix is desirable for the direct method, such as GAUSS elimination. In addition, the dominance of diagonal elements is necessary for iterative methods, such as Conjugate Gradient(CG) or Bi-Conjugate Gradient(BCG), to converge. Therefore, the DEQSOL translator generates the matrix in the form shown by case.1 of Table 2. Furthermore, the generated matrix is manipulated into the compressed form which retains only non-zero elements.

The BCG algorithm for the advanced DEQSOL implicit solution function is described as follows. If the small block matrix consisting of elements for each discrete point is regarded as an element of the matrix in Fig.6, the matrix form coincides with that of the conventional FDM matrix. As a result, the BCG algorithm for this block matrix can be constructed in similar way to conventional ones if the operation between the elements is carefully defined. Here, the product corresponds to the matrix product, and the quotient corresponds to the product of the inverse matrix, where these operations are non-commutative.

```
SCHEME;
  AOLD=AO; /*INITIAL SETTING*/
  BOLD=BO; /*INITIAL SETTING*/
  COLD=CO; /*INITIAL SETTING*/
  ITER N1  UNTIL  NORM  LT  eps;
    SOLVE  A  OF   F(A,B,C) = 0
      BY  'ILUBCG';
    SOLVE  B  OF   G(A,B,C) = 0
      BY  'ILUBCG';
    SOLVE  C  OF   H(A,B,C) = 0
      BY  'ILUBCG';
    CALL NORM2(A,B,C,AOLD,BOLD,COLD,NORM);
    AOLD=A;
    BOLD=B;
    COLD=C;
  END  ITER;
END  SCHEME;
```

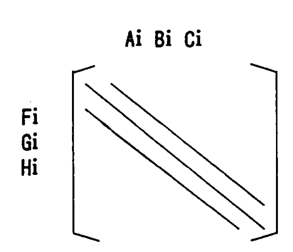
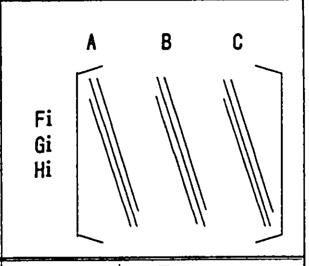
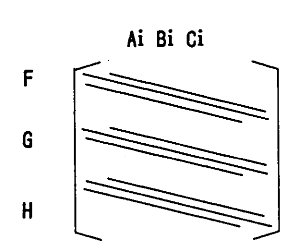
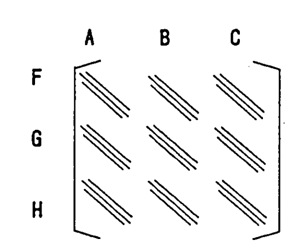
(a). Iteration scheme using the former function

```
SCHEME;
  SOLVE  A,B,C  OF
    F(A,B,C) = 0
   ,G(A,B,C) = 0
   ,H(A,B,C) = 0
  UNDER BCS1
  BY  'ILUBCG';
END  SCHEME;
```

(b). Scheme using the new function

Fig. 5 Numerical Schemes for Simultaneous PDEs

Tab. 2 Forms of the Generated Matrix

CASE．1	CASE．2
({Ai, Bi, Ci})	({Ai} {Bi} {Ci})
({Fi, Gi, Hi})	({Fi, Gi, Hi})
	
CASE．3	CASE．4
({Ai, Bi, Ci})	({Ai} {Bi} {Ci})
({Fi} {Gi} {Hi})	({Fi} {Gi} {Hi})
	

(note). ({Ai} {Bi} {Ci})=(A_1, A_2··B_1, B_2··C_1, C_2·)
({Ai, Bi, Ci})=(A_1, B_1, C_1, A_2, B_2, C_2···)

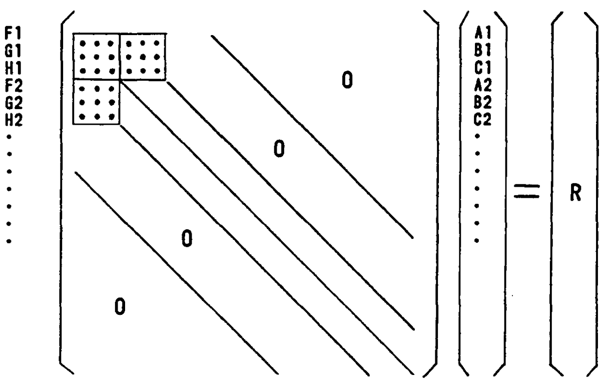


Fig. 6 Matrix Form Representing Discretized Simultaneous PDEs

The DEQSOL translator calculates the necessary matrix size. This size depends on the number of coupled systems involved for each 'SOLVE' statement in the given numerical algorithm. Only the maximum region is secured and it is shared alternatively among all 'SOLVE' statements.

3.2 Implicit solution function for partial regions

It is desirable that the implicit solution function can be applied to not only rectangular region, but also to the unions or differences of rectangular regions. Furthermore, this function should be applied to partial regions for processing problems which have inner boundaries. It is also beneficial for block iteration schemes to be developed for large-scale problems in order to reduce the region size needed for computations. To accommodate these requirements, the implicit solution function is expanded so that it can be applied to any region or regions which can be expressed by unions or differences of sub-rectangular regions.

An extended form of the statements 'SOLVE' and 'BOUND' is shown in Fig.7. The 'AT' specification is introduced into the 'SOLVE' statement to express such partial regions. For the boundary conditions, four types are allowed: 'equation', 'discrete equation', 'GIVEN' and 'NOC'. Here 'equation' specifies the ordinary conditions(first,second,and third types of boundary condition). Next, 'discrete equation' specifies the discretized form, and 'GIVEN' specifies that the point on the boundary of the partial region takes the value which was already calculated as the first type of boundary condition. Finally 'NOC' specifies that the PDE given on the inner point is also adopted and discretized on the boundary point. On this boundary point, the referred points outside of the designated partial region take previously calculated values.

The specification of 'GIVEN' and 'NOC' mainly aim at constituting the block iteration scheme for large-scaled or multi-block problems. On the other hand, 'equation' and 'discretized equation' aim at setting ordinary conditions and also at developing schemes for inner boundary problems.

The translation method for this function is basically similar to that explained in Section 2.

```
BOUND  [<BC-list> , ···  ]
       [BCSm             ]
       [<BC-list> , ···  ]
          :          ;

SOLVE  V1, V2, ··, Vn  OF        : Objective variables
  F1(V1, V2, ··)=W1              : Objective PDEs
 , F2(V1, V2, ··)=W2

 , Fn(V1, V2, ··)=Wn
  UNDER  BCSm                    : Boundary conditions
  AT  REG1+REG2+·····            : Objective regions
  BY  'LIBRARY'  WITH ·· ;       : Matrix solution library
```

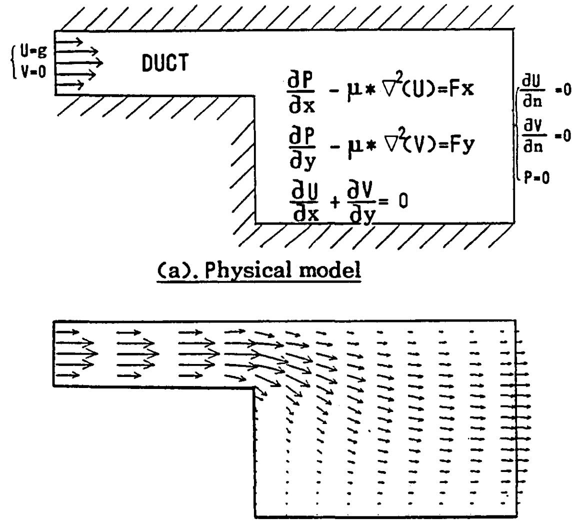
```
(note1).
 <BC-list>::= equation AT sub-reg +···
               discrete equation AT sub-reg +···
               NOC AT sub-reg +···
               GIVEN AT sub-reg +···
(note2). Example of
 ·equation : U=U0,
             A*DX(U)+B*DY(U)+··=0,
 ·discrete
  equation : (-U(I+2)+4*U(I+1)-3*U(I))/(2*DLX(I))
```

Fig. 7 Statement Form of Advanced 'SOLVE'

4. Evaluation

As the physical models increase in complexity, an advanced implicit solution function becomes necessary for simultaneous PDEs. Such a model can be found in the fields of fluid analysis, LSI device simulation, etc.. The results of a benchmark test for the analysis of Stokes flow in a non-rectangular region are now given. Stokes flow in the 2-dimensional domain is expressed using three equations for pressure P, velocity of flow U and V, and external forces Fx and Fy for each coordinate direction. This benchmark test utilizes both facilities as stated in the above section.

The physical model is shown in Fig.8(a). An iteration scheme for this model in which each equation is solved for one specific variable can not obtain a convergent solution. Two numerical schemes which utilize the advanced implicit solution function for simultaneous equations are shown in Fig.9. Fig.9(a) represents a scheme in which block iteration is applied. The simultaneous PDEs are iteratively solved by the implicit solution function in each partial region REG1 and REG2, until the norm of the difference of the new and old values becomes less than the specified value "eps". In Fig.9(b), the simultaneous PDEs are solved in the total region REG1 + REG2. The generated FORTRAN program from scheme (a) uses far less memory during execution than that from (b), but setting of the initial value for the iteration in (a) is quite critical. From either of these schemes, however, a good approximate solution can be obtained.


U=g
V=0
DUCT
$\frac{\partial P}{\partial x} - \mu * \nabla^2(U) = Fx$
$\frac{\partial P}{\partial y} - \mu * \nabla^2(V) = Fy$
$\frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} = 0$
$\frac{\partial u}{\partial n} = 0$
$\frac{\partial v}{\partial n} = 0$
P=0

(a). Physical model

(b). Numerical Result (vector of velocity)

Fig. 8 Problem for Benchmark Test

Tab. 3 Evaluation of the Advanced Function

Program Name		STOKES
Discretization Method (Dim.)		FDM (2)
Lines-of-code	DEQSOL	43
	Generated FORTRAN	866
	(Ratio)	(20.1)
Vectorization Ratio (S-810) [%]		91.5
Translation CPU Time (M-200H) [sec]		29.4

STOKES: Analysis of Stokes Flow in the Duct

```
BOUND BCS1
      U=g  AT  L1.
      V=0  AT  L1.
      U=NOC  AT  L2.
      V=NOC  AT  L2.
      .
      BCS2
      U=NOC  AT  L2.
      V=NOC  AT  L2.
      DX(U)=0  AT  L3.
      DX(V)=0· AT  L3.
      P=0  AT L3.
      .
      .     ;
SCHEME;
  UOLD=UO; /*INITIAL SETTING*/
  VOLD=VO; /*INITIAL SETTING*/
  POLD=PO; /*INITIAL SETTING*/
  ITER  N1  UNTIL  NORM  LT  eps;
    SOLVE  U.V.P  OF
      DX(P) - MYU*LAPL(U) = FX.
      DY(P) - MYU*LAPL(V) = FY.
      DX(U) + DY(V) = 0
      UNDER  BCS1
      AT  REG1
      BY  'ILUBCG';
    SOLVE  U.V.P  OF
      DX(P) - MYU*LAPL(U) = FX.
      DY(P) - MYU*LAPL(V) = FY.
      DX(U) + DY(V) = 0
      UNDER  BCS2
      AT  REG2
      BY  'ILUBCG';
    CALL NORM2(U.V.P.UOLD.VOLD.POLD.NORM);
  END  ITER;
END  SCHEME;
```

(a) Scheme Using Block Iteration

```
BOUND BCS3
      U=g  AT  L1.
      V=0  AT  L1.
      .
      DX(U)=0  AT  L3.
      DX(V)=0  AT  L3.
      P=0  AT L3.
      .
      .     ;
SCHEME;
   SOLVE  U.V.P  OF
    DX(P) - MYU*LAPL(U) = FX.
    DY(P) - MYU*LAPL(V) = FY.
    DX(U) + DY(V) = 0
    UNDER  BCS3
    AT  REG1 + REG2
    BY  'ILUBCG';
END  SCHEME;
```

(b) Scheme Applying Advanced Implicit Solution Function

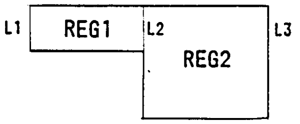


Fig. 9 Numerical Scheme for Stokes Flow

The flow velocity for the calculated results of the program generated from scheme (a) in Fig.9 is shown in Fig.8(b). Furthermore, the results of the DEQSOL translation and the execution of the generated FORTRAN program are shown in Table.3. The descriptive efficiency of DEQSOL still attains more than 10 times that of FORTRAN, and the vectorization ratio of the generated FORTRAN has marked over 90 %.

5. Conclusion

The advanced implicit solution functions in FDM schemes concerning simultaneous PDEs and partial regions have been developed. Due to these functions, the fields of application have been extended and the descriptive capability of the numerical algorithms have been greatly improved. The problems formulated by tightly coupled PDEs - those involving inner boundaries and those defined in non-rectangular regions - have become solvable by DEQSOL. Aside from these extensions in capability, DEQSOL has maintained a high descriptive efficiency versus FORTRAN, and the created FORTRAN program has demonstrated an extremely high vectorization ratio.

6. Acknowledgements

The authors would like to express their appreciation to Dr. Sakae Takahasi and Dr. Toru Toyabe of Hitachi Ltd. for their helpful technical discussions. Thanks are also due to Dr. Tsuneyo Chiba, Dr. Hisasi Horikosi and Dr. Yasutsugu Takeda of Hitachi Ltd. for their continuous encouragement.

< References >

[1] S.M.Morris and W.E.Sciesser : "SALEM - A Programming System for the Simulation of Systems Described by Partial Differential Equations", Proc. Fall Joint Computer Conference, Vol. 33, 1968, pp. 353-357

[2] A.F.Cardenas and W.J.Karplus : "PDEL - A Language for Partial Differential Equations", Com.ACM, March, 1970, pp.184-191

[3] J.R.Rice and R.F.Boisvert : "Solving Elliptic Problem Using ELLPACK", CSD-TR414, Computer Science Department, Purdue University, September, 1982(Revised May, 1983)

[4] J.R.Rice : "ELLPACK: An Evolving Problems Solving Environment" Proc. of IFIP WG 2.5 Working Conference on Problem Solving Environments for Scientific Computing, 1985, to be published

[5] Y.Umetani et al : "Numerical Simulation Language DEQSOL (Japanese)", Transaction of Information Processing Society of Japan, 26, 1985, pp.168-180

[6] Y.Umetani et al : "DEQSOL - A Numerical Simulation Language for Vector/Parallel Processors", Proc. of IFIP WG 2.5 Working Conference on Problem Solving Environments for Scientific Computing, 1985, to be published

FORTRAN AND TUNING UTILITIES AIMING AT EASE OF USE OF A SUPERCOMPUTER

Hiroshi Katayama - Makoto Tsukagoshi

Basic Software Development Division
NEC Corporation
10, 1-chome, Nisshin-cho, Fuchu-city
Tokyo 183, Japan

ABSTRACT

The supercomputer SX system is an ultra high-speed, large-scale computer system with a large capacity of main memory and extended memory. One of its main design objectives was to offer an easy-to-use environment to users as well as high-speed vector and scalar computation capabilities. This paper outlines an optimizing FORTRAN compiler with automatic vectorization functions and some tuning utilities to enhance vectorization ratio, which we developed to attain these goals.

INTRODUCTION

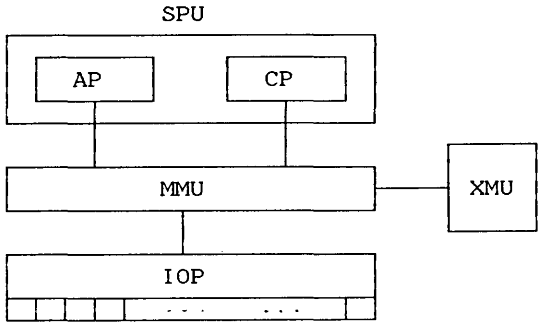
Early supercomputers were mainly used by professional people in big laboratories or companies. Though the main programming language was FORTRAN, it was necessary to use special vector extension added to FORTRAN or an assembly language to make full use of their capabilities.

Today, with the advance in computer technology, supercomputers have come to be able to provide far more computational power at lower cost. They are being brought into common use,for example, in universities and in industry. Non-professional people have begun to use them. Under these circumstances, ease of use has become very important, as well as computational speed. To meet this requirement, recent supercomputers provide a standard FORTRAN77 compiler with a sophisticated automatic vectorization function. FORTRAN77/VP of Fujitsu VP series, FORT77/HAP of Hitachi S810 series and FORTRAN77/SX of our SX system are examples. Cray Research enhanced an automatic vectorization function of their CFT compiler. Furthermore, they also provide some compiler functions or utilities for program tuning to get more performance easily. CFT's flow-trace function, VP's Interactive Vectorizer, S810's VECTIZER, SX's ANALYZER/SX and VECTORIZER/SX are examples.

In designing our supercomputer SX system, the first design objective was, of course, to realize high-speed vector and scalar computation capabilities. We attained this goal by the following hardware features:

- fast machine cycle (6 nanoseconds)
- multiple parallel vector pipelines consisting of four identical sets of four arithmetic pipelines
- a fast main memory of up to 256 million bytes and a semiconductor extended memory unit of up to 2 billion bytes
- the control processor (CP) integrated into the system to perform supervisory functions and cooperating with the arithmetic processor (AP) which takes charge of high-speed scientific computations.

With these features, the SX system has a peak performance of 1.3 gigaflops in vector operations. In scalar operations, it is about 1.5 times as fast as our current fastest mainframe computer ACOS S1500. The configuration of the SX system is shown in Fig.1.



SPU : SCIENTIFIC PROCESSING UNIT
AP : ARITHMETIC PROCESSOR
CP : CONTROL PROCESSOR
MMU : MAIN MEMORY UNIT
IOP : INPUT OUTPUT PROCESSOR
XMU : EXTENDED MEMORY UNIT

Fig.1 SX System Configuration

Considering the above requirements, our next design objective was to offer an easy-to-use environment both in programming and in operation. To meet this goal, we developed the following software products.

- SX system control program (SXCP) which was developed on the basis of the ACOS mainframe operating system and has abundant functions including a TSS function
- FORTRAN77/SX with automatic vectorization functions and optimization functions
- various utilities including ANALYZER/SX and VECTORIZER/SX to assist program preparation, debugging, tuning and maintenance.

All these programs run on the CP using its own memory. Thus programs running on the AP are not affected by them.

CH2345-7/86/0000/1034$01.00 © 1986 IEEE

In this paper, we describe the outlines of FORTRAN77/SX: mainly its vectorization and optimization functions, and also the outlines of tuning utilities to improve vectorization ratio.

FORTRAN77/SX

In the SX system, there are two FORTRAN systems: FORTRAN77 and FORTRAN77/SX. FORTRAN77 generates object codes for the CP and is used for debugging with the interactive debugging support program IDSP. FORTRAN77/SX generates vectorized object codes for the AP and is used for actual computations. Their language specifications conform to the standard FORTRAN, that is, the ANSI FORTRAN 77. As a result, users can make full use of the SX system through standard FORTRAN programs.

Automatic Vectorization Functions

FORTRAN77/SX compiler has sophisticated automatic vectorization functions. The outlines are shown in Table 1. Extended automatic vectorization functions here are the features to vectorize loops wholly or partially which are very hard or almost impossible to vectorize as they are. To increase vectorization effect by loop transformation is also one of its features. Some examples of vectorizable loops are shown in Fig.2.

```
    DO 10 I=1,N                         Vectorize using masked
      IF(A(I).GE.0.0)THEN      ==>      vector operations and
        X(I)=SIN(B(I)+C(I))             compress/expand oper-
      END IF                            ations for SIN
 10 CONTINUE
(1) A loop including an IF statement

    DO 10 I=1,N                                   Vectorize using
      A(IX(I))=B(IX(I))+T*C(IX(I))    ==>         gather/scatter
 10 CONTINUE                                      operations
(2) A loop including indirectly addressed vectors

    DO 10 I=1,N
      IF(A(I).GT.AMAX)THEN                   Vectorize using
        AMAX=A(I)                  ==>       vector operations
        MXIDX=I                              for MAX/MIN
      END IF
 10 CONTINUE
(3) A loop finding the maximum value and its index

    DO 10 I=1,N                              Vectorize using a
      IF(A(I).EQ.X) GOTO 20        ==>       vector operation
 10 CONTINUE                                 to find the 1st on
 20 CONTINUE                                 bit of the mask
(4) A loop searching for the element which has the specific
    value

    DO 10 I=1,N                            Vectorize by inserting
      A(I)=R(I)+S(I)              ==>      W(I)=A(I+1) before the
      S(I)=X(I)+Y(I)                       A(I)=R(I)+S(I) and sub-
 10   X(I)=A(I+1)+B(I)                     stituting A(I+1) by W(I)
(5) A loop including unsuitable array references

    DO 10 I=1,N
      A(I)=B(I)+C(I)                          Vectorize the outer
      DO 20 J=1,N                  ==>        loop as well as the
        X(I,J)=A(I)*Y(I,J)+Z(I,J)             innermost loop
 20   CONTINUE
 10 CONTINUE
(6) Nested DO Loops
```

Fig.2 Examples of Vectorizable Loops

The compiler analyzes global data flows of all variables and arrays in a program unit and uses this information in automatic vectorization. For example, in the case that a variable is defined in a loop, it does not generate codes to ensure the final value for the variable, if it is not alive on the exit of the loop.

Table 1 Summary of FORTRAN77/SX Automatic Vectorization

(1) Basic Conditions for Automatic Vectorization

Vectorizable Loop	DO loop
Types of Vectorizable Data	Integer, Logical Real (Single, Double) Complex (Single, Double)

Vectorizable Statements	Assignment, CONTINUE
	IF(Arithmetic, Logical, Block)
	ELSE IF, ELSE, END IF
	Unconditional GO TO

Vectorizable Operations	Add, Subtract, Multiply, Divide Power, Type Conversion Intrinsic Functions Inner Product, Summation Continued Product, Iteration Maximum, Minimum Search
Vectorizable Vector	Contiguous Vector Equal Distance Vector Indirectly Addressed Vector Index Variable

(2) Extended Automatic Vectorization Functions
(a) Extension of Vectorizable Loops

A loop including unvectorizable parts	Vectorize by splitting it into vectorizable parts and unvectorizable parts
Nested loops	Vectorize the outer loop by dividing it before and after the innermost loop
A loop including user function calls	Vectorize by putting a system subroutine between an object code and the functions to invoke them as many times as the number of loop iterations
A loop including array references that have dependencies unsuitable for vectorization	Vectorize by exchanging statements or substituting the references by a work vector
A loop that the compiler cannot decide whether it may be vectorized or not	Vectorize under specifications of a vectorization directive

(b) Increase of Vectorization Effect

Tightly nested loops	Exchange an outer loop and the innermost one or make some of them into a single loop to expand the vector length or to avoid access to the same memory bank
A loop including a special processing for boundary conditions	Eliminate IF's by moving them out of the loop

With the vectorization functions, together with the multiple parallel vector pipelines, the compiler generates efficient object codes for the DO loops ranging from simple loops to complicated loops. We show the results of the Livermore 14 kernels in Table 2 as an example.

Table 2 The Performance of the SX-2 Measured by the Livermore 14 Kernels

Loop Number	MFLOPS
1	757.5
2	423.5
3	559.3
4	122.9
5	14.6
6	15.6
7	810.1
8	156.6
9	724.6
10	163.8
11	24.4
12	255.2
13	8.3
14	25.2
Average	290.1

Optimization Functions

In addition to the automatic vectorization functions, FORTRAN77/SX compiler uses all sorts of optimization techniques to obtain full performance from the SX system. Some of the conventional ones such as common subexpression elimination, code motion and strength reduction were extended to apply to vectorized codes. A register assignment optimization was also extended to make full use of 128 scalar registers and 80 K byte vector registers. Besides them, an instruction scheduling optimization was newly introduced to minimize pipeline interlocks and functional unit interlocks.

In an instruction scheduling, the compiler makes reordering at three levels: a source expression level, an intermediate code level and an object code level. This is because more than one instruction is generated from an intermediate code, though the first two hold more information about a source program.

In determining which instruction is to be scheduled next, the compiler uses the following information:

- the number of instructions waiting for their completion
- the duration of each instruction
- the busy time of each vector functional unit.

A large number of scalar registers ease this optimization and increase its effect. This optimization has more effect on scalar codes than on vectorized codes and can reduce the execution time by about 20%-40% for an average program. Table 3 shows the effect of an instruction scheduling for the Livermore 14 kernels.

Let us show a small example of reordering. The SX system has four sets of independent vector pipelines: add, multiply, logical and shift, which can work concurrently. Suppose a vectorized part contains an expression, A(I)*B(I)+C(I)*D(I)+E(I). The compiler evaluates it in the following order to fully utilize vector pipelines:

(A(I)*B(I)+E(I))+C(I)*D(I)

Let us show another small example of optimization. Suppose a loop includes a vector addition in the form of A(I)+A(I). The compiler generates a vector addition, vector multiplication or vector double instruction for it depending on which pipeline is free. Note that a vector double instruction is executed by a shift pipeline.

Table 3 The Effect of the Instruction Scheduling Measured by the Scalar Codes for the Livermore 14 Kernels

Loop No.	Not Scheduled (MFLOPS)	Partially Scheduled (MFLOPS)	Fully Scheduled (MFLOPS)
1	17.3	18.4(6.4%)	18.1(4.6%)
2	17.9	26.8(49.7%)	31.4(75.4%)
3	13.3	13.9(4.5%)	15.9(19.5%)
4	11.5	11.5(0.0%)	11.5(0.0%)
5	12.0	16.4(36.7%)	15.9(32.5%)
6	12.0	14.9(24.2%)	16.7(39.2%)
7	20.0	33.8(69.0%)	32.2(61.0%)
8	24.7	43.9(77.7%)	46.2(87.0%)
9	20.5	29.3(42.9%)	32.0(56.1%)
10	13.7	13.6(-0.7%)	18.5(35.0%)
11	8.3	8.2(-1.2%)	8.3(0.0%)
12	8.3	8.3(0.0%)	8.3(0.0%)
13	6.3	7.2(14.3%)	9.3(47.6%)
14	11.6	12.0(3.4%)	14.3(23.3%)
Average	14.1	18.4(30.5%)	19.9(41.1%)

Note:(1) Parenthesized figures show the effects of the instruction scheduling.
(2) "Partially scheduled" means reordering at an object code level is not performed.
(3) "Fully scheduled" means full reordering is performed.

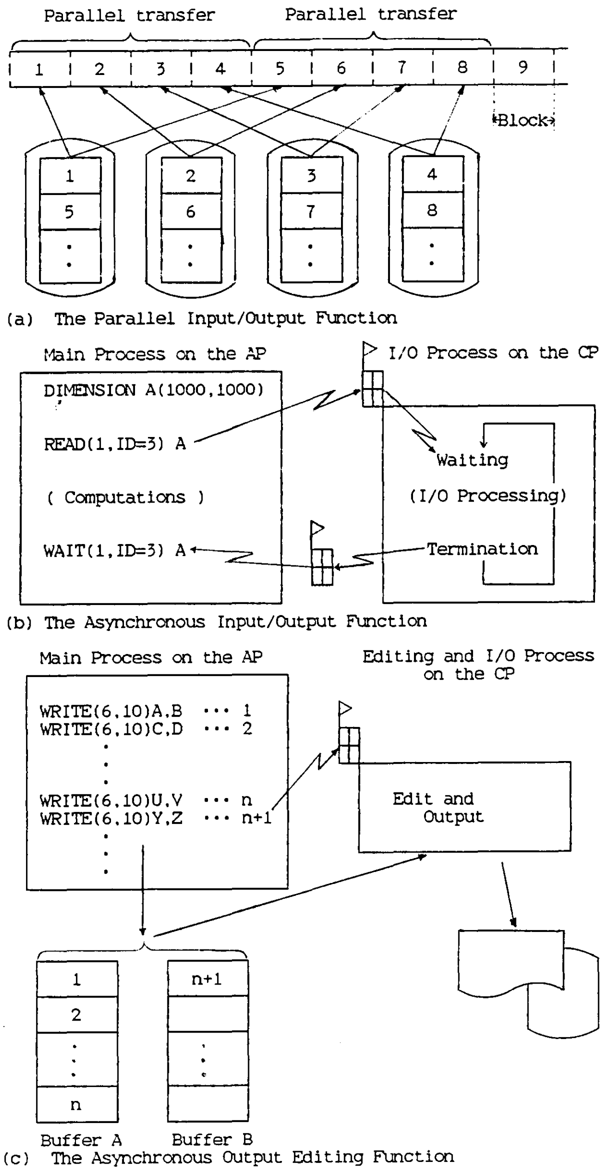
High-speed Input/output Functions

As the speed of a computer becomes faster, the size of an application program for it becomes larger. Though the main memory of the SX system has a capacity of up to 256 million bytes, it sometimes cannot accommodate its data. In this case, the disk I/O time may become a system bottleneck. To avoid this, FORTRAN77/SX provides high-speed input/output functions such as:

- Asynchronous input/output function which allows unformatted input/output operation to proceed on the CP in parallel with program execution on the AP
- Parallel input/output function which allows parallel data transfer between a file and a main memory where the file is subdivided and distributed among multiple disk units
- Asynchronous output editing function which allows editing and output processing to proceed on the CP in parallel with program execution on the AP.

The asynchronous input/output function makes full use of a distributed configuration of the SX system and is useful to reduce the turnaround time. However, the program must be modified to use it. The parallel input/output function, on the contrary, can be used simply by changing the JCL specification. No program modification is necessary. The asynchronous output editing function is very similar to the asynchronous input/output function, but it is to speed up a formatted output statement, whereas the latter is to speed up an unformatted input/output statement that transfers a large amount of data. Fig.3 shows these functions.

Besides them, the extended memory unit (XMU) can be used as a high-speed secondary storage. Users can use it through FORTRAN READ or WRITE statements simply by assigning a file to the XMU in a JCL. It achieves a drastic reduction of data transfer time.



(a) The Parallel Input/Output Function

(b) The Asynchronous Input/Output Function

(c) The Asynchronous Output Editing Function

Fig.3 High-Speed Input/Output Functions

High-speed Libraries

FORTRAN77/SX provides high-speed intrinsic functions both in scalar and vector. Some performance data is shown in Table 4. Besides them, the Advanced Scientific Library SX (ASL/SX) is provided including an FFT subroutine using a vector bit reversal instruction, a simultaneous linear equation solving subroutine and an eigen value calculating subroutine. For example, a one-dimensional complex FFT of data length 1024 can be solved in 133 microseconds (462 MFLOPS). Two-dimensional complex FFT of data length 256 x 256 can be solved in 9.41 milliseconds (557 MFLOPS). Simultaneous linear equations of order 1000 can be solved in 429 milliseconds (792 MFLOPS).

Table 4 The Performance of Some Intrinsic Functions

Function	Scalar(S) (microsec.)	Vector(V) (microsec.)	Ratio(S/V)
SQRT DSQRT	0.79 1.09	0.05 0.05	16 22
EXP DEXP	0.76 1.44	0.03 0.04	25 36
ALOG DLOG	1.10 1.30	0.04 0.06	28 22
SIN DSIN	0.89 1.37	0.03 0.03	30 46

Linkage With Other Languages

In the SX system, FORTRAN is the only language available for the AP. Because the CP has the same architecture with the ACOS mainframe computers, all the language processors for them are available on the CP. In addition, subroutines on the CP and FORTRAN subroutines on the AP can intermix freely. For example, a PASCAL subroutine can call a FORTRAN subroutine with the normal CALL statement whether it is for the CP or the AP. The linker, which makes up a load module, judges whether each object module is for the CP or the AP, and set the last bit of the pointer to the callee's entry point to 1 if a CP subroutine calls the AP subroutine or vice versa. When a CP subroutine calls an AP subroutine, the CP firmware knows it by the last bit of the pointer and gives control to the AP subroutine. When an AP subroutine calls a CP subroutine, a CP-call exception is caused by the bit and the exception handler gives control to the CP subroutine.

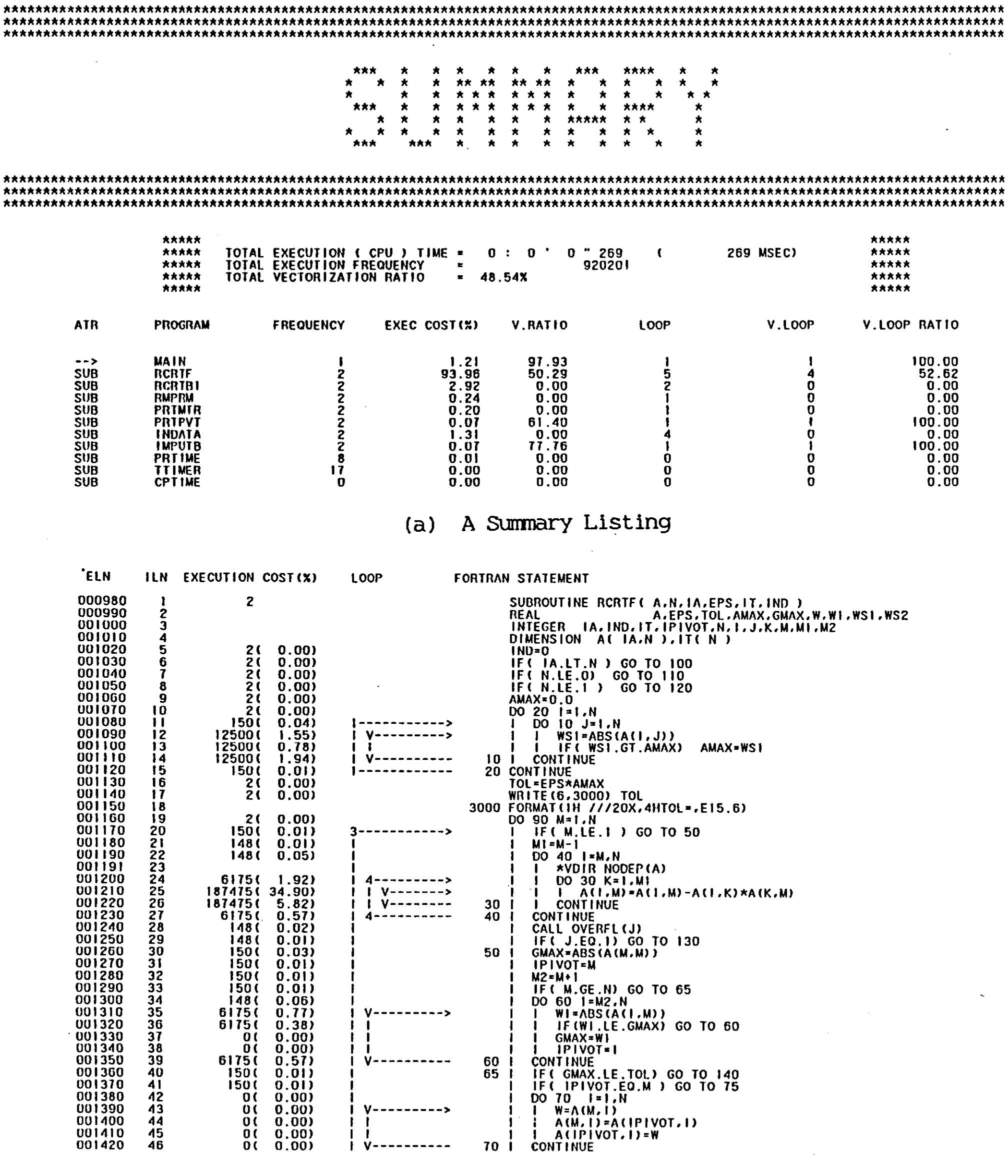
TUNING UTILITIES

The key factor to obtain a high performance from the SX system is to increase the vectorizable part of a program as much as possible. Another important factor is to make generated codes more efficient: for example, to lengthen the vector length, avoid the memory bank conflict, and so on. In order to make these tuning activities easier, the SX system provides two utilities named ANALYZER/SX and VECTORIZER/SX.

ANALYZER/SX analyzes and reports the dynamic characteristics of source programs. The information reported is as follows.

- For a whole program:
  - its vectorization ratio indicating whether it is already vectorized enough or not
- For each program unit:
  - the number of times it is called
  - its vectorization ratio
  - its "cost" which is an approximated index of the time spent by it
  - the total number of the DO loops processed, the number of the vectorized DO loops and the ratio of their costs
- For each DO loop:
  - its cost
  - the average loop length
  - vectorization diagnostic messages which indicate why it is not vectorized
- For each statement:
  - its execution frequency
  - its cost
  - a mark indicating whether it is vectorized or not

ANALYZER/SX can get these data by inserting some statements for measuring them in the source program, executing the program on the CP or optionally on the AP and collecting them after execution. Fig.4 shows an output listing by ANALYZER/SX. The cost and vectorization ratio is calculated by accumulating execution frequencies of each statement weighted by the number of instructions generated for it.



```
*****                                                                              *****
*****   TOTAL EXECUTION ( CPU ) TIME =   0 :  0 '  0 " 269    (         269 MSEC)  *****
*****   TOTAL EXECUTION FREQUENCY    =              920201                         *****
*****   TOTAL VECTORIZATION RATIO    =  48.54%                                     *****
*****                                                                              *****

ATR      PROGRAM        FREQUENCY    EXEC COST(%)    V.RATIO        LOOP          V.LOOP    V.LOOP RATIO

-->      MAIN                   1            1.21      97.93           1               1          100.00
SUB      RCRTF                  2           93.96      50.29           5               4           52.62
SUB      RCRTBI                 2            2.92       0.00           2               0            0.00
SUB      RMPRM                  2            0.24       0.00           1               0            0.00
SUB      PRTMTR                 2            0.20       0.00           1               0            0.00
SUB      PRTPVT                 2            0.07      61.40           1               1          100.00
SUB      INDATA                 2            1.31       0.00           4               0            0.00
SUB      IMPUTB                 2            0.07      77.76           1               1          100.00
SUB      PRTIME                 8            0.01       0.00           0               0            0.00
SUB      TTIMER                17            0.00       0.00           0               0            0.00
SUB      CPTIME                 0            0.00       0.00           0               0            0.00
```

(a) A Summary Listing

```
'ELN     ILN  EXECUTION COST(%)    LOOP             FORTRAN STATEMENT

000980    1          2                                   SUBROUTINE RCRTF( A,N,IA,EPS,IT,IND )
000990    2                                              REAL            A,EPS,TOL,AMAX,GMAX,W,WI,WS1,WS2
001000    3                                              INTEGER  IA,IND,IT,IPIVOT,N,I,J,K,M,MI,M2
001010    4                                              DIMENSION  A( IA,N ),IT( N )
001020    5          2(   0.00)                          IND=0
001030    6          2(   0.00)                          IF( IA.LT.N ) GO TO 100
001040    7          2(   0.00)                          IF( N.LE.0)  GO TO 110
001050    8          2(   0.00)                          IF( N.LE.1 )  GO TO 120
001060    9          2(   0.00)                          AMAX=0.0
001070   10          2(   0.00)                          DO 20 I=1,N
001080   11        150(   0.04)   |------------>         |  DO 10 J=1,N
001090   12      12500(   1.55)   | V---------->         |  |  WS1=ABS(A(I,J))
001100   13      12500(   0.78)   | |                    |  |  IF( WS1.GT.AMAX)  AMAX=WS1
001110   14      12500(   1.94)   | V-----------      10 |  CONTINUE
001120   15        150(   0.01)   |-------------      20 CONTINUE
001130   16          2(   0.00)                          TOL=EPS*AMAX
001140   17          2(   0.00)                          WRITE(6,3000) TOL
001150   18                                        3000 FORMAT(1H ///20X,4HTOL=,E15.6)
001160   19          2(   0.00)                          DO 90 M=1,N
001170   20        150(   0.01)   3------------>         |  IF( M.LE.1 ) GO TO 50
001180   21        148(   0.01)   |                      |  MI=M-1
001190   22        148(   0.05)   |                      |  DO 40 I=M,N
001191   23                       |                      |  |  *VDIR NODEP(A)
001200   24       6175(   1.92)   | 4---------->         |  |  DO 30 K=1,MI
001210   25     187475(  34.90)   | | V-------->         |  |  |  A(I,M)=A(I,M)-A(I,K)*A(K,M)
001220   26     187475(   5.82)   | | V---------    30 |  |  CONTINUE
001230   27       6175(   0.57)   | 4-----------    40 |  CONTINUE
001240   28        148(   0.02)   |                      |  CALL OVERFL(J)
001250   29        148(   0.01)   |                      |  IF( J.EQ.1) GO TO 130
001260   30        150(   0.03)   |                 50 |  GMAX=ABS(A(M,M))
001270   31        150(   0.01)   |                      |  IPIVOT=M
001280   32        150(   0.01)   |                      |  M2=M+1
001290   33        150(   0.01)   |                      |  IF( M.GE.N) GO TO 65
001300   34        148(   0.06)   |                      |  DO 60 I=M2,N
001310   35       6175(   0.77)   | V---------->         |  |  WI=ABS(A(I,M))
001320   36       6175(   0.38)   | |                    |  |  IF(WI.LE.GMAX) GO TO 60
001330   37          0(   0.00)   | |                    |  |  GMAX=WI
001340   38          0(   0.00)   | |                    |  |  IPIVOT=I
001350   39       6175(   0.57)   | V-----------    60 |  CONTINUE
001360   40        150(   0.01)   |                 65 |  IF( GMAX.LE.TOL) GO TO 140
001370   41        150(   0.01)   |                      |  IF( IPIVOT.EQ.M ) GO TO 75
001380   42          0(   0.00)   |                      |  DO 70  I=1,N
001390   43          0(   0.00)   | V---------->         |  |  W=A(M,I)
001400   44          0(   0.00)   | |                    |  |  A(M,I)=A(IPIVOT,I)
001410   45          0(   0.00)   | |                    |  |  A(IPIVOT,I)=W
001420   46          0(   0.00)   | V-----------    70 |  CONTINUE
```

(b) A Format Listing

Fig.4 ANALYZER/SX Output Listing

ANALYZER/SX has a special feature to measure and report only the execution time for each program unit. Users can see real effect of vectorization by comparing execution time in scalar mode and vector mode for each program unit by using this function. When executed on the AP, the vectorized operation ratio is measured by using hardware counters and also output in the listing.

ANALYZER/SX has another features to analyze static characteristics of a source program, such as a tree diagram depicting a module structure of a program, a COMMON data cross reference listing, a module cross reference listing and a listing showing the correspondence between actual arguments and dummy arguments. This information is useful when considerable modification is necessary.

VECTORIZER/SX is the utility program to help users tune their programs to get higher vectorization ratio. It works in a screen mode of the TSS environment. The dynamic characteristic information of the program to be tuned is passed from ANALYZER/SX through a file so that users can see it through a terminal screen. Vectorization diagnostic messages are also passed from FORTRAN77/SX compiler so that they can know why DO loops are not vectorized. In addition, VECTORIZER/SX has screen editor functions in it, and the above information is merged with source program lines in an edit buffer. Consequently, users can modify their source programs directly while referring to necessary information. An operational flow of VECTORIZER/SX is shown in Fig.5. Its program editing panel is also shown in Fig.6.

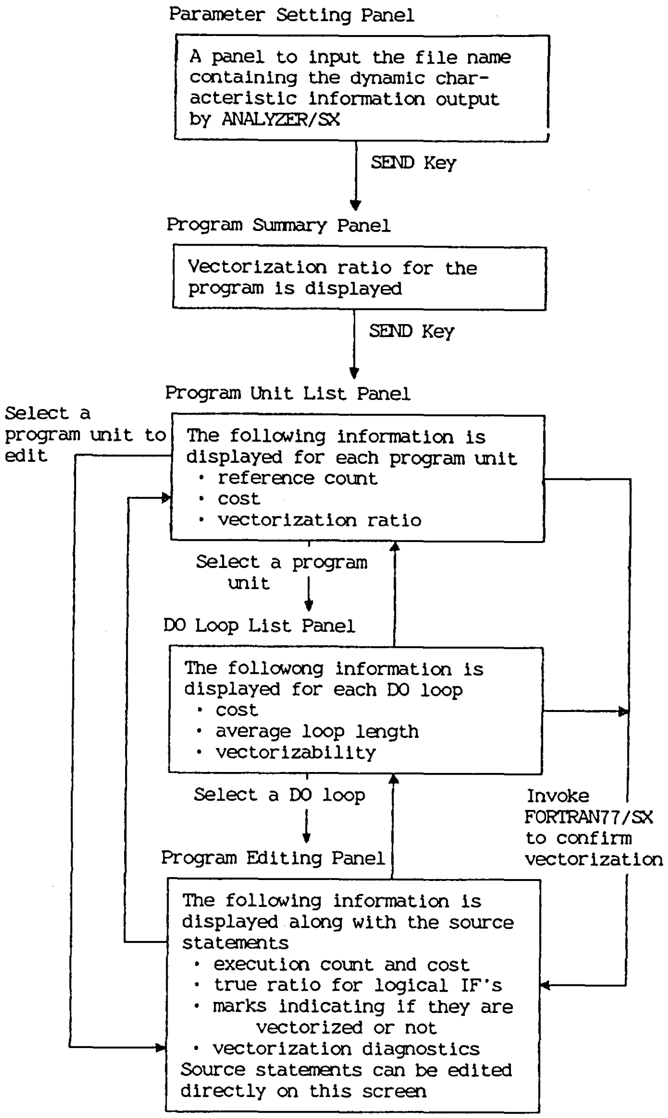
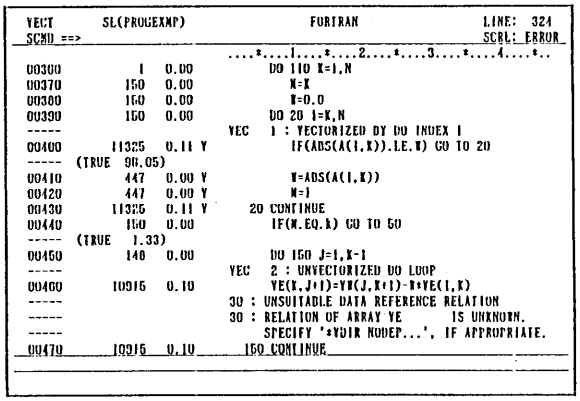

Parameter Setting Panel
A panel to input the file name containing the dynamic characteristic information output by ANALYZER/SX
SEND Key
Program Summary Panel
Vectorization ratio for the program is displayed
SEND Key
Program Unit List Panel
Select a program unit to edit
The following information is displayed for each program unit
• reference count
• cost
• vectorization ratio
Select a program unit
DO Loop List Panel
The followong information is displayed for each DO loop
• cost
• average loop length
• vectorizability
Select a DO loop
Invoke FORTRAN77/SX to confirm vectorization
Program Editing Panel
The following information is displayed along with the source statements
• execution count and cost
• true ratio for logical IF's
• marks indicating if they are vectorized or not
• vectorization diagnostics
Source statements can be edited directly on this screen

Fig.5 An Operational Flow of VECTORIZER/SX



```
VECT      SL(PROGEXMP)                    FORTRAN                 LINE:  324
SCMD ==>                                                          SCRL: ERROR
                                ....*....1....*....2....*....3....*....4....*..
00360           1   0.00              DO 110 K=1,N
00370         150   0.00                M=K
00380         150   0.00                W=0.0
00390         150   0.00              DO 20 I=K,N
-----                           VEC   1 : VECTORIZED BY DO INDEX I
00400       11325   0.11 Y              IF(ABS(A(I,K)).LE.W) GO TO 20
----- (TRUE  98.05)
00410         447   0.00 Y              W=ABS(A(I,K))
00420         447   0.00 Y              M=I
00430       11325   0.11 Y         20 CONTINUE
00440         150   0.00              IF(M.EQ.K) GO TO 50
----- (TRUE   1.33)
00450         140   0.00              DO 150 J=1,K-1
-----                           VEC   2 : UNVECTORIZED DO LOOP
00460       10915   0.10              VE(K,J+1)=VW(J,K+1)-W*VE(I,K)
-----                           30 : UNSUITABLE DATA REFERENCE RELATION
-----                           30 : RELATION OF ARRAY VE       IS UNKNOWN.
-----                                SPECIFY '*VDIR NODEP...', IF APPROPRIATE.
00470       10915   0.10          150 CONTINUE
```

Fig.6 A Program Editing Panel of VECTORIZER/SX

CONCLUSION

The outlines of FORTRAN77/SX and some tuning utilities of the SX system have been described. Now the use of supercomputers has become very common, and the programmers who have no special knowledge on them have begun to use them. As we have mentioned, we think that the ease of use is essential to today's supercomputers. Furthermore, high performance must be attainable at the same time. We believe we could achieve these goals with above software products.

REFERENCES

[1] NEC: "NEC Supercomputer SX-1E/SX-1/SX-2 General Description", Pub. No. GAZ01E-2, 1985.

[2] T. Furukatsu, T. Watanabe and R. Kondo: "Supercomputer SX system with a vector peak performance of 1.3 Gflops and 6 nanosecond cycle time", Nikkei Electronics, 1984.11.19, No. 356, pp 237-272 (in Japanese).

[3] T. Watanabe: "Architecture of Supercomputers - NEC Supercomputer SX System", NEC Research & Development, No.73, pp 1-6, April 1984.

THE IX SUPERCOMPUTER FOR KNOWLEDGE BASED SYSTEMS

Tetsuya HIGUCHI, Tatsumi FURUYA, Hiroyuki KUSUMOTO,
Ken'ichi HANDA, and Akio KOKUBU

Electrotechnical Laboratory
1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki 305, Japan

Abstract. This paper describes an IXM machine for the parallel processing of large knowledge based systems written in a semantic network language, IXL. The potential performance of IXM is thousands faster than conventional machines, as IXM can find all the solutions of the query simultaneously. IXM consists of a pyramid-shaped network(PYN) and one thousand proccesing elements(PEs). IXM utilizes two thousand associative memories in order to exploit the concurrencies in the fundamental operations of semantic network: association, set operation, and marker propagation. A network processor at each node of the PYN has an associative memory for parallel marker propagation. A PE also has two types of associative memories; one for the parallel processing of association and set operation, the other for exploiting the parallelism in IXL language interpreter described in non-procedural instructions.

1. INTRODUCTION

The semantic network is one of the most important representation schemes to develop knowledge based systems[1,2]

The authors are developing the IX[iks] system which includes a semantic network language for knowledge representation (IXL)[3,4] and a massively parallel hardware (IXM)[5]. The objective of the IX system is to support all the processes of semantic network processing in an integrated fashion: from using IXL in the modeling and description of knowledge based systems to their parallel execution on an IXM machine.

This paper focuses on the architecture of IXM. IXM is a high performance multiprocessor thousands of times faster than current machines used for semantic network processing. IXM consists of a pyramid-shaped network and one thousand processing elements.

It is very important for the design of the semantic network machines[6,7] to utilize the parallelism in three fundamental operations in the semantic network: marker propagation, association, and set operation.

Therefore, the network processor located at every node of the pyramid-shaped network employs an associative memory in order to exploit the concurrency in marker propagation.

Furthermore, the processing element also includes two kinds of associative memory; one to perform association and set operations in the SIMD manner, and the other to exploit the concurrency in an IXL language interpreter.

Section 2 describes the background of the semantic network and the programming language IXL since the IXM architecture design is IXL oriented. Section 3 is an overview of IXM. The design features are discussed in Section 4. Section 5 describes the processing element and Section 6 describes the pyramid-shaped network.

2. BACKGROUND

2.1 Semantic network

The advantage of the semantic network is the representation of declarative knowledge. Taxonomical knowledge and inheritance can be represented more naturally than predicate logic. Inheritance of properties from the super class is very useful to describe especially large knowledge bases because it can save memory spaces. Compared with frame systems, the meaning of the relation between concepts can be defined more flexibly. Procedural knowledge, however, can not be expressed well using only the semantic network. Therefore, semantic network languages have to include methods for procedure addition.

The semantic network can be processed efficiently by assigning marker bits to each node of the network2. A marker bit is a one-bit flag on which the result of processing is stored. Fundamental operations in the semantic network are performed using maker bits. For example, the question, "Which member belongs to both A-group and B-group ?" is asked in the semantic network of Figure 1.

In order to find the solution, the association operation is executed firstly to set the marker bit No.1 of the A-group node. Then the marker propagation is executed to set the marker bits No.1 of the nodes which belong to the lower hierarchy of the A-group along the 'isa' links, starting from the A-group node going downward. These nodes represent members of the A-group. As well as the A-group members, the marker bits No.2 of the B-group members are set using association

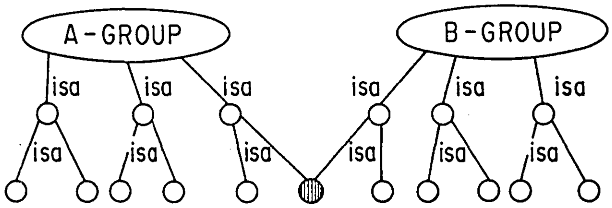


Figure 1. A Semantic Network

CH2345-7/86/0000/1041$01.00 © 1986 IEEE

and marker propagation. Finally, the set operation is executed to find the intersection of the A-group and the B-group; marker bits No. 1 and No.2 of each node is ANDed.

Parallelism in marker propagation can be utilized to greatly reduce the execution time in large knowledge bases. This is because marker propagation can find all the members of a set in the length of time proportional to the depth of the tree. Therefore, if the set is represented as a binary tree of a million nodes, only 20 steps are required to mark all the members. Association includes as many concurrencies as there are nodes in the semantic network. This also applies to set operations. As will be mentioned later, the IXM machine can exploit these parallelisms fully by using associative memories.

2.2 IXL

IXL is a semantic network language for knowledge representation. The design philosophy of IXL is to solve the problems of existing semantic network languages: the description capability of relations themselves, the method of procedure addition, and the treatment of negative knowledge.

As shown in Figure 2, IXL is a command language which looks like a logic predicate. IXL can perform (1) modifications, (2) inquiries, (3) descriptions of knowledge bases. For example, an IXL command for knowledge description, 'link(is_a, penguin, bird)', is used to state that a penguin is a bird. Once this knowledge is stored and an IXL command for inquiry,'isa(penguin,X)', is issued, the answer becomes bird.

The main features of IXL are:

1) User-defined relation represented not by a link but by a network of nodes in order to define the meaning of the relation precisely and flexibly.

2) Procedural knowledge described in logic-based expressions.

3) Negative knowledge explicitly described.

A problem with the semantic network is the expression of procedural knowledge which is used to describe complicated inference rules for knowledge bases. IXL defines procedural knowledge in the form of a clause in Prolog. For example, the following procedural knowledge represents an inference rule to determine whether a restaurant is classified as high-class or not:

```
isa( X, high_class_restaurant):-
                    asst( entree, X, Y),
                    asst( main_dish, X, Z),
                    Z + Y > 100.
```

This rule classifies a restaurant, X, as a high-class restaurant if the sum total of the entree and the main dish exceeds 100 dollars. The predicates in the above clause are all IXL commands. Arithmetic and logical expressions are also allowed to appear in the body of the clause. Notice that IXL commands can be used to describe both declarative and procedural knowledge. Therefore, once the IXM machine is designed to execute IXL commands efficiently, the IXM can process both types of knowledge uniformly.

3. IXM SYSTEM ARCHITECTURE

3.1 Overall structure

IXM operates under the control of the host computer. IXM consists of the pyramid-shaped network and processing elements (PEs) as shown in Figure 3.

The pyramid-shaped network has a network processor at each node of the network. One network processor at the bottom layer of the pyramid-shaped network connects four PEs. Each upper network processor is connected to four lower network processors. In addition, a network processor is connected to four adjacent network processors of the same layer. A network processor includes message routing logic and an associative memory to exploit the concurrency in marker propagation.

The PE includes two kinds of associative memories. The first one is for the storage and the processing of the partitioned semantic networks, for a large semantic network for knowledge bases is partitioned into sub-semantic networks and distributed among the PEs. Association and set operations are executed using the function of the associative memory in parallel and equal to the number of the memory words.

The second one is to store the IXL language

```
To connect nodes by a link:
 link(is_a, X, Y).
 link(not_isa, X, Y).
 link(instance_of, X, Y).
 link(a_kind_of, [X,Y,..], Z).
 link(source, R, X).
 link(destination, R, Y).
 link(rule, X, ((
   asst(R, X, Y):- ....
   prop(R, X, Y):- ....
   isa(X, Y):- ....
   instance(X, Y):- ....)).

To construct a relation:
 assertion(R, X, Y).
 property(R, X, Y).
```

```
To inquire about a link:
 isa(X, Y).
 instance(X, Y).
 ako(X, Y).
 source(R, X).
 destination(R, Y).

To inquire about a relation
 asst(R, X, Y).
 prop(R, X, Y).

System-supported primitive links;
 is_a,  not_isa,  instance_of
 not_instanceof,  a_kind_of,
 source,  destination

X,Y: concept node
R: relational node
```

Figure 2. The list of IXL commands

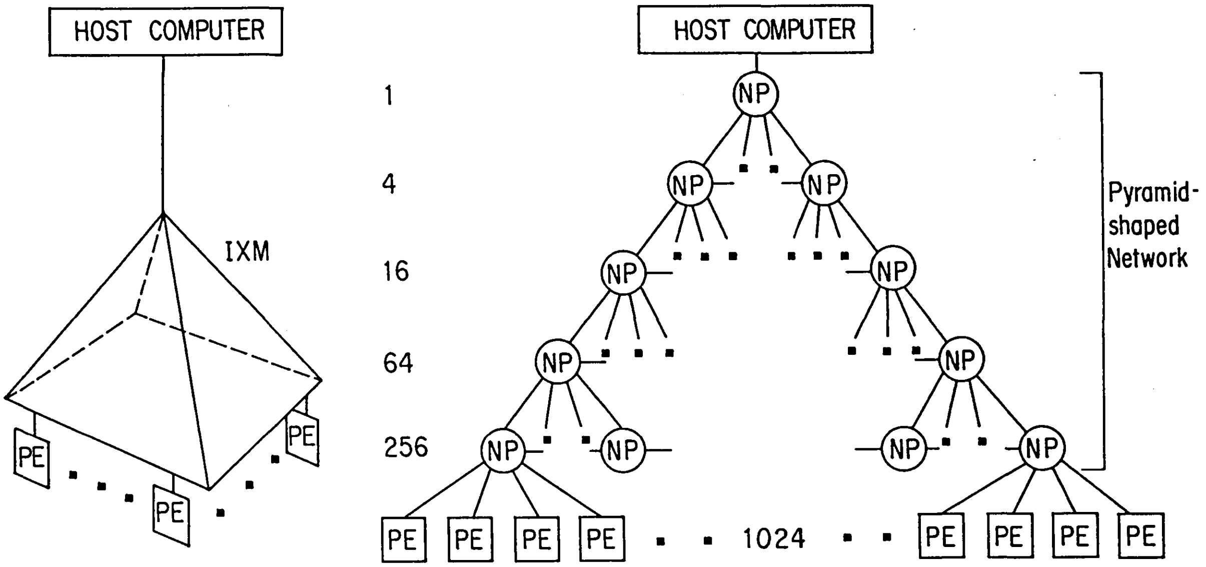


Figure 3. System organization

interpreter. The interpreter is described in the IXM machine instruction set of which the execution is asynchronous; the associative memory is used to fetch asynchronous IXM machine instructions.

A microprogrammed control unit is also included in the PE to execute IXL commands and arithmetic operations specified in procedural knowledge.

3.2 Instruction levels

There are three instruction levels in the IXM; (1) IXL commands, (2)IXM machine instructions, and (3) IXM micro instructions.

The interface between the IXM and its user is an IXL command. For example, user inputs an IXL command, 'is_a(penguin, X)' into the host computer in order to find out what a penguin is. The host computer accepts the IXL command and broadcasts it to each PE via the pyramid-shaped network, as shown in Figure 4. The IXL command is input into the IXM one at a time from the host computer; synchronization is needed each time a new IXL command is input into IXM. All the PEs execute the same IXL command in parallel because each PE stores the IXL language interpreter and the sub-semantic network in two associative memories.

The IXL language interpreter consists of subroutines; a subroutine is a non-procedual program and executes an IXL command asynchronously. Because subroutines are written in IXM machine instructions, an IXL command is executed in IXM by the IXM machine instructions.

Execution of IXM machine instructions are asynchronous to exploit the parallelism existing in IXL commands. During the execution of an IXL command, PEs require no synchronization with each other; a processing element can return the answer to the host computer as soon as the solution of the IXL instruction is found in the PE. This concurrency speeds up the IXL interpreter. Notice here the difference in execution modes; the IXM machine instructions are executed asynchronously (MIMD mode) , while the IXL commands are executed synchronously (SIMD mode).

IXM micro instructions are used mainly for the emulation of the IXM machine instructions.

4. DESIGN FEATURES

4.1 Nodes of equivalence for parallel marker propagation

Marker propagation can mark all the members of a set with O(log N) if the set is represented in a tree-shaped semantic network of N nodes. It seems to be very effective in speeding up large semantic network processing. However, it is not necessarily true when many links concentrate on one node, since such a node would have to repeat propagations as many times as the number of the links connected to it and would become a bottleneck in marker propagation. Nodes with hundreds of links appear frequently in semantic

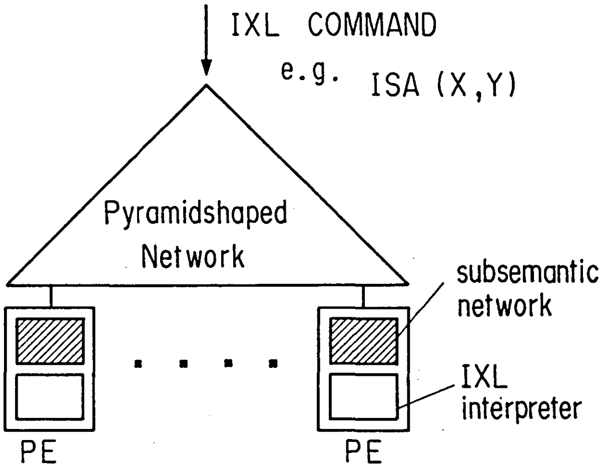


Figure 4. Execution of an IXL command

network applications because semantic networks are useful to represent taxonomical knowledge.

The authors introduced the idea of 'nodes of equivalence' in order to solve this problem and exploit concurrency in marker propagation. The idea is to partition the bottleneck node into nodes with a smaller number of connecting links and to distribute them among the PEs. A partitioned node is called a node of equivalence. For example, the student node in Figure 5(a) is the bottleneck of marker propagation, because it must repeat propagations 26 times. Therefore, the student node is divided,for example,in this case, into three nodes of equivalence as shown in Figure 5 (b). If the bottleneck node is partitioned into N nodes of equivalence, the parallelism increases N times.

If a message is sent to one of the nodes of equivalence, the message hasto be duplicated and sent to the other nodes of equivalence. For example, the hatched PEs in Figure 6 (i.e. B,C,D) contain the nodes of equivalence. If a message comes from the PE A and the destination is B, the message is duplicated and sent to C and D by the network processors of the pyramid-shaped network. The details are described in Section 6.

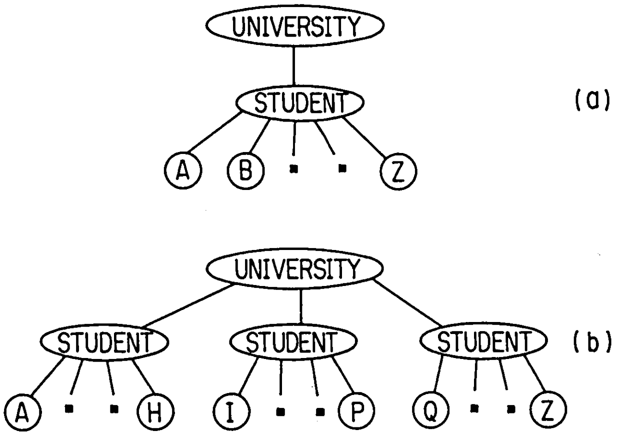


Figure 5. The nodes of equivalence

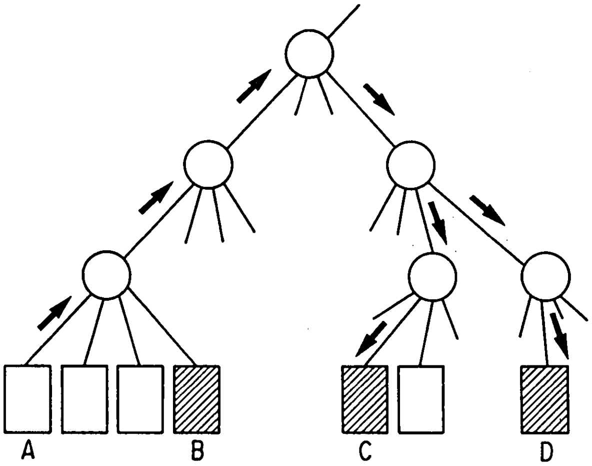


Figure 6. Message transfer using nodes of equivalence

4.2 Asynchronous IXM machine instructions

The authors have proposed an asynchronous execution mechanism of IXM machine instructions in order to utilize the parallelism in the IXL interpreter. As mentioned earlier, the IXL interpreter consists of subroutines written in IXM machine instructions; a subroutine corresponds to an IXM command. The parallelism in the IXL interpreter can reduce considerably the total execution time of the IXM.

For example, suppose the IXL command, 'is_a(X,animal)', is entered in order to find the solutions which come under the heading of animal. In Figure 7, the solutions are bird, penguin, and robin. Marker propagation marks these solutions, starting from the animal node. Therefore, the bird node is marked earlier than any other node.

So, if the bird node is returned immediately to the host computer as a solution (without waiting for the completion of marker propagation), the turn around time can be reduced to a large extent. In order to realize this, IXM machine instructions must be asynchronous.

Now we shall explain the asynchronous execution mechanism. Here it is assumed that each node has a subroutine for an IXL command to be executed; the hatched rectangular in Figure 7 represents a subroutine. The subroutine is written in asynchronous IXM machine instructions. An IXM machine instruction in the subroutine specifies a marker bit in the input marker field of the instruction format; as soon as the marker bit of a node is set, the IXM machine instruction can be started to the node, independent of the other nodes.

In Figure 7, if the marker bit No.1 of the bird node is set by marker propagation from the animal node, the bird node searches its subroutine for the IXM machine instruction which specifies the marker bit No.1 as the input marker. If there are two such IXM machine instructions in the subroutine and if they are the return and the marker propagation instructions, the bird node can immediately return its identifier to the host controller and then mark the marker bit No.1 of the penguin node. Thus,the asynchronous execution of the IXM machine instructions utilizes the parallelisms existing in the IXL interpreter.

The PE employs an associative memory to find

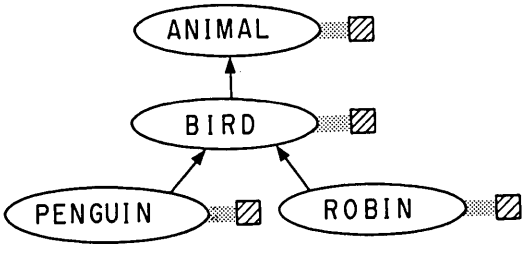


Figure 7. Taxonomical knowledge

executable IXM machine instructions quickly. The details are described in Section 5.

4.3 Associative memory to store and process semantic networks

The association operation and the set operation are very important in the IXM design because they appear in semantic network processing very frequently as well as marker propagation. The IXM employs an associative memory for each PE to utilize the parallelism in these operations.

The large semantic network of the knowledge base is partitioned into sub-semantic networks. Each sub-semantic network is stored in the associative memory of a PE. Therefore, multiple nodes are stored in a PE. For these nodes, the association operation and the set operation are executed in the SIMD manner, using the function of the associative memory.

Because the IXM consists of one thousand PEs, it is an advantage that the IXM design highly utilizes the associative memories suitable for VLSI implementation.

5. PROCESSING ELEMENT

As shown in Figure 8, the PE of the IXM consists of three components: an associative memory for semantic networks (SNAM), an associative memory for the IXL interpreter (IRAM), and a microprogrammed control unit (CU).

SNAM stores a sub-semantic network which is a partition of the large semantic network. Each word of SNAM contains not only a node but also the marker bit field where the processing results to the node are stored. Using its association function, the set operation and the association operation are executed simultaneously for all the data in the memory.

IRAM stores the IXL interpreter described in the IXM machine instructions. If a marker bit of a node in an SNAM word is set by the semantic network operation, then it causes the execution of another IXM machine instruction. Therefore, IXM machine instructions are asynchronous just like dataflow instructions[8]. IRAM finds executable IXM machine instructions by its associative function.

CU is a microprogrammed processor and it controls SNAM and IRAM to perform three functions: the execution of IXL commands, arithmetic and logical operations, and the processing of communication packets sent from either the host controller or other PE's. CU executes the IXM micro instructions.

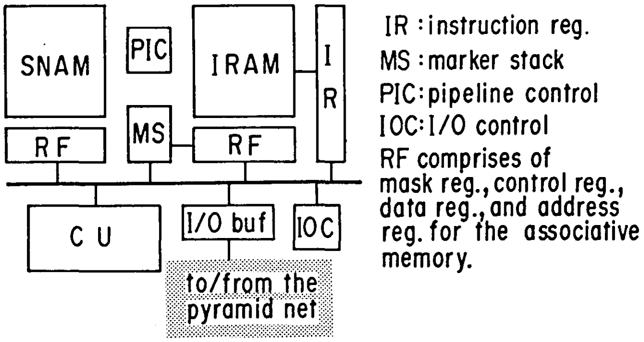


Figure 8. PE block diagram

5.1 Execution cycles of an IXM machine instruction

The microprogrammed control unit (CU) controls the fetch and the execution of an IXM machine instruction, as shown in Figure 9. Firstly, an IXM machine instruction is fetched from IRAM. Secondly, the IXM machine instruction is executed by CU, using SNAM. The marker bit of SNAM is updated as the result of the execution.

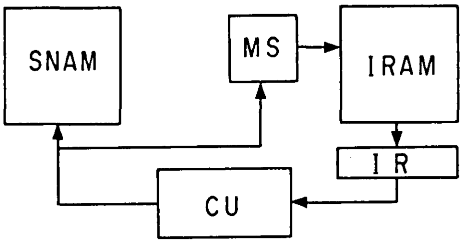


Figure 9. The execution mechanism of an IXM machine instruction

When the marker is updated, the marker bit number is also stored into the marker stack (MS). Thirdly, CU searches IRAM for the next executable IXM machine instruction, using the marker bit number in the marker stack, since the update of a marker bit enables the execution of another IXM machine instruction. These cycles are repeated to find the solutions of an IXL command.

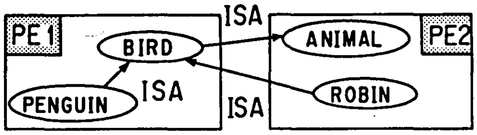
5.2 SNAM

Figure 10 shows the organization of SNAM and how the partitioned semantic networks are stored in SNAMs. The semantic network in the figure is partitioned into two sub-networks and stored separately in two SNAMs.

SNAM is 2K-word associative memory with a width of 64 bits. One word consists of four fields: the identifier field (21 bits), the destination field (21 bits), the link name field (4 bits), and the marker bit field (18 bits).

An SNAM word represents a link and the node which is connected to the link. A link occupies two SNAM words , since a link has two nodes on both edges. For example, as shown in Figure 10, the 'isa' link between the animal node and the bird node occupies the two words: the first word of SNAM1 and the first word of SNAM2. The identifier field distinguishes between the two million links stored in the IXM. The link name field indicates a system-support link such as 'isa' and 'instance_of' ; four bits are required to distinguish between the system-support links (a user-defined link is translated into system-support links). The destination field consists of the PE number and the displacement within SNAM where the node is stored; for example, the destination of the robin node at the first word of SNAM2 is represented as 'PE1(1)', because the animal node, which is the destination of the robin node, is located at the first word of SNAM1 in PE1.

The marker bit field contains 18 marker bits (numbered from 0 to 17). Each marker bit holds the result of the semantic network processing such as association and set operations. For example,



IDF(21bit)	DSF(21bit)	LN(4b)	MBF(18b)
BIRD	PE2(1) ANIMAL	ISA	·
BIRD	PE1(4) PENGUIN	RISA	·
BIRD	PE2(2) ROBIN	RISA	·
PENGUIN	PE1(1) BIRD	ISA	·
SNAM1 (PE1)			

ANIMAL	PE1(1) BIRD	RISA	·
ROBIN	PE1(1) BIRD	ISA	·
SNAM2(PE2)			

LEGEND IDF: identifier field LN: link name
DSF: destination field -> PE No.(displacement)
MBF: marker bit field RISA: reverse isa

Figure 10. SNAM organization and the partition of a semantic network

suppose an IXM machine instruction, 'ASSOC(bird, 2)', is executed in the IXL interpreter program, this instruction is for the association operation. It searches for the SNAM words which contain the bird node. If it exists, each marker bit No.2 of the corresponding words is set; in this example, the No.2 marker bits of the three words in SNAM1 are set.

When the marker bit is set in SNAM, the marker bit number is also stored in the marker stack. The marker stack is accessed by IRAM to find IXM machine instructions which have become executable.

Using SNAM has two advantages. The first one is the parallel processing of the association and the set operations. The IXM can process both of them in O(c) steps, while the algorithms for them on a sequential machine take O(n) and O(n*log n) steps respectively. The second advantage is that the establishment of new links is relatively easy. The two new words of SNAM are added in order to establish a new link between the two concepts.

5.3 IRAM

Figure 11 shows the organization of IRAM. Each word consists of the operation code field (3 bits), the input marker field (5 bits), the argument field (21 bits) and the resulting marker field (5 bits). The input marker field specifies a marker bit of SNAM. If an IXM machine instruction specifies a marker bit in the input marker field, the IXM machine instruction becomes executable as soon as the marker bit is set. The resulting marker field specifies the marker bit which is to be set as the result of the execution.

The IXL interpreter consists of subroutines for IXL commands and each subroutine is written with these IXM machine instructions. Figure 12 shows an IXL command, 'isa(penguin,X)', and its subroutine written with IXM machine instructions. This IXL command searches for the higher-class concepts of the penguin. In order to find the solution, two steps are required. First, the penguin node is determined. Then, 'isa' links are traversed starting from the penguin node. The nodes on the traversed 'isa' links become the solutions of 'isa(penguin,X)'; they are the bird node and the animal node.

op-code field (3bit)	input marker field (5bit)	argument field (21 bit)	resulting marker field (5bit)
ASSOC	*	PENGUIN	0
MARK	0	ISA	0

Figure 11. IRAM organization

```
IXL command:    ISA(penguin, X)

IXM machine instruction:
                ASSOC(penguin,0)
                MARK(0, isa, 0)
```

Figure 12. An IXL command and the subroutine for it

The subroutine to implement this IXL command is stored in IRAM as shown in Figure 11. At first, 'ASSOC(penguin, 0)' is selected from IRAM. After the execution, the marker bit No.0 of the fourth word of SNAM1 in Figure 10 is set because it is the penguin node. After the marker bit No.0 is set, IRAM is searched to find the IXM machine instruction which has the marker bit No.0 in the input marker field. Then, 'MARK(0,is_a,0)' is selected.

Figure 13 explains the MARK instruction. Suppose 'MARK(n1,is_a,n2)' is to be executed. If node A has the marker bit No.n1 which is set and if an 'isa' link emerges from node A, the MARK instruction becomes executable. As the result of the MARK instruction, the marker bit No.n2 of node B which is connected to node A with the 'isa' link is set.

Therefore, if MARK(0,is_a,0) is executed to the example of Figure 6, the marker bit No.0 of the bird node is set. Then, MARK(0,is_a,0) is selected once again from IRAM and is executed. So, the marker bit No.0 of the animal node is set. Thus, the execution of the IXM machine instruction proceeds asynchronously.

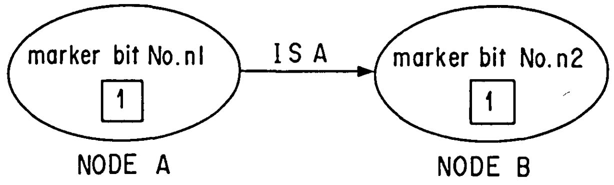


Figure 13. MARK instruction

5.4 Control unit (CU)

Processing by CU is roughly classified into the following two types. The first type is the emulation of IXL machine instructions. The IXM machine instruction such as MARK is executed in terms of IXM micro instructions. As mentioned in Section 5.1, read and write operations to SNAM and IRAM are frequently done in order to fetch and execute IXM machine instructions. These associative memories are controlled with IXM micro instructions.

The second type of CU processing is the arithmetic and logical operations which appear in the description of procedural knowledge; the inference rule mentioned in Section 2.2 is shown again as an example.

```
isa( X, high_class_restaurant):-
                     asst( entree, X, Y),
                     asst( main_dish, X, Z),
                     Z + Y > 100.
```

The first IXL command in the body, 'asst(entree,X,Y)', looks up the price of the entree,'Y'. The second looks up the price of the main dish, 'Z'. The third calculates the sum and decides whether the sum exceeds 100 dollars or not.

Although the rule looks like a clause in Prolog, notice that all the solutions of an IXL command can be determined simultaneously. Solutions of a predicate in Prolog are determined sequentially. Therefore, the IXM can be regarded as a parallel logic machine taking this viewpoint.

6. PYRAMID SHAPED NETWORK

The structure of the pyramid-shaped network (PYN) is shown in Figure 3. Network processors are connected in the shape of a pyramid. However, the connection path between the PEs of the same layer of the network is not used in the first version of the IXM for the convenience of implementation.

PYN is used for the following three types of packet communication.

(1) Host computer-to-PE: Broadcast of an IXM command.

(2) PE-to-PE : Marker, value, and (or), node identifier are transferred from PE to PE.

(3) PE-to-Host computer : Collection of computation results.

The destination of the packet in the PE-to-PE communication is the node identifier to which the packet is sent. The node identifier consists of the PE number and the displacement in the SNAM where the node is stored. The network processor routes the packet according to the node identifier. The network processor always watches the node identifier to distinguish whether the node is the member of the particular nodes of equivalence or not. If the packet is sent to a member of the nodes of equivalence, the network processor copies the packet and send the copies to all the members of the nodes of equivalence. For this operation, a table for nodes of equivalence is stored in an associative memory of each network processor.

6.1 The node of equivalence

Figure 14 shows the marker propagation to the node of equivalence. It shows the case of the university node in PE No.1 sending a packet to the

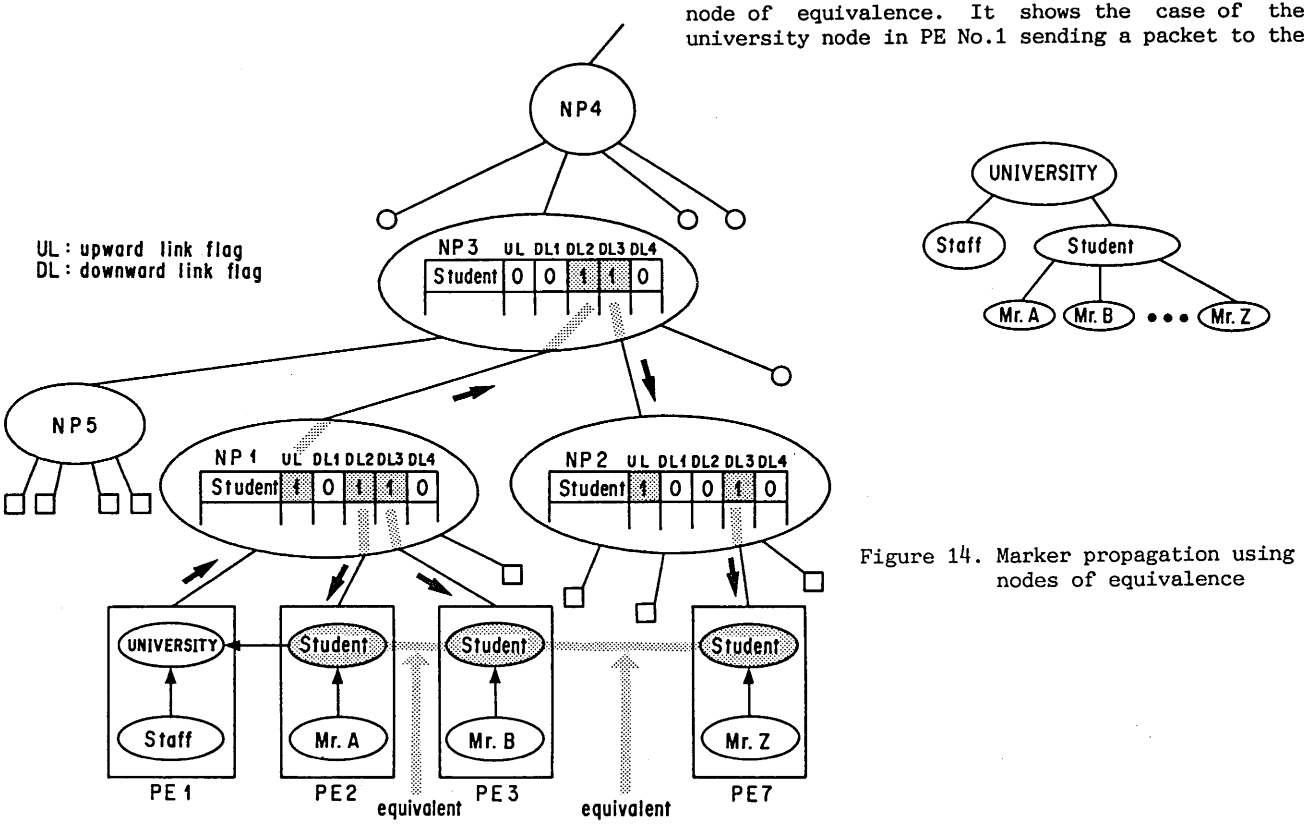


Figure 14. Marker propagation using nodes of equivalence

nodes of equivalence,'student'. The 'student' nodes are distributed in processing elements No.2, No.3, and No.7. Therefore, the packet has to be copied and the copied packets have to be sent to these PE's by the network processor No.1, No.2, and No.3.

The network processor has a table for nodes of equivalence (EQNT). Each item of EQNT consists of the node identifier of the node of equivalence and the routing information to deliver the copied packets.

The routing information is in five flags indicating the distribution of the nodes of equivalence among PEs. Four bits correspond to four downward links to four lower network processors or PE's, and the remaining one bit corresponds to one upward link to an upper network processor in the network hierarchy.

It is required to determine which network processors have the EQNT items for a particular node of euivalence. For example, network processors No.1, No.2, and No.3 have the EQNT items for 'student' node as shown in Figure 14.

The algorithm to select the network processors where the EQNTs are to be located is as follows:

(A1) Select the PEs which contain the nodes of equivalence.

(A2) Find a root network processor of the minimum sub-tree which has all the PEs selected in (A1) as its leaf.

(A3) Traverse the connection path starting from each PE of (A1) to the root network processor selected in (A2).

(A4) Select the network processors traversed in (A3).

The routing information of each EQNT item is determined as follows:

(B1) If a network processor has downward links which are connected to either the network processors of (A4) or the PEs of (A1), then the downward link flags corresponding to the downward links are set.

(B2) If a network processor has an upward link which is connected to the network processor of either (A2) or (A4), then the upward link flag is set.

For example, the downward link flags No.2 and No.3 of the 'student' item in the network processor No.1 are set as shown in Figure 14.

6.2 Structure of network processor

The network processor consists of the following components:

(1) Input/output buffer queues for asynchronous communications between network processror and a PE, or between two network processors

(2) decision logic for packet routing

(3) the associative memory to store the table for nodes of equivalence (EQNT)

(4) the control processor

Figure 15 shows the block diagram of the network processor. A packet arriving at a network processor is queued into input buffer registers. Then, the destination of the packet is examined using the associative memory in order to identify whether the destination is a node of equivalence or not. If it is identified, the packet is copied and sent to the other member of the node of euivalence according to the routing information in EQNT. If the destination is not a node of equivalence, the packet is delivered using simple routing logic.

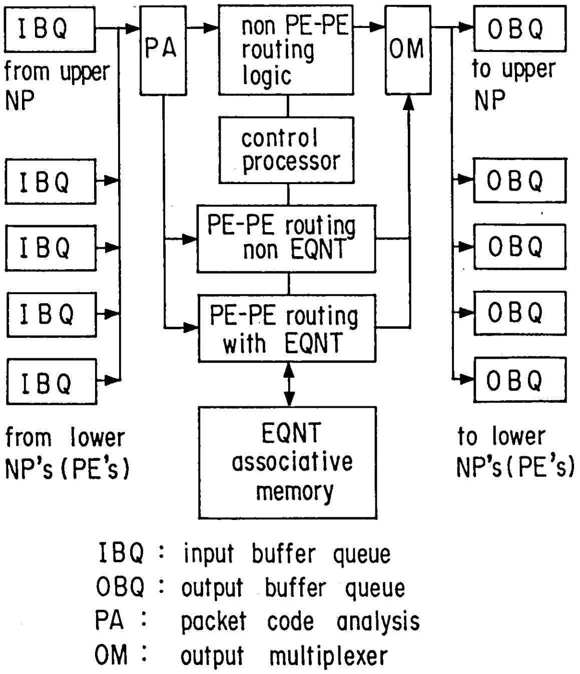


Figure 15. Network processor
- block diagram

7. PERFORMANCE

We show here the simulation results of the IXM machine and discuss its performance. At first, the execution time of semantic network processing on a conventional computer (Micro VAX II) is shown for the comparison with IXM.

We developed the IXL interpreter in K-Prolog on Micro VAX II. A French wine knowledge base has been written in IXL. A user can issue an IXL command to get various knowledge about wine. Table 1 shows the execution time of two types of queries to three knowledge bases of different size.

The query (1) asks the knowledge base whether a particular fact holds or not. For example, an IXL command, "prop(rate, 'chablis_wine', '**')",is issued to ask whether the chablis wine is rated as two-starred or not. As shown in Table 1, a user can get an answer (yes or no) in about two second, although the execution time depends on where the fact is located in the knowledge base.

The query (2) asks the knowledge base all the solutions of the query. For example, "bagof(X, prop(rate, X, '**'),L)" command replies all the wines rated as two-starred in the variable L. The execution time of this type of query increases abruptly as the knowlede base grows larger. A user have to wait fifteen minutes for the query to the knowledge base of 4500 links which is far from practical knowledge base. This is because of the exponential explosion of the search space. Even

Table 1. Execution time of the knowledge base(KB) search by IXL language processor on Micro VAX II

KB size	1600 links	3300 links	4500 links
query (1)	66 - 600 msec	66 - 1183 msec	66 - 1633 msec
query (2)	2.1 min.	8.7 min.	14.6 min.

Table 2. IXM machine cycles for the knowledge base search

KB size	1600 links	3300 links	4500 links
query (1)	1560	2112	2404
query (2)	2452	2756	3556

if the IXL interpreter is implemented not in Prolog but in Lisp, the result will show the same tendency as in Table 1.

On the other hand, as shown in Table 2, the result of IXM machine simulation shows that the total of IXM machine cycles required for query (2) does not increase abruptly even if the knowledge base grows. The potential performance of IXM is thousands faster than conventional computers. This is because IXM machine can find all the solutions in parallel using the procedures written in asynchronous IXM machine instructions.

We are currently designing the details of the IXM machine of which machine cycle is 100 nano second, assuming the associative memory of which access time is 300 nano second and 6 mega byte/sec transfer rate between network processors.

8. CONCLUSION

Knowledge bases of a few million nodes described in the semantic network will be used in practical applications within a decade. The authors have proposed a massively parallel architecture which highly utilizes associative memories to exploit the parallelism in semantic network processing.

The associative memories in IXM are used for three purposes. The first is the utilization of parallel structure in association and set operation. The second is the parallel marker propagation. The third is the exploitation of the parallelism in IXL interpreter.

The IXM machine is also a language (IXL) oriented machine. IXL gives a method to treat declarative and procedural knowledge uniformly. IXM machine can process both declarative and procedural knowledge efficiently by adopting the IXL oriented approach.

The simulation result shows that the potential performance of IXM machine is thousands faster than conventional machines. In order to utilize IXM fully for large knowledge based systems, it is also very important to develop an optimizing allocator which partitions a large semantic network and allocates each sub-semantic network onto a PE. This is because the throughput of IXM is greatly influenced according to the allocation algorithms; the communication pattern in IXM, the parallelism in marker propagation, and the conflicts on the pyramid-shaped network change considerably.

Associative memories can be implemented with a high integration density because of its regular cell structure. The authors have started the VLSI implementation of IXM[9]. Though associative memories of large capacity are not in practical use at present, the associative memory approach will lead to one of the most promising architectures for large knowledge based systems.

ACKNOWLEDGEMENT

The authors thank Dr. Hiroshi Kashiwagi for providing the opportunity to pursue this research.

REFERENCES

[1] J.R.Carbonell: "AI in CAI: An artificial intelligence approach to computer-assisted instruction", IEEE Trans. on Man-Machine Systems, MMS-11:190-202,1970.

[2] S.E.Fahlman: "Design sketch for a million NETL machine", Proc. of First Annual National Conf. on AI, 1980.

[3] K.Handa, T.Higuchi, A.Kokubu and T.Furuya: "Flexible Semantic Network for knowledage Representation", to appear in Journal of Information Processing Society of Japan.

[4] K.Handa, T.Higuchi, A.Kokubu and T.Furuya: "Semantic Memory System IX - Knowledge Representation in IXL", WGAI Rep. of IPSJ, Mar. 1985. (in Japanese)

[5] T.Higuchi,K.Handa,T.Furuya and A.Kokubu: "A Semantic Network Language Machine", Proc. of EUROMICRO, 1985.

[6] G.E.Hillis : "Connection machine", TR-646, MIT, 1981.

[7] D.I.Moldovan et al:"Semantic Network Array Processor",Univ. of Southern California, Tech. Rep. PPP-84-2,1984.

[8] J.B.Dennis : "Data Flow Schemas", Lecture note in computer science, Vol.15,1972.

[9] A.Kokubu et al: "Orthogonal Memory- A Step Toward Realization of large Capacity Associative Memory", Proc. of VLSI85, 1985.

METHODS FOR ACHIEVING INTEGRATED OPERATION IN A HIGH PERFORMANCE OPTICAL LOOP INTER-COMPUTER COMMUNICATIONS SYSTEM

Masahiro KURATA, Seishiro TSURUHO, Takafumi ISOGAWA, and Hisao NAKASHIMA

NTT Electrical Communications Laboratories
P.O. Box.8, YOKOSUKA POST OFFICE, KANAGAWA, JAPAN

Abstract

This paper describes high-performance optical loops for main-frame coupling and its application to complex, large-scale computer systems. Emphasis is placed on problems concerning alteration of the system configuration, system operation and rapid system recommencement in the event of break-down. A communications method using the optical loop and the "Global Manager" (GM) concept in the "System Control Processor" (SCP) are proposed. The rapid system recovery method using total integration via optical loops along with the GM concept are discussed and evaluated in terms of enhanced reliability.

1. Introduction

NTT has developed and furnished nation-wide data communications and processing systems for banking, national and municipal government and many other public services. [1] Enhancing the expandability of these large-scale systems became a crucial theme, because the rapid expansion of services and system scale, such as nation-wide information-base access, was needed with this movement toward on information society.

To achieve such large-scale systems, a method for loosely-coupled multi-processor construction is appropriate from the view points of performance and reliability. This, in turn, requires development of a high speed inter-processor communications method which allows for easy alteration of a multi-processor construction. Such development will simplify system operations and improve reliability.

In light of the above, high-performance optical loops, consisting of Processor-Processor Communications Interface equipment (PCI) and optical fiber cables, has been developed. A System Control Processor (SCP) has also been developed, which centralized and simplified the command and control functions in a loosely-coupled multi-processor construction. PCI and SCP development led to creation of the Denden Information Processing System (DIPS) Computer Complex with Optical Loops (CCOL) offered for nationwide commercial services. Even though the DIPS-CCOL system consists of a number of subsystems, operation is smooth. In the event of trouble in a given subsystem, the immediate switch-over of processing to another (standby) subsystem serves to enhance the reliability of the whole system.

This paper describes the characteristics of the optical loops used in the DIPS. Additionlly, it outlines the DIPS-CCOL and technology employed, and provides an evaluation of this highly reliable system.

2. High-performance Optical Loops for Coupling Processors

NTT's optical fiber loops have a token-passing ring consisting of PCIs and dual optical cables. The objectives involved in achieving a complete subsystem integration using optical fiber loops are as follows.

1) Reduction of the required CPU load for inter-subsystem communications processing
2) Simplification of subsystem addition and alteration
3) Enhancing optical loop reliability

Inter-subsystem Communication Processing

Multiple data-links for inter-subsystem communications are used in the optical loop communications method to achieve complete subsystem integration. The method to dynamically establish data-links for every message[2] is not suitable because of the excessive overhead in real-time processing systems and the complexity of a subchannel assignment mechanism. Consequently, a method for statically holding data-links was adopted. The data-link maintenance capability was developed using the PCI subchannels to enable the followings:

1) Maintaining the predetermined data-link control information between processors for each subchannel.
2) Control of data-link information by any processor in the PCI loop.

Figure 1 outlines an inter-subsystem communications processing method. To achieve a full duplex path, a logical path with two one-way data-links was constructed to avoid retranmission due to conflict and associated transmissions rights control. Another feature of this system is that, by permanently setting the data-receiving end of a data-link on Read Command, Attention Interruption requiring the reading of transmitted data can also be reduced.

CH2345-7/86/0000/1050$01.00 © 1986 IEEE

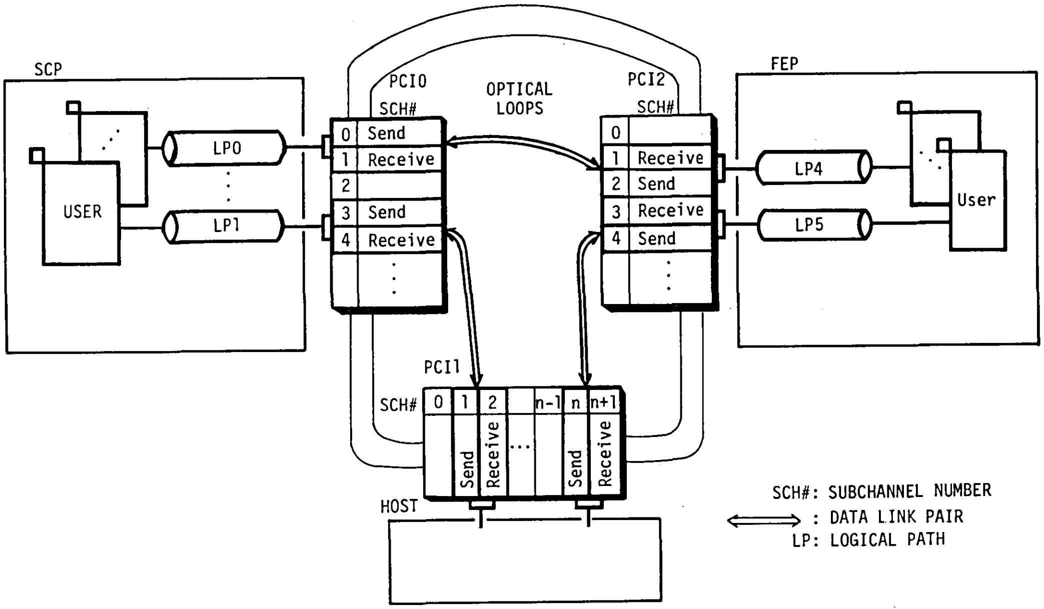


Figure 1. Inter-Sub-System Communications Processing

System Expansion

Information concerning configuration status, including the data-link control information of all PCIs, must be simultaneously altered with the addition of a new subsystem. A related difficulty concerns the method of load balancing to multiple PCIs.

The DIPS-CCOL Global Manager (GM) function maintains unified control of all data, including the configuration status information. It determines data-link control information for each PCI, and distributes this to each subsystem. The GM uses a specified subchannel pair as a "control data-link" in order to distribute information before establishing a data-link.

Two methods were considered to distribute data to multiple PCIs: dynamic and static routing. A logical path allocation method based on the latter was developed because of the necessity for a high-performance facility. In a real-time processing system required to satisfy specific response-time, the maximum subsystem loads must be smaller than the total PCI performance potential which is found in the linked subsystem. As a result, the following mehtod was proposed to determine the route automatically and simplify the equipment plan.

1) Logical path classification
2) Allocation of PCI data-links for each classification at initial use
3) Making uniform the number of opened paths belonging to the same class in all the PCIs of a given subsystem.
4) Making possible path allocate to any PCI by control of the initial use.

Implementation of the methods described above, as well as utilization of the developed components resulted in the achievement of high-performance, computer-to-computer communications to which extra subsystems could be easily added.

Enhancing the High Reliability of Optical Loops

Both optical cable and PCI breakdown may occur in optical loops, and this reflects badly on system reliability. Figure 2 illustrates an automatic loop back, using a dual path, which can be carried out by PCIs when such breakdown occur.[3] With PCI breakdowns, message processing can also be continued by the reallocation of other PCI data-links to the damaged logical paths. It should be noted that this is possible only when the subsystem has more than one PCI. Reserve PCIs, shared by the whole system, can also be set up and with automatic integration executed by the SCP. As a result, continued operation is possible despite optical loop breakdown, thereby enhancing the overall reliability of the system.

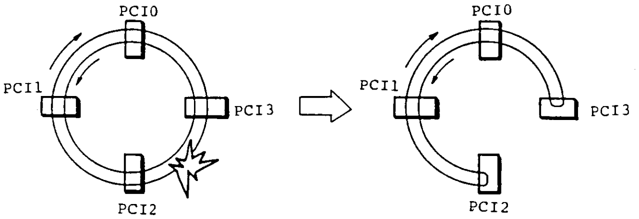


Figure 2. Safety mechanism in the event of Optical loop breakdown

3. An Outline of the DIPS-CCOL

The configuration of a highly reliable, DIPS-Computer Complex with Optical Loops is given in Fig.3. The special features of this system are that optical loops have been used to join the subsystems and that a System Control Processor (SCP) has been installed. It should be noted that use of the term subsystem, within the contex of the system outline, refers to the Host and Front End Processors (FEP).

SCP is the actualization of the Global Manager (GM) for governing operation of the whole system. It also supervises other subsystems and controls overall system structure. In addition to a database containing the knowledge required for system operation, the GM concept, at the core of the SCP, contains the functions enabling management and control of inter-subsystem links as well as the handling of console messages transmitted via the optical loop to other processors. Each subsystem has the ability both to send messages concerning the whole system to the SCP and to process messages from the SCP as console commands. The SCP exchages messages with each subsystem and carries out hardware reconfiguration based on the knowledgebase. Two SCPs can also be utilized, one on a standby basis, to create a redundant system structure offering high reliability.

As part of the attempt to ensure high reliability in the system, one "reserve" subsystem is installed for n subsystem units. As all subsystems share the magnetic storage units, the reserve unit can either remain on standby or execute other jobs.

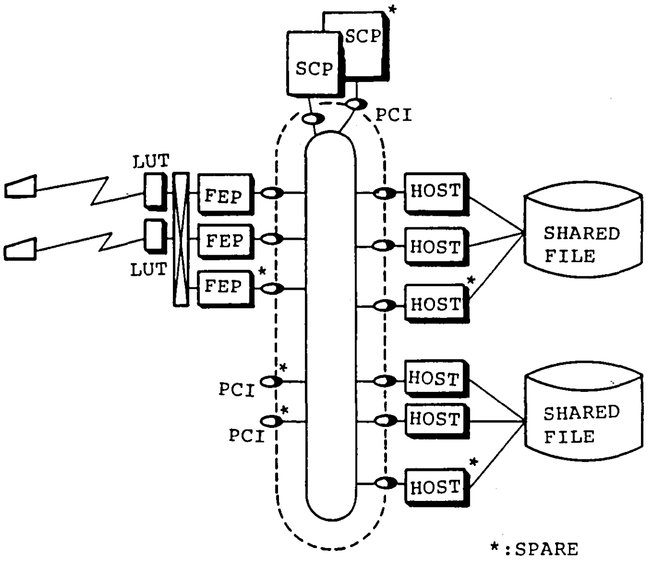


Figure 3. DIPS Computer Complex with Optical Loops

4. DIPS-CCOL Developmental Topics

The installation of extra hardware and modified software and the enhancement of system reliability have become major goals in order to provide increased system capability and more comprehensive user services. This is especially true in regard to the current trend toward a wider range of services in computer systems. One current proposal for achieving these goals concerns utilization of the distributed processing method, in which the system is constructed by a number of sybsystems.[4] [5], [6]

However, system alteration and operation is not so easily achievedwhen dealing with large-scale distributed processing systems which employ main frame computers. There are two main difficulties. The first lies in the greatly complicated interface between subsystems made necessary by the increase in overall system size. The second concerns the more complex operations which arise in conjunction with the increase in component numbers.

In light of these problems, technological development was chosen in two areas as means to achieve the goals mentioned above. The first area concentrated on Global Manager development for smooth operation of the overall system. The second sought continuous transaction processing capability at times of Host breakdown.

Global Management of the system

An increase in the number of subsystems not only causes an increase in the operations required to manage the system but also complicates the operations of the whole system because of associated complex operations for the joint use of peripheral devices between subsystems. The concept of a Global Manager for the SCP has been proposed as a mean of solving these problems. The main features of this Global Manager are:

1) Unified management and control of all subsystems. Status responsibility includes all device connections as well as execution of operations in response to messages.
2) Substitution for human operators as regards operating instructions.
3) Harmonious global system management in conjunction with operators.

Enhancement of System Reliability

The larger the system scale, the greater the effect of service suspension on society becomes. Consequently, the recommencement of service after a subsystem breakdown, before the user is even aware of any interruption, is most effective from the dual viewpoints of MTBF (Mean Time Between Failure) extension and MTTR (Mean Time To Repair) reduction. Through use of an optical-loop-based, complete integration, quick service recommencement can be achieved. In times of Host breakdown, this function puts messages on hold in the FEPs, switches over to a stand-by Host and then carries out high-speed file recovery.

5. Global Management

In a CCOL system, the following conditions must be satisfied in order to facilitate system operations:

1) Each subsystem can be individually controlled.
2) Inter-subsystem control can be either automatic or by a single operation.
3) Various system operating conditions can be achieved。

Thus, noting that system operation centers on console interation, the basic fuctions of the GM were identified as automatic console response and response procedure creation, as well as a console message transmission function.
Figure 4 diagrams the Global Manager of an entire system.

Automatic Console Response Function

A function which invokes procedures describing the automatic response process is the most primitive function in terms of achieving an automatic response. Two cases, either the arrival of a console message or a specified time, were selected as triggers for procedure invocation. The following functions were developed in order to simplify this definition.

1) Registration of both the procedure and the console message invoking it.
2) Registration of both the procedure and the time at which it is invoked.
3) Invoking the procedure when the required conditions are satisfied.

This function is designed to be present in both the SCP and each subsystem, and renders automatic response processing hierarchical and distributed.

Automatic Response Procedure Creation Function

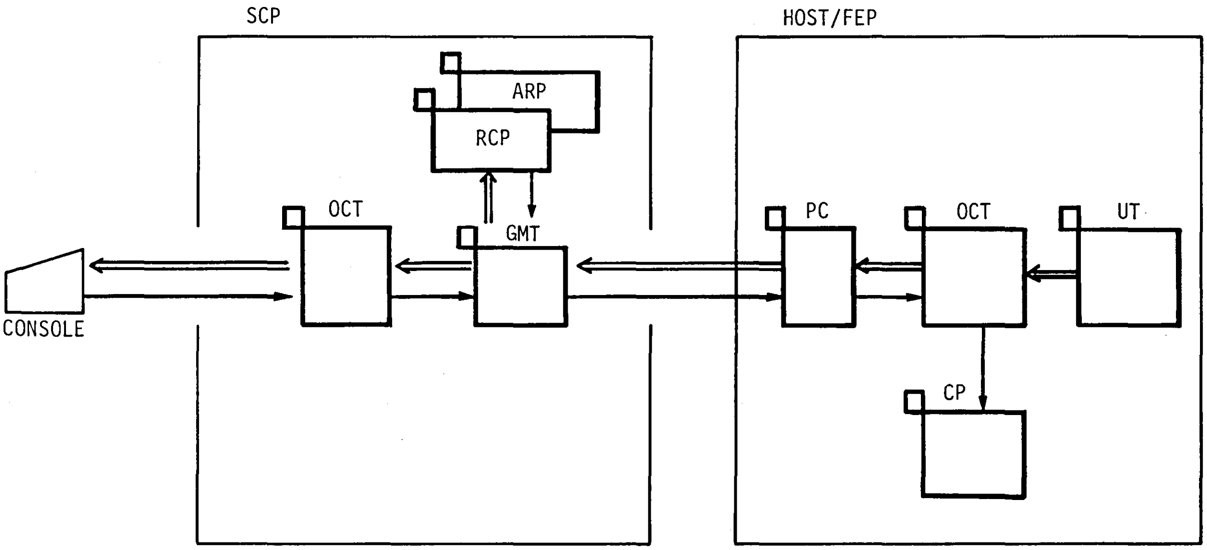
Because automatic response procedures vary from one system to another, a simple language function has been provided to describe the response process. This allows a new command definition due to the function's ability to input console commands from the procedure.

Console Message Transmission Function

Transmission of console messages to the GM has been achieved in order to permit exclusive resource control and processing synchronization spanning multiple subsystems. An example of these functions would be the necessity for simultaneous alteration of line control information in multiple subsystems which are sharing line switchs, when these line switchs are modified. In a case like this, the console message controlling the hardware switch can be transmitted to the SCP, from where a command affecting the reconfiguration of the whole system can be issued.

It is additionally necessary that reprocessing of console messages, given the possibility of various abnormalities, be done accurately and rapidly, for function reliability in commercial systems. In this regard the function is accompanied by the following.

1) Console message loss prevention
2) A check-point which makes the global parameters maintain these values at system restart.



⇐ : MESSAGE
← : COMMAND
ARP: AUTOMATIC RESPONSE PROCESSING
RCP: RECONFIGURATION PROCESSING
OCT: OPERATOR CONSOLE TASK
GMT: GLOBAL MANAGER TASK
PC: PROGRAMMED CONSOLE
CP: COMMAND PROCESSOR
UT: USER TASK

Figure 4. The Mechanism of Global Manager Processing

The system presented in this paper is superior to conventional system,[7] which employ a pseudo-console device, for the following reasons.

3) The alteration response in system operating conditions is simplified.
4) Because operations knowledge is distributed to the subsystems, autonomous operation by each subsystem is simplified. This feature also enables prevention of excessive loads being placed on the SCP.
5) Because console messages are transmitted by optical loops, transfer is rapid and does not require separate hardware for connections.

6. Rapid Operation Recommencement

System reliability can be greatly enhanced by making possible the rapid recommencement of processing, that is, before most users are even aware of trouble in their subsystems. This calls for early detection of any abnormality in subsystem operating conditions automatic reconfiguration, and increased speed in message recommencement processing on standby.

Early Detection of Abnormal Conditions in the Subsystem

This is of vital importance in the reduction of both user trouble and the need for recovery, but the problem lies in the incidence of no-response events in loops due to software bugs. In order to speed detection and increase its accuracy, a "health checks" was programed, to mutually supervise in both Host and FEP, addition to a "self check" hardware of whether a timer extension operation had been carried out within a specified time.

Automatic Reconfiguration

When the GM contained the SCP is notified of a malfunction, it invokes appropriate reconfiguration action according to the information received. When a malfunction occurs in the Host, the GM selects a substitute Host, checks that this substitite is on Final Standby and then commands it to take over operations. So that system designers can re-edit the reconfiguration procedures to match each individual system, they were given a hierarchical structure using an automatic response procedure creation function.

Continuous Reception of Messages

When it takes too much time to get the standby Host ready, the time supervisor at the users terminal cuts the circuits. If, however, the switch-over and Host recommencement are completed in a shorter time, and messages can be stored in the FEP, the user can remain completely unaware of the fact that the Host went down. The method chosen for this was to use an optical loop to preset the logical path between processors as an integrated configuration, and to execute a switch-over of message transmission merely by having the FEP's communication processing program alter the logical path.

Intial Setting of the Reserve System

To achieve faster processing of messages retransmitted from the FEP, the proposed design calls for the reserve Host to be on standby, having both joint use of the present file and also the ability to operate as a load sharing system. This particular Final Standby method eliminates the need for intial system setting during service recommencement processing.

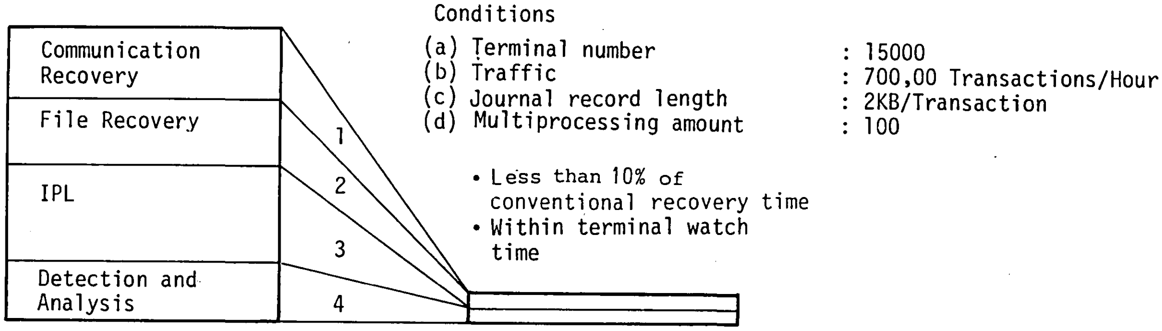
Quick System Recovery

During an on-line, system recommencement following system shutdown, both file restoration and transaction recovery are essential because file storing is accomplished by message processing. Thus, executing data restoration simultaneous with on-line services makes recovery time far shorter than that found in conventional methods. In the latter, services are recommenced only after the damaged transactions have been re-processed or abandoned following data recovery.

Services are able to be quickly recommenced in a file-sharing system because the files which need restoring are excluded by a suspended subsystem in use as a Serial Reusable Resource (SRR). However, extreme requests to the SRRs indicates that the services are barely recommenced. The effect of this conflict must be lessened, and this is done by retrieving the data under commencement from the journal file, prohibiting access to the damaged data in the file, and then releasing the SRR. In this way access is given to almost all files, making quick service recommencement possible. The data to which access was prohibited can be restored to present status with journal information. It then can be used in service.

Continuous job Execution when on Standby

If jobs that were being processed continue to be processed even when the system is on standby, there is a very real danger of a Host switch-over. This could lead to an excessive on-line system load. To avoid this hazard, a CPU has been utilized, along with a memory and I/O resource schedule. These allocate remaining resources according to high or low-priority jobs. This method suits the Final Standby method due to the fact that jobs to be executed that are currently on standby are usually processed in the background of real time operations.


Communication Recovery
File Recovery
IPL
Detection and Analysis
1
2
3
4
• Less than 10% of conventional recovery time
• Within terminal watch time

Conditions

(a) Terminal number	: 15000
(b) Traffic	: 700,00 Transactions/Hour
(c) Journal record length	: 2KB/Transaction
(d) Multiprocessing amount	: 100

Strategies for Rapid Recommencement

1. Lack of necessity for IPL processing due to FEP's independent operation.
2. No need for operator assistance in recovery processing and high-performance equipment.
3. Introduction of a Final Standby method, making IPL unnecessary for reserve switch-over
4. Avoidance of delayed detection and analysis through program supervision

Figure 5. Rapid Service Recommencement

7. Conclusion

In this paper, two major features of high-performance optical loops for inter-computer communications and the computer complex have been described. The first, global management, is constructed by centralizing the data-link control, moving the physical console interface from each subsystem and programing system operations. The second, rapid operation recommencement, enhances system reliability by switching FEP message transfer and actualizing the Final Standby method.

These technologies have made large-scale on-line systems far more reliabile than previously possible. In the system example shown in Fig.5, more than 95% of messages which would have been damaged in the past can now be continuously processed even under the maximum load condition. Its message abnormality results from excessive loads associated with recovery.

Acknowledgement

The authors would like to thank Mr. Arai and Mr. Murata, Information Processing Department, NTT Electrical Communication Laboratory, for thier valuable suggestions.

References

[1] T. Kawai, et al, "Large Scale Data Communication System for Nation-wide Banking Activities and Development of its Software, "ICCC-78, 1978.

[2] Katsuo IKEDA, et al. "Computer Network coupled by 100 MBPS Optical Fiber Ring Bus, "COMPCON 80 FALL pp.159-165

[3] J. J. Wolf, et al. "Design of a Distributed Fault Tolerant Loop Network, "ISFTC, June 1979, pp.17-24

[4] Hiroaki IKUTA, et al. "Super minicomputer Complex System, "I.P.S. of Japan, Vol.23, No.1, Jan. 1982

[5] P.H. Enslow, Jr., et al. "What is a Distributed Data Processing System ?" COMPUTER, Jan. 1978, pp.13-21

[6] J. F. Bartiett, "A NonStop Operating System, "Proc. Hawaii International Conference on System Sience, 1978, pp.103-117

[7] Itujiro ARITA, "An application of the intelligent console to the operation management system for a computer center, " I.P.S. of Japan, Vol.23, No.5 Sep. 1982, pp.472-479

Autonomous Decentralized Software Structure and Its Application

Kinji Mori*, Hirokazu Ihara*, Yasuo Suzuki*, Katsumi Kawano*,
Minoru Koizumi*, Masayuki Orimo*, Kozo Nakai** and Hiroaki Nakanishi**

* Systems Development Laboratory, Hitachi, Ltd.
1099 Ohzenji, Asao-ku, Kawasaki 215, Japan
** Ohmika Works, Hitachi, Ltd.
5-2-1 Ohmika, Hitachi, Ibaraki 319-12, Japan

Abstract

An autonomous decentralized software structure has been developed to achieve software on-line expansion and on-line maintenance and fault-tolerance. In this structure, a software subsystem installed in each of the distributed computers has an autonomous operating function for selfmanagement and coordination with the subsystems without external direction and/or execution. The data field(DF) concept is introduced in order to realize autonomy in each software subsystem. Every data is broadcasted, with the attached content code corresponding to its meaning, into the DF without specifying the receiver. Each module in the software subsystem is connected only to the DF and judges whether to receive the data, or not, from the DF on the basis of its attached content code. The module runs independently from all the other modules after receiving all the necessary data from the DF. This autonomous data-driven mechanism and DF structure ensure that an application software module can be independently produced, loaded through the DF, tested by the data in the DF and begin execution while the other modules are operating. Every subsystem checks the consistency of the received data. Hence, fault tolerance of the software system in event of the software subsystem's failure is attained. Evidence of the effectiveness of this software system is provided by its application to real-time control systems for steel production process control.

1. Introduction

Distributed computer systems have developed rapidly based on the cost reduction of microelectronics and the advancement of communication techniques. On the other hand, the relative cost of software has increased in comparison with the hardware cost. Hence in the distributed system, production, expansion, maintainance and fault tolerance of the application software itself rather than economical sharing of the hardware resources have become important.

Recently distributed operating systems have been discussed but most of them are experimental systems (1). Some of them are network operating systems and others are only extensions, which manage to communicate and share resources and are added to the existing operating system. HYDRA (2) and Medusa (3), the distributed operating systems of the Carnegie-Mellon University C.mmp and Cm* multimicroprocessor systems, are aimed at not only efficient resource sharing but also system reliability. COCANET UNIX (4) developed at the University of California, Berkeley, a network operating system, modifies UNIX so that the existing programs can share remote resources using the standard UNIX interprocess communication mechanism. A language concept, communication port, is proposed to support communication between modules in distributed processors(5), but this concept is limited to one-to-one communication. This communication relation cannot be changed during system operation. CONIC, a language and support system, has been developed for dynamic configuration of distributed systems. However, in this system a specific configuration manager exsists to translate requests to change the system, expressed in CONIC language(6).

In these conventional operating systems and languages, every software subsystem or specific manager is assumed to know the partial and/or total system structure represented by the communication relation between software subsystems. But in a large and complex system in which the structure frequently changes and partially fails and/or recovers, this assumption can not be easily satisfied and it becomes difficult to manage the system in a centralized manner.

Therefore Autonomous Decentralized Software Structure is proposed here to meet the requirements of on-line expansion, on-line maintenance and fault tolerance of the software system. This software structure has the feature that every software subsystem has autonomy to manage itself and coordinate with the other software subsystems. Coordination is attained by communicating with the other subsystems through a proposed data field(DF), in which the data circulates and the software subsystem selects whether to receive the data on the basis of its content. In this system, there exists no specific manager and no master/slave relation among the software subsystems. This software system structure is proposed on the basis of the Autonomous Decentralized System Concept (7)-(9). This concept has already been realized in the hardware systems of a local area network : ADL and a multi-microprocessor system : FMPA and these hardware systems have operated successfully (7)-(9).

2. Design Concept - Autonomous Decentralized System Concept

2.1 Software Features and System Requirements

Previously, computing system hardware has been expensive in comparison with software. The software structure has been mainly designed under the constraints

CH2345-7/86/0000/1056$01.00 © 1986 IEEE

of the predetermined hardware structure. But the recent advancement of LSI technology and the reduced cost of LSI have gradually made the software structure design flexible and substantially reducing the dependency on the hardware structure. Moreover there have been increasing demands for the software system to satisfy the following objectives.

(1) On-Line Expansion

As system size increases, step by step construction is required. Open-ended system construction during partial operation should be made possible. Even after completion of the construction, the system may be improved. Hence on-line expandability of not only hardware but also software is required.

(2) On-Line Maintenance

In the large system, the frequency of fault occurence somewhere in the system increases. It should be ensured that the maintenance and the test procedures can be carried out without suspending system operation, especially in the case of on-line and real-time systems.

(3) Fault-Tolerance

The reliability of the hardware has been sufficiently improved in comparison with the software. But even if the software includes some bugs, the system is not required to stop its entire operation to prevent the fault.

(4) Performance

Of course, high system performance is needed. For attaining high performance, reducing the peak of the computer load and making the load smooth by improving the software processing mechanisms are required.

All these criteria should be met in the construction and operation of large-scale systems. However the conventional centralized and distributed systems are developed from standpoint that the total system must be previously determined. Thus the hardware and software structures are fixed and have little flexibility. It is therefore difficult to satisfy these requirements.

2.2 Autonomous Decentralized System Concept

The Autonomous Decentralized System Concept has been proposed to attain on-line expandability, on-line maintainability and fault tolerance of not only the hardware but also the software system.

This concept, based on biological analogy, has the perspective that a system almost always has faulty parts and undergoes modifications. That is, a total system cannot be previously defined. A system is defined as the result of the integration of subsystems. A system is called an autonomous decentralized system, if the following two properties are satisfied.

(1) Autonomous Controllability

If any subsystem fails, the other subsystems can continue to manage themselves.

(2) Autonomous Coordinability

If any subsystem fails, the other subsystems can coordinate their individual objectives among themselves.

In these properties, only subsystem failure is considered. But they can be applied to subsystem addition and repair. Hence, these properties assure fault tolerance, on-line expandability and on-line maintainability of the system. They suggest that every subsystem requires an intelligence to manage itself and to coordinate with the other subsystems. The software subsystem in the subsystem is called the Atom. The Atom consists of not only the application software but also its own management system software. This system software in each Atom is called the ACP (Autonomous Control Processor).

To realize an autonomous decentralized software system with autonomous controllability and autonomous coordinability, each ACP is required to satisfy the following three conditions.

(1) Uniformity

Each ACP must be uniform. That is, every ACP functions are self-contained. The ACP can manage its own application software modules without being dependent on the other ACPs functions. This condition means that even if the ACPs in some Atoms stop their operation, the other Atoms can contiune their operation.

(2) Equality

Each ACP must be equal. Hence, every ACP can manage its own application software modules without being directed by or giving directions to the other ACPs. That is, there is never a master-slave relation among the ACPs. This condition assures that each ACP can continue its operation even if the other ACPs fail.

(3) Locality

Each ACP must be able to manage its own Atom and to coordinate with the other Atoms based only on local information. This condition means that every ACP can continue its operation even if the ACP cannot collect the global information because of the other subsystems' failures. The locality condition excludes a common global information file.

It is difficult to satisfy these three conditions under the conventional software structure and its management techniques. Hence the Autonomous Decentralized Software System has been proposed.

3. Software Structure

3.1 Data Field

The basic feature introduced in the Autonomous Decentralized Software Structure is the Data Field (DF) where the data circulates among the modules in the Atom and moreover among the Atoms. The DF in the Atom is called the Atom Data Field (ADF). In the DF, each data has a content code which indicates its meaning.

This feature means that the software module broadcasts all data with its content code into the DF and it judges whether or not to receive the data from the DF on the basis of its content code. Here not every software module uses the destination address of the data. The message format is shown in Fig. 3.1.

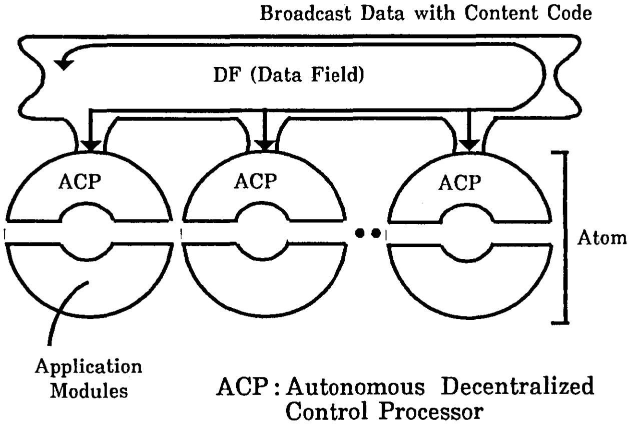
In the DF concept, there is no distinction regarding broadcast among the data, the parameter, the file and the program. All of them, attaching their corresponding content codes, are broadcasted in the DF in the same message format. Moreover no message in the DF has priority. Each Atom independently judges what data, parameters, programs in the DF to accept and how to process them on the basis of their attached cotent codes. Moreover each Atom broadcasts every processed result data, parameter, file or produced program into the DF without knowing how to process them or which modules to receive.

F	CC	SA	C	Data	CRC	F

F : Flag
CC : Content Code
SA : Sender Address
C : Control Code
CRC : Cyclic Redundancy Check

Fig. 3.1 Message Format Including Content Code

In this software structure, every Atom is connected to the DF (Fig. 3.2). This feature indicates the autonomous data-driven mechanism of the module in the sense that no module ever drives the others nor directs them to receive and process the data. This mechanism makes the modules loosely coupled. No module controls the others but each independently controls itself. Moreover any module can communicate with any other module connected to the DF. The module can collect all information necessary to coordinate with the others according to its situation since all information is broadcasted without specifying the destination modules.



ACP : Autonomous Decentralized Control Processor

Fig. 3.2 Autonomous Decentralized Software Structure

This communication protocol using the content code makes it possible for every Atom to control itself and coordinate with the others without having global information on the entire system but having local information on the content codes necessary for the modules in the Atom itself. The content code protocol is not designed for the sender and receiver modules, but for the data itself.

For example, physically, the DF among the Atoms corresponds to a computer network. Each Atom is installed in one computer unit. For example, the ADF corresponds to FIFO memory which is divided into every software module and can be locally accessed only by its module (Fig. 3.3).

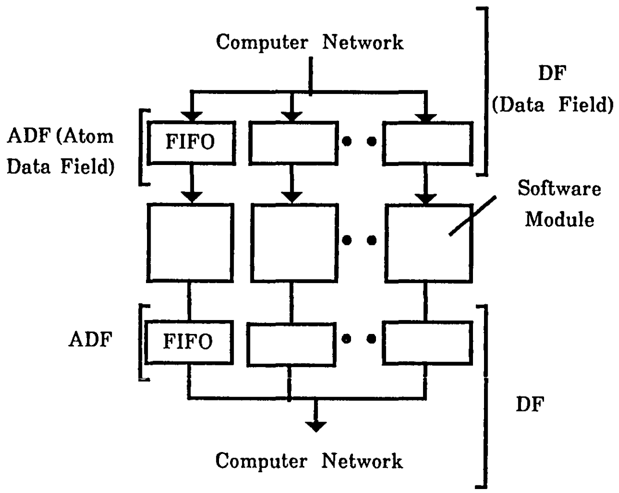


Fig. 3.3 Data Flow in Software Subsystem (Atom)

3.2 Modular Software Structure in System Software and Application Software

Not only the autonomous data-driven mechanism but also the modular structure of the system software : ACP and application software are derived from the DF concept. That is, every software module in the ACP and the application software is a unit of function which receives data from the DF and sends out data into the DF (Fig. 3.4).

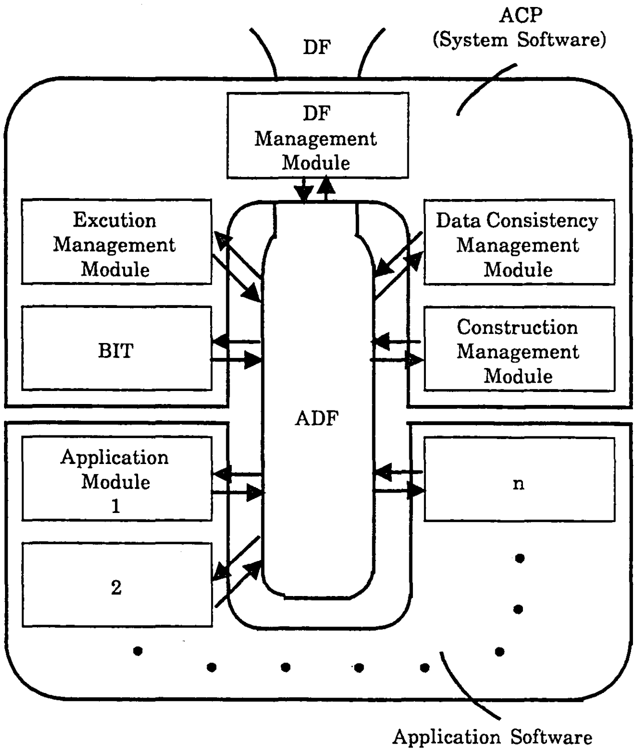

DF
ACP
(System Software)
DF Management Module
Excution Management Module
Data Consistency Management Module
BIT
Construction Management Module
ADF
Application Module 1
n
2
Application Software

Fig. 3.4 Modular Software Structure in System and Application Software

Each ACP has the functions of managing the data flow route, checking the data, supporting the test and diagnosis and so on. The function of the application software module is characterized by the relation between the content codes of the input data and those of the output data.

Each application software module can be installed in any distributed computer without knowing where the other application software modules are installed. This is because all modules anywhere are connected only to the DF and driven by the data in the DF. Moreover the application software module can be moved from computer to computer without informing the others of its movement. Hence, portability of the application software modules is attained.

The ACP is installed in every computer.

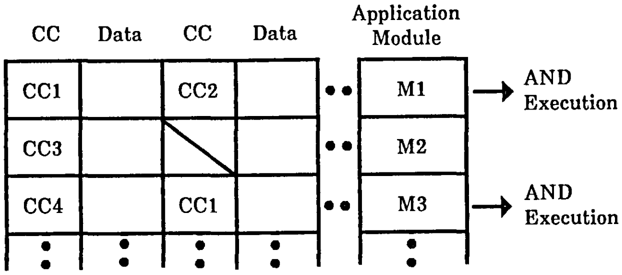
4. ACP Functions

4.1 Data-Driven Mechanism

The autonomous data-driven mechanism of the application software modules is realized by the following two management modules of the ACP.

(1) DF management

The DF management module acts as the interface between the DF and the ADF. The ADF includes the table of the relationship between the application software module in the Atom and the content codes necessary to execute this application software module (Fig. 4.1). According to these registered content codes, the DF management module receives the data from the DF and stores it in the corresponding area in the ADF. The data originating within the Atom is broadcasted into the DF by this management module.



CC	Data	CC	Data		Application Module	
CC1		CC2		••	M1	→ AND Execution
CC3		╲		••	M2	
CC4		CC1		••	M3	→ AND Execution
⋮	⋮	⋮	⋮		⋮	

CC : Content Code

Fig. 4.1 ADF Format

The same data may be used by the different application software modules in one Atom and then it is stored in all the corresponding areas in the ADF.

The DF management module may be installed in a separate communication I/O processer to improve computer performance.

(2) Execution Management

The execution management module monitors the ADF. As soon as all the data necessary to the application software module is received in the ADF by the DF management module, the execution management module drives the application software module. The number of data necessary to execute the application software module is one or more (Fig. 4.1). With this execution management module, the application software modules run asynchronously and freely. This autonomous execution property ensures that the application software module cannot be directed to execute by any other application software module. In addition it ensures that the application software module can continue its operation even in the event of fault occurence, expansion and maintenance of the other application software modules.

4.2 Data Consistency

The asynchronous and free run mechanism of the application software modules makes it possible to replicate the same application software module to be stored in the Atoms and to independently execute these software modules. The number of replications of the module can be arbitrarily settled according to the fault tolerance level required. The replicated application software module runs independently and sends the processed result data into the DF. But some of them may be faulty because of the failures of their corresponding ADF memory and the computers themselves.

Thus, the first problem is for each ACP to check the replicated data received from the DF and select the correct one.

The second problem arises in the case of "AND execution condition", which means that an application software module needs more than two data to execute (Fig. 4.1). The application software modules independently process and they send their result data into the DF. Hence these multiple data are asynchronously received from the DF by the Atom. The second problem is for each ACP to logically synchronize these multiple data which are asynchronously received from the DF for the AND execution condition.

The data consistency management function for the first problem is to select the correct data from among "the same data", which are sent out from the replicated application software modules. A threshold-voting technique is proposed. The data consistency management module identifies "the same data" not only by the content code but also the event number attached to the data. The event number is originally set at the module that received the information from an external source via input devices such as sensors and terminals. Although the data is successively processed by the modules, the original event number is preserved. "The same data" with the same content code and the event number is collected from the DF within a predetermined interval or up to a predetermined number. The correct data is selected from among "the same data" through the majority voting logic which can be flexibly adapted corresponding to the total number of received data or the minimum time interval. By this logic, the fault occurence is detected and each application software module can avoid being affected by fault propagation. This data consistency management module in the Atom can detect a fault that occured in the Atom itself by making a comparison with data that originated from the Atom itself and the other Atoms. The thresholds of the predetermined interval and number for the majority voting logic are applied to the systems aiming at high responsibility and reliability.

The second problem resulting from the "AND execution condition" is resolved by using the event number. The data consistency management module selects all of the multiple data with the content codes satisfying the "AND execution condition" of the application software module and it arranges the data in the order of the event number set by the original module. With this arrangement mechanism, the data consistency management module can detect whether all of the necessary multiple data have been completely received and whether some data is missing.

Through data consistency management, every Atom can independently check the correctness of the data and logically synchronize multiple data that originated from the other Atoms. Hence, in data consistency the autonomous controllability and coordinability properties are satisfied.

This data consistency mechanism to assure fault tolerance is easily adopted to system reconfiguration while the system is operating.

4.3 Test and Maintenance

The autonomous execution mechanism of the Atom and the DF concept make it easy for the application software modules to be debugged and tested while the system is operating. This on-line maintenance is supported by the BIT(Bilt-In Tester module) in each ACP and by the EXT(External Tester module) as the application software.

The BIT in each ACP has the functions of setting the application software module into the test mode, generating test data, and checking test results independently of the other modules.

After the installation of the application software module into the Atom and the recovery from the fault of the application software module in the Atom, the BIT in the Atom determines to begin the start-up test for this application software module and sets the application software module into the test mode. The application software module being in the test mode receives only test data with the attached test flag from the DF and processes them. It broadcasts the test result data attaching the test flag into the DF. However, it is prohibited to output the signal to output devices such as controllers. The EXT monitors the test data and the test result data in the DF. By collating the test data with the test result data, the EXT can check for fault occurence in the application software module in the test mode and broadcasts the fault detection. The BIT independently decides whether to start the operation of the application software module on the basis of the test result by the BIT itself and/or its check by the EXT.

The test data can be generated by the EXT or the BIT itself. The test data for the start-up test is previously stored in the BIT. The EXT can successively generate the test data according to the fault occurence situation in the system.

It is possible to drive the application software module while it is in the normal mode by the test data. This module broadcasts the test result data into the DF, but it prohibits sending the signal to the output devices. The test result data is successively used to test the other application software modules. With this mechanism, the application software modules can be successively tested while they are operating. The EXT monitors the test data and the test result data successively output from the application software modules and it detects any failed application software module in this test process.

The BIT independently tests the application software modules. The EXT does not direct the other modules to test and diagnose them. Hence the autonomous controllability and coordinability properties in test and fault diagnosis are obtained.

4.4 Program Loading and Expansion

In the DF, the program as well as the data circulates with its corresponding content code. The application software module including the relation between the content codes of the input and the output data is produced on the conventional OS in the software development subsystem and by using the conventional language. This produced application software module is broadcasted with its content code. The construction management module in the ACP, in which the content codes of the necessary application software modules have been previously registered, judges whether or not to receive the application software module from the DF on the basis of the registered content codes. This management module stores the application software module into the program area, makes the table of the relation between the content codes of the input and the output data for the application software in the ADF and sets the application software module into the test mode for the start-up test.

The ADF corresponding to the application software module is generated whenever this module is loaded in the computer. The construction management module does not need to know about the expansion and the replacement of the application software modules in the other Atoms . That is, the construction management module can independently install the application software module into the Atom without interrupting the other Atoms. Hence, the autonomous controllability and coordinability properties in software construction are attained.

5. System Design Criteria

In the autonomous decentralized system, the data originating within the computer is broadcasted at the time of its generation. The data is not stored in the memory of its originating computer or in the global common memory. Therefore, the module need not access the data in the memory of the other subsystems and the global common memory. The application software module can use the data received from the DF as needed without any time lag or concern for the condition of other subsystems.

The computer in this system executes the modules whenever the necessary data for the modules originates within itself or from the other computers. The computer is not directed to execute at any specific time by the others. Thus, the performance of the system is substantially improved by the reduction of the peak load of the computer. In this system, the computer load is normally high and almost always smooth. That is, in the autonomous decentralized system design, lower performance computers is sufficient compared with those required by the conventional system.

6. Application

One of the applications of the autonomous decentralized software system is a steel production process control system. As an example, the cold mill process control system is explained below (Fig. 6.1).

The computerization of steel production process control has already reached an advanced level. Recently the requirement for integrating the steel production subprocesses, which have been separately computerized, in order to reduce the stocks between the production subprocesses and improve the steel quality has gradually

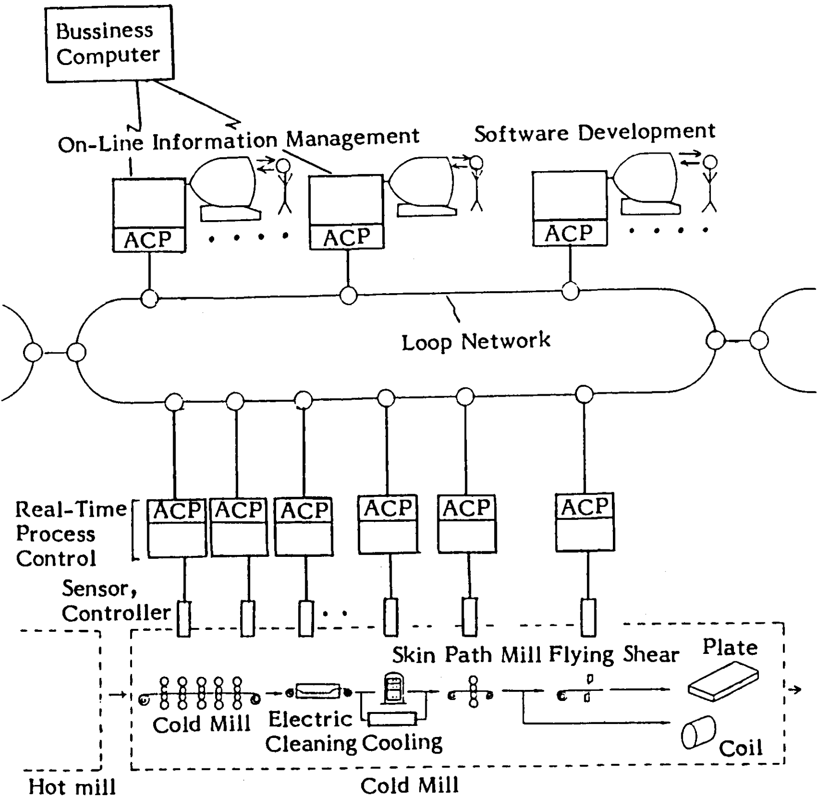


Fig. 6.1 Steel Production Process Control System

increased. Hence, the on-line expandability, on-line maintainability and fault tolerance of not only the hardware system but also the software system have become indispensable.

In this system, there are three major subsystems for real-time process control, on-line information management and software development. The functions in each major subsystem are distributed into several mini-computers HIDIC V90/5 series. These computers are connected by the loop local area network ADL(Autonomous Decentralized Loop Network) using optical fiber cable(8).

In the software development subsystem computers, the application software modules are produced, partially revised, and remotely loaded into the computers for real-time process control and on-line information management.

The main application software modules in the on-line information management subsystem are the steel production scheduling, the process data logging, the linkage to the large-scale business computer system, the EXT and the man-machine communication modules. The steel production scheduling module makes the detailed schedule for the production process on the basis of the rough production schedule transmitted via the linkage module from the business computer. This scheduling module dynamically revises it according to the current condition of the process. The condition is recognized from the current process data broadcasted from the real-time process control subsystems into the DF.

The production schedule data broadcasted from the steel production scheduling module is recevied by the real-time process control subsystem. The real-time process control subsystem consists of the cold mill process, the electric cleaning process, the cooling process, the skin path mill process, the flying shear process and so on. One or more computers are distributed to every process and each computer has its own responsible control region. The real-time process control software subsystem installed in each computer includes the application software modules for tracking and controling the iron slabs in its own control region. The tracking module receives the signal upon the detection of the iron slab from the detector in its own control region. This module identifies the iron slab by consulting the production schedule for this region and then generates the tracking information.

This tracking information is broadcasted from this module and is received at the controlling module in the same real-time process control subsystem. It is also received by the tracking module in its adjacent subsystems on the downstream process, the process data logging module and the man-machine communication module. The control module driven by the tracking information controls the apparatus for processing the iron slab in its own control region. The tracking information received by the adjacent subsystem is used to check the production schedule and, if necessary, to revise its own production schedule. The process data logging module driven by the tracking information edits

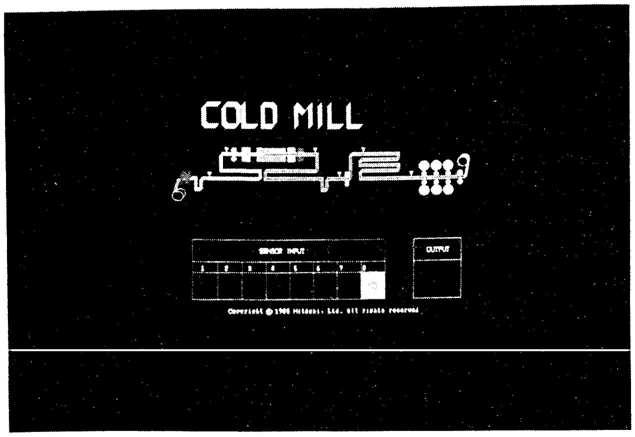


Fig. 6.2 Display of Tracking Information on Cold Mill process

and stores this information. The man-machine communication module is driven by the tracking information to display that information (Fig. 6.2).

The EXT is driven by the test data and the test result data, and it can detect faults in the software modules.

In this system, these application software modules are replicated according to their level of importance. This system has been gradually expanded function by function in the same process. Expansion has continued process by process in the cold mill process control system and from the cold mill process control system to the upperstream and downstream process control systems.

As the byproduct of development of this autonomous decentralized steel production process control system, software productivity has been much improved. In this system development, only the protocol of the content codes is previously determined. Then each application software module can be designed in a closed form being independent from the others. Each module is independently produced, loaded, tested, begins operation and is modified. Even when the system expands module by module, the existing system need not be modified. Therefore, in particular, the load for each software module debug and for total software module linkage tests decreases.

7. Conclusions

Autonomous Decentralized Software Structure aimed at software on-line expansion and on-line maintenance together with fault tolerance has been developed. Each software subsystem consisting of the system software ACP and the application software has autonomy through the ACP functions and the data field(DF) concept. Every data with the attached content code corresponding to its meaning is broadcasted. The ACP has the functions of receiving the data from the DF on the basis of its content code and checking the correctness of the received data. The application software module is driven only by the correct and necessary data received by the ACP. The process result data is also broadcasted with its content code. This autonomous data-driven mechanism makes it possible to test and repair the application software module during the operation of the other modules since the module can be independently set in the test mode. The program as well as the data is broadcasted into the DF together with its content code. Each application software module is produced by the software production subsystem and can be remotely loaded into one or more computers while the other application software modules are operating.

The autonomous decentralized software structure has been applied to the steel production process control system and so on, and its validity has been verified.

Acknowledgements

We would like to thank Mr. Hiroshi Kuwahara, general manager of Ohmika Works, Hitachi Ltd. for his continuous support and many helpful suggestions with this development. We are also grateful to Dr. Takeo Miura, board director, Group Exective of Computer Group, Hitachi Ltd. and Dr. Jun Kawasaki, general manager of Systems Development Laboratory, Hitachi Ltd. for giving us the opportunity for this research.

References

(1) J. A. Stankovic, "A Perspective on Distributed Computer Systems," IEEE Trans. on Comput., vol.C-33, no.12, 1102-1115, Dec. 1984

(2) W. Wulf, et al., "HYDRA : The Kernel of Multiprocessor Operating System," Communication of the ACM, vol.17, no.6, 337-345, June 1974

(3) J. K. Ousterhout, et al., "Medusa : An Experiment in Distributed Operating System Structure," Communication of the ACM, vol.23, no.2, 92-105, Feb.1980

(4) L. A. Rowe and K. P. Birman, "A Local Network Based on the Unix Operating System," IEEE Trans. on Soft. Eng., vol.SE-8, no.2, 137-146, March 1982

(5) T. W. Mao and R. T. Yeh, "Communication Port : A Language Concept for Concurrent Programing," IEEE Trans. on Soft. Eng., vol.SE-6, no.2, 194-204, March 1980

(6) J. Kramer and J. Magee, "Dynamic Configuration for Distributed Systems," IEEE Trans. on Soft. Eng., vol.SE-11, no.4, 424-436, Apr. 1985

(7) H. Ihara and K. Mori, "Autonomous Decentralized Computer Control Systems," IEEE Computer, vol.17, no.8, 57-66, Aug. 1984

(8) K. Mori and H. Ihara, "Autonomous Decentralized Loop Network," COMPCON Spring '82, 192-195, 1982

(9) K. Mori, et al., "On-Line Maintenance in Autonomous Decentralized Loop Network : ADL," COMPCON Fall '84, 323-332, 1984

APPROACHES TO AN INTEGRATED OFFICE ENVIRONMENT

MAKOTO YOSHIDA, MAKOTO KOTERA, KYOKO YOKOYAMA, SADAYUKI HIKITA

Computer Systems R&D Division, OKI Electric Industry Co.,Ltd.
11-22 Shibaura 4-chome, Minato-ku, Tokyo 108, Japan.

ABSTRACT

There has been a rapid improvement in the technology of local networks and office equipment. Unfortunately, our current understanding of office equipment is only for a single machine, operated in a separate 'stand alone' environment.

Nowdays, it is required to have a methodology for extending these environments over many machines without being hampered by any new problems of integrating remote processes. This paper describes our approaches to integrate the office environment, in which several pieces of equipment such as database servers, file servers, personal computers, workstations, and minicomputers, jointly work through a local area network. The transaction concept is exploited and defined as the network-wide transaction to integrate various environments. The resources integrated in this paper are the file server, the database server, and several application programs. New efficient implementation techniques for network-wide transaction processing are proposed.

1. INTRODUCTION

The development of network technology allows us to use several distributed resources. Distributed database systems make it possible to access remotely located databases[11,14]. The same benefits arise in distributed file systems[2,4,5,7,8]. The development of office automation equipment on local networks has also increased the demand for using Remote Procedure Calls for distributed applications[3]. It is the tendency of a distributed system to treat everything attached to the network as a network resource[13].

In a highly integrated environment, the key point is the effective use of network resources. One defines the integration as "glue"[1]. The resources may be servers, application programs, personal computers, workstations, mini-computers and so on. Taking into account the integration of these resources, we are developing the "integrated information network system", as is shown in Fig. 1.

In this paper, only the "integrated LAN system" is focused on and discussed, in which a file server, a database server and dispersed application programs, those connected by local networks, are treated as network resources[12]. The rationales and some techniques applied for integration are described.

This paper describes the integration constraints imposed in an integrated office environment, and some techniques to solve these constraints. Some novel integration techniques of several servers and the dispersed application programs through a local network are presented. In section 2, the transaction in an integrated environment is modeled and defined. In section 3, implementation techniques for the transaction are described. Section 4 describes the experimental development of multi-bit-map oriented file server system, in which the proposed extended transaction model is implemented.

2. THE ENVIRONMENT

2.1 TRANSACTION

A transaction is a unit of atomic actions, and the transaction itself constitutes an atomic action. An atomic action has the property of either being completely done or nothing being done. Also, the atomic action must inherit the above property, called atomicity, even in cases of failure or concurrent execution[9,14]. This means the transaction is closely related to recovery and concurrency control. We explicitly define the network-wide transaction's behavior as follows.

CH2345-7/86/0000/1064$01.00 © 1986 IEEE

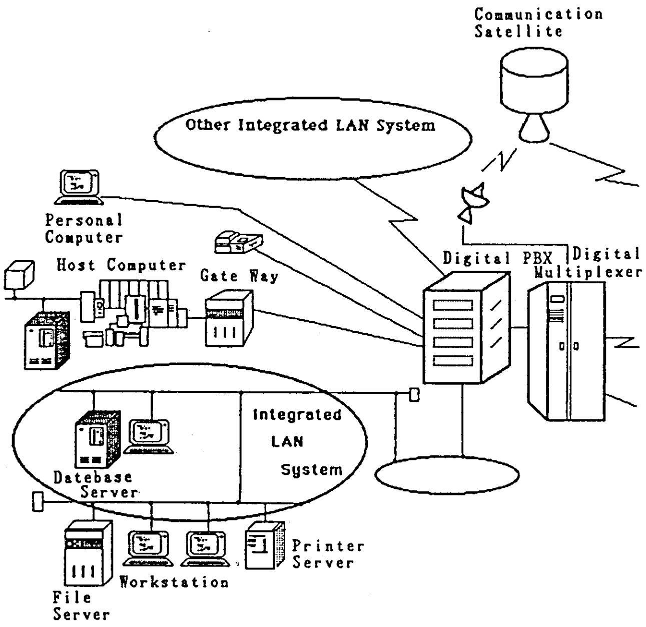


Figure1. Integrated Information Network System

(i) **ATOMICITY** --- In all cases, including node failure, the atomicity of a transaction must be guaranteed.

(ii) **CONSISTENCY**--The transaction must be processed consistently. Namely, atomic actions defined in the transaction must have serial execution on it's order without affecting it's environmental factors such as communication delay or existing concurrency control . Communication delay or concurrency control must not cause atomic actions within a transaction to be executed other than sequentially.

To maintain the above constraints in an integrated environment, we introduce the traditional transaction concept to the application itself. Introducing the transaction concept to the application program is equivalent to defining n-party transaction control. Any resources accessed by the application are integrated into one transaction. This transaction is called the "network-wide transaction". One application might consist of several transactions, in which all of these are affiliated with the network-wide transaction.

2.2 APPLICATION ENVIRONMENT

The construction of an integrated system must be based on either of two types of external environments . One environment bases the integrated environment on previous existing systems, and is called the bottom up approach. In this environment, the restrictions on existing systems, such as imposed on transaction processing and concurrency control, must be inherited. The other environment does not have any restrictions on it's construction, and this is called the top down approach. Efficiency is the whole purpose of the top down approach.

Our approach assumes the bottom up approach in an office environment. For components in our integrated environment, a file server, a database server, and several dispersed application programs, which are distributed throughout the network, are assumed[12]. Each server is basically a traditional transaction oriented system, and can only perform transaction processing over it's own resources[9].

Our step-wise approaches toward the integrated system are as follows. The first step focuses the integration of one server and several dispersed applications(see Fig.2-1). N co-related application programs and one transaction based server, all connected by a local network, are integrated as a network-wide transaction. The second step expands the first step to several servers(see Fig. 2-2).

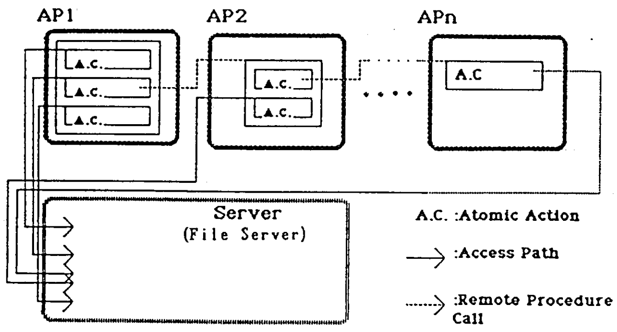


Figure2-1 Step1 : Single Server - Multiple Applications

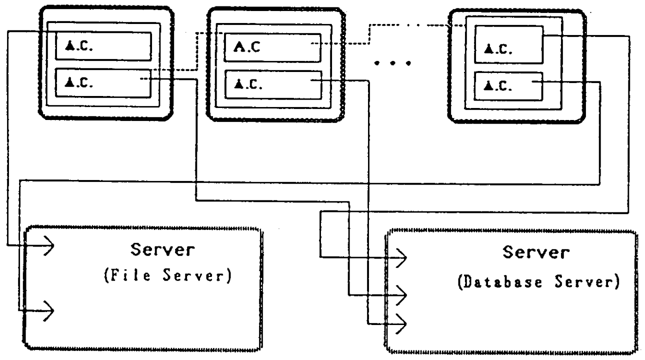


Figure2-2. Step2:Multiple Servers -Multiple Applications

For the construction of an integrated system in our environment based on the network-wide transaction, the following are assumed.

- Concerning the application program, the serialized execution of atomic actions must be guaranteed. (The implementation being based on the process model makes this assumption reasonable.)
- Each server is a transaction based closed system in terms of accessing a single kind of resource.

2.3 PROBLEM ANALYSIS

There are a lot of problems to be solved in constructing an integrated system based on the bottom up approach. One problem is affiliated with the concurrency control. In a closed environment, as the transaction is defined in it's single environment, concurrency control is inevitably affected by the life time of the transaction. There has been proposed many concurrency control techniques[10]. These algorithms usually assume that concurrently accessed data is inaccessible to other transactions until the end of transaction. At commit time, every resource accessed in the transaction is released for other transactions to access. From the view of the bottom up approach, this causes the problem of inaccessibility of the shared data from the network-wide transaction. In a traditional transaction system, transactions are prevented from interfering with one another if they access shared data concurrently. This leads to the deadlock phenomenon of the network-wide transaction.

The other problem is commitment control. The data committed by one of the members of a transaction, which belongs to a network-wide transaction, can never be rollbacked from other members.

The rationale of solving the above problem is to propagate the committed data of each transaction to the network-wide transaction. In a later section, we call the techniques which solve the above problems the "nested commitment control".

For the integration of several existing transaction oriented systems into one integrated transaction system, we insist that the following problems must be resolved.

- Communication delay should be transparent to the application program. The serialization of atomic actions in a transaction must be guaranteed across the network.
- Different transactions might be grouped together into one transaction without conflicts, in which the data may be shared.

3 IMPLEMENTATION TECHNIQUES

3.1 APPROACHES TO THE NETWORK-WIDE TRANSACTION

Two types of models are given to support the network-wide transaction in a server, which we call the "multi-bit-map oriented model". One is the model which inherits the accessed data from the previous transaction, and makes the committed data accessible(see Fig.3). This model makes possible the recovery of inherited data in case of failure. The other is the model which makes the non-shared data for cooperative transactions in a network-wide transaction to be safely committed(see Fig.4). Both models make the data accessed by the transactions recoverable to the consistent state one at a time.

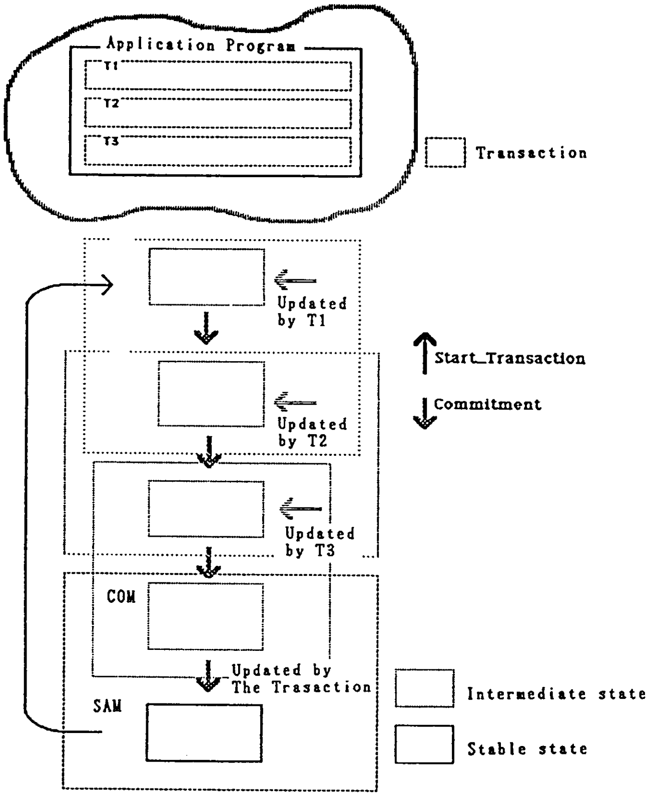


Figure 3. Nested Commitment Model for Shared Data

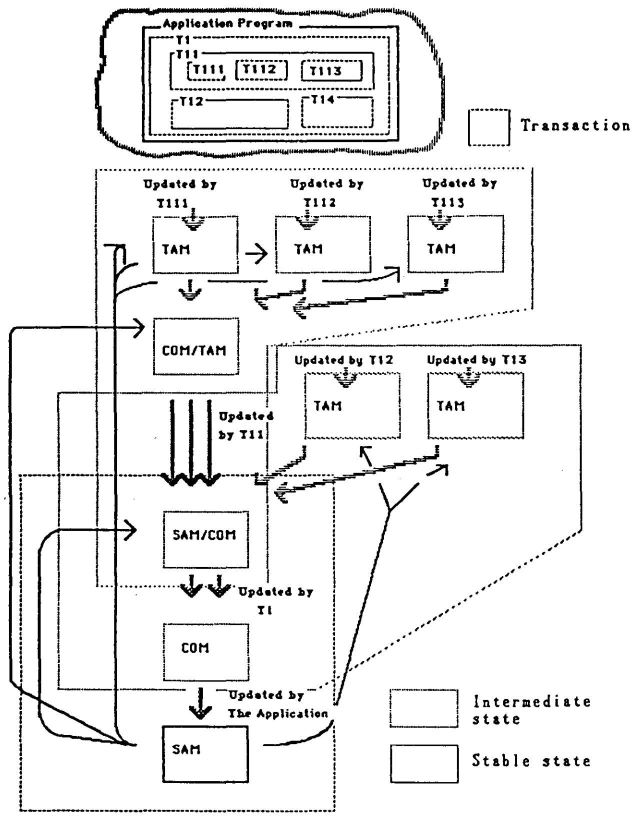


Figure 4. Nested Commitment Model for Non-Shared Data

3.1.1 Rules For Commitment

The following regulates the rules of the models. For integrated applications, either of these two models are applied on demand depending upon the ordered sequences of atomic actions in the server.

A) Rules For Shared Data In Nested Commitment For Network-wide Transaction

<Assumption>

1) The transaction's commit requests in a network-wide transaction must be ordered as presented by the application program, when these arrived at the server.

2) There exists some mechanisms which identify the network-wide transaction.

<Rules For Commitment>

When one of the members of the network-wide transaction commits the updated data, these data are inheritated to the next transaction step by step using the intermediate table until the network-wide transaction is committed(see Fig.3). The intermediate table is sequentially propagated to the next intermediate table from transaction to transaction.

B) Rules For Non-Shared Data Access For Network-wide Transaction

<Assumption>

1) Member transactions of a network-wide transaction may commit at any time during the life time of the network-wide transaction.

2) Member transactions of a network-wide transaction access data which are not shared by the same group of transactions in the network-wide transaction.

<Rules For Commitment>

The data accessed by the transactions in a network-wide transaction would not be committed until all of the members of the network-wide transaction commit. Iff all of the members of the network-wide transaction commit, the data committed might be propagated to the next group of transactions, in which the same analogy of the shared data in nested commitment is applied to the propagated data(see Fig.4).

In the above, two types of commitment control for the network-wide transaction were described. In a real environment, these two types of commitment controls are combined depending upon the serialized execution of the application program. In either case, all of the above mentioned properties are maintained.

The advantages of multi-bit-map oriented commitment control are summarized as follows.

- Transactions grouped in a network-wide transaction may commit or rollback at any time during the network-wide transaction's lifetime.
- Shared data can be accessed without interfering with other members of the transaction.

3.2 MULTI-BIT-MAP ORIENTED COMMITMENT CONTROL ALGORITHM

The algorithm which uses the multi-bit-map based boolean operations for commitment control is presented.

The basic multi-bit-map algorithm, indicated in Fig.5, uses three tables. The first COM(Common Table) indicates the intermediate state, and can be referenced from all transactions. The SAM(Stable Table) represents the committed area of the COM

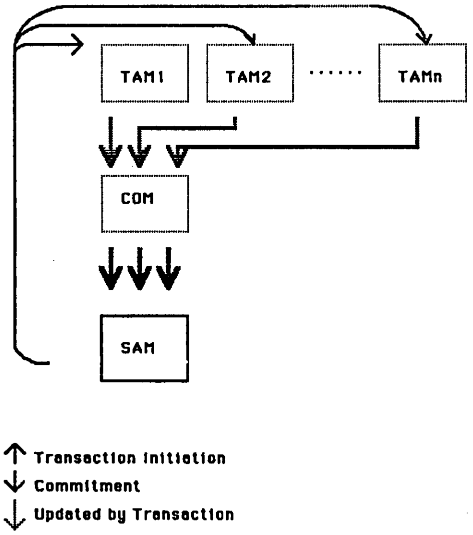


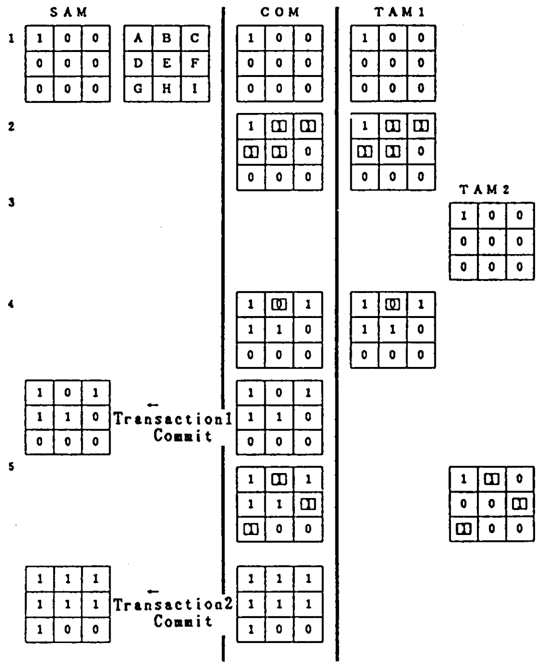
Figure5. Basic Multi-Bit-Map oriented Commitment Model

table. The TAM(Transaction Table) is defined for each transaction and indicates the transaction states. Several boolean operations on these three tables are defined, as illustrated in Table 1. TAM is created from SAM whenever the new transaction emerges(see Table 1-1). When the transaction is committed, the operation shown in Table 1-2 is performed and SAM is updated and the disk is also updated.

Fig.6 displays one example performed using these operations. In this figure, it is assumed that there are only nine areas, and one area has already been committed. The stepwise explanation is as follows.

When transaction 1 is initiated, TAM1 is generated by copying SAM (1). If transaction 1 acquires four data areas, the COM table is searched and the empty areas are assigned (2). After this, if transaction 2 is initiated, TAM2 is generated from SAM(3). If transaction 1 releases one area and commits, that area is assigned to empty areas and the operations in Table 1-2 are performed, and the area indicated by SAM table are updated(4). After this if transaction 2 acquires three areas, COM is searched again, and empty areas are assigned (5). If transaction 2 commits after this, the operations in Table 1-2 are performed and the areas indicated by SAM table are permanented(5).

This algorithm guarantees the freedom from failure in transaction processing.



A~I···storage address of pre-allocated
(1)···Initiation of Transaction1
(2)···4 areas request by Transaction1
(3)···Initiation of Transaction2
(4)···Release request by Transaction1(B:data area)
(5)···3 areas request by Transaction2

Figure6. Example

Table1. BIT MAP operations

Table1-1

SAM	COM	TAM
0	0	0
0	1	0
1	0	1
1	1	1

Table1-2

SAM	TAM	COM	SAM
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table1···Transaction Initiation

Table2···Trasaction Commitment

The multi-bit-map can be extended to the nested commitment by replacing the SAM to COM of the above algorithm. By replacing stable state with common state, nested commitment is easily supported. Stable state can be replaced with the common state propagatelly according to the depth of nest, as offered. And only one lowest stable table is affected by the committed action. The correspondence of this algorithm with the conceptual model is shown in Fig.4.

4 FILE SERVER SYSTEM

The experimental development of a multi-bit-map oriented file server system which reflects the nested commitment processing is described.

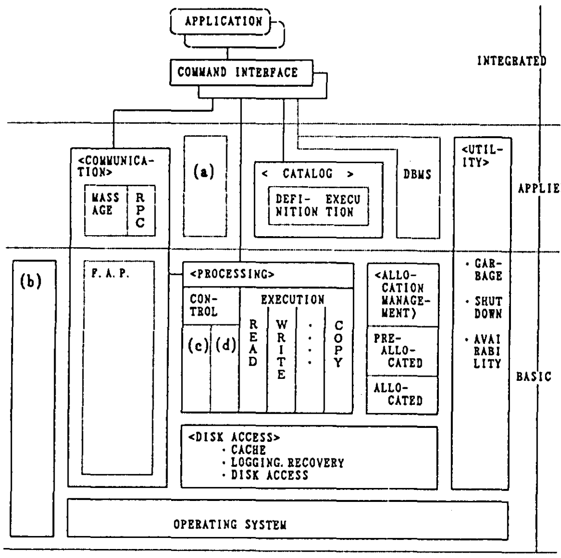
4.1 OVERVIEW OF THE FILE SERVER SYSTEM

We developed the network-based file server system with the following in mind,

- be able to store and access a large amount of data whose size is variable.
- include basic functions for shared data.
- support mechanisms for an integrated office environment.

Based on the requirements above, we constructed a three layered system model. It consists of a basic layer, an applied layer, and an integrated layer. Fig.7 indicates the system model designed.

The basic layer includes the functions which are commonly required for general applications. Included functions are concurrency control, commitment control, and directory services. The applied layer includes those functions which are specific to special applications. It also includes general functions which can be constructed by utilizing the basic layer. Functions include remote procedure call or message transmission. The integrated layer integrates the office environment utilizing the basic functions and applied functions. The explanation of each sub-systems in the model, and techniques applied to each one is shown in Table 2.



(a) ···Distributed Processing
(b) ···Data Communication
(c) ···Concurrency
(d) ···Commitment

Figure7. System Model

Table-2 SYSTEM'S CONFIGURATION

LAYER	SUBSYSTEM NAME		FUNCTION	TECHNIQUES DEVELOPED
INTEGRATED	COMMAND INTERFACE		Query interpretation	High Level Language embeded interface
APPLIED	CATALOG PROCEDURE		Command file definition and its execution	Command definition language
	COMMUNI-CATION	R P C	Remote Procedure Call of Catalog File	
		MESSAGE	Message transmission among application processes	SEND. RECEIVE
	:			
BASIC	PROC-ESSING	CONTROL	Transaction Processing (Commitment & concurrency)	Multi-Bit-Map based commitment control Extended hashing to Concurrency Control applied
		EXECUTION	Execution of each command	
	ALLO-CATION MANAGE-MENT	PREALLO-CATED	Virtual allocation management by pre-allocating staic data	Multi-Bit-Map Control
		ALLOCATION	Actual disk allocation and deallocation	Applied Buddy System for efficient continuous allocation management
	DISK ACCESS		Actual disk access and atomicity of each call	atomicity of each call
	COMMUNICATION		Protocol processing of F. A P	F. A. P

4.2 OVERVIEW OF THE FUNCTIONS

- INTERFACE

Two interfaces for the basic layer and two interfaces for the applied layer are supported. As for the basic layer interface, file access protocol which can be used from remotely located applications residing on the same node as the file server are supported. As for the applied layer interface, the remote procedure call and synchronous message transmission among applications are supported for integration.

- TRANSACTION

A transaction is supported by the specific command, START-TR and TR-END. By enclosing access commands with those commands, atomicity of a transaction is guaranteed. The method used for commitment is shadow management of file directory, in which the multi-bit-map technique is applied.

- REMOTE PROCEDURE CALL AND MESSAGE TRANSMISSION

The system supports a synchronized message transmission mechanism using SEND and RECEIVE commands that are used as a tool for integrated processing management in an office environment. These commands are implemented on the UD(user defined)command when transmitted to the network. The remote procedure call is an extension of message transmission.

The remote procedure call is supported as an applied layer command. To provide n-party conversation, and to support distributed applications, synchronous message transmission among applications, using SEND and RECEIVE commands, is supported.

5 CONCLUSION

This paper described the approaches to an integrated office environment .

Several cooperating application programs which use traditional atomic transaction systems, such as file servers and database servers, are operated consistently within a local network. The atomicity and the serialization problem of network-wide transactions are solved by introducing the transaction concept on the application itself, and by introducing the multi-bit-map techniques to the servers. The proposed multi-bit-map oriented commitment control solved the problem of concurrency control in a traditional atomic transaction system. It made possible the sharing of data from several transactions.

The experimental development of a multi-bit-map oriented file server system was also described to verify the correctness of our model. The current system is now operating in our company.

REFERENCE

[1] J. Donahue, "Integration Mechanisms in Ceder," SIGPLAN Notices, vol.20, no.7, 1985.

[2] M. R. Brown, K. N. Kolling, and E. A. Taft, "The Alpine File System," ACM Trans. on Computer Systems, vol.3, no.4, Nov.1985.

[3] A. D. Birrell, B. J. Nelson, "Implementing Remote Procedure Call," ACM Trans. on Computer Systems, vol.2, no.1, Feb.1984

[4] L. Svobodva, "File Servers for Network-Based Distributed Systems," ACM Computing Surveys, vol.16, no.4,1984

[5] H. Stugis, J. Mitchell, and J. Israel, "Issues in the Design and Use of a Distributed File Systems," ACM SIGOPS Oper. Syst. Rev, 14, 3, July, 1980

[6] J. L. Peterson, T. A. Norman, "Buddy Systems," CACM, vol.20, no.6, June, 1977

[7] L. Lampson, "Atomic Transaction, in Distributed File Systems: Principles and Design, " 10th. Operating Systems Principles, Dec.1985.

[8] M. Satyanarayanan, J. H. Howard, etc, "The ITC Distributed File Systems: Principles and Design," 10th. Operating Systems Principles, Dec.1985.

[9] J. Gray, "The transaction concepts: Virtues and Limitations," VLDB, Sep.1981.

[10] P. A. Bernstain, N. Goodman, "Approaches to concurrency control in distributed Database," NCC 1979.

[11] M. Yoshida, K. Yokoyama, etc., "Time and Cost Evaluation Schemes of Multiple Copies for Data in Distributed Database Systems," Trans. on Software Engineering, vol SE-11, no.9, September, 1985.

[12] S. Hikita, S. Kawakami, K. Sano, "Extended Functions of the Database Machine FREND for Interactive Systems," IEEE-1984 Workshop on Visual Languages, Dec.1984.

[13] J. H. Morres, M. Satyanarayanan, etc, "ANDREW: A Distributed Personal Computing Environment," CACM, MAR.1986, vol.29, Num.3.

[14] S. Ceri, G. Pelagatti, "Distributed Databases: Principles and Systems," McGraw-Hill,1984.

OPERATING SYSTEMS AND DATA BASE ARENA

Operating Systems

TRACK CHAIR: Dr. James Peterson
MCC

Distributed Operating Systems

TRACK CHAIR: Dr. Jack Stankovic
Carnegie Mellon University

Data Bases

TRACK CHAIR: Dr. Anil Nigam
IBM T. J. Watson Research Center

USE OF PETRI NET INVARIANTS
TO DETECT STATIC DEADLOCKS IN ADA PROGRAMS

B. Shenker, T. Murata, S.M. Shatz

Department of EECS
University of Illinois at Chicago
Chicago, Ill. 60680

ABSTRACT

This paper presents a method for detecting static deadlocks in Ada tasking programs using structural and dynamic analysis of Petri nets. Automatic translation of the Ada programs into Petri nets which preserve control flow and message flow properties is described. Properties of these Petri nets are discussed and algorithms are given to analyze the nets to obtain information about static deadlocks that can occur in the original programs. Petri net invariants are used by the algorithms to reduce the time and space complexities associated with dynamic Petri net analysis (i.e., reachability graph generation).

1. INTRODUCTION

Over the past few years, a few techniques have been proposed for use in validation of Ada tasking programs. Taylor [1] has presented a general static analysis technique for Ada tasking and has shown that such analysis has an exponential time complexity (in terms of the number of tasks). German [2] has described an approach for detecting deadlocks in Ada tasking programs using dynamic analysis. Since the approach uses dynamic analysis, results are dependent on the supporting environment, especially on the scheduler characteristics. In this paper, we propose the use of a Petri net invariant method for detecting static deadlocks in Ada tasking programs. Our technique reduces to some extent the time and space complexities associated with the general state enumeration method used by Taylor.

Our deadlock analysis technique starts with translation of Ada tasking programs into Petri net models [3] and uses structural and dynamic analyses of these specific Petri nets, which we call Ada nets. Ada nets are abstract models of the source programs since an Ada net only models the Ada source program's control flow and message flow. As such, only those Ada statements which can alter the control flow (such as If, Loop, and Select statements) or which constitute a rendezvous (Entry calls and Accepts) contribute to the Petri net translation. Further details on the translation of Ada programs into Ada nets will be given in Section 2.

Unlike the previously reported techniques of Shatz and Cheng [4], which are concerned with both deadlock and general tasking analysis, our method is not entirely based on generation of the complete Ada net's reachability graph. Instead, we have been able to combine the specific properties of Ada nets with the Petri net concepts of place and transition invariants in order to reduce the complexity associated with generating the complete reachability graph.

Since the intuitive concepts and definitions of Petri nets are well known and to save space, we will follow the terminology and notations of Petri nets as defined in [5], unless otherwise stated. Since the concepts of Petri net invariants are not so widely known and since they play a key role in our subsequent discussion, we briefly discuss them now. In particular, we define an <u>S-invariant</u> (<u>T-invariant</u>) as an integer solution y (x) of the homogeneous equation,

$Ay = 0$ ($A^T x = 0$), where A is the transition-to-place incidence matrix of a Petri net [5]. The subset of places (transitions) corresponding to nonzero entries of an m-vector y (an n-vector x) is called the <u>support</u> and is denoted by $||y||$ ($||x||$), where m is the number of places and n is the number of transitions in a Petri net. A support is said to be <u>minimal</u> if no proper nonempty subset of the support is another support. An S-invariant (T-invariant) whose support is minimal is called a <u>minimal S-invariant</u> (<u>T-invariant</u>). We use only non-negative invariants.

2. OVERVIEW OF ADA TRANSLATION BY EXAMPLE

Automated translation from Ada programs into Petri net models produces what we call Ada nets. The process of Ada net generation can be simply described as follows. Whenever the right-hand-side of any Ada syntactic construct is recognized during parsing, the Ada actions associated with the construct are invoked. These actions generate a part of the Ada net - the part that models the given construct. Control then passes to the syntactic parent of the translated construct. The translation process continues until the root of the parse tree is reached. At that point, the input Ada program is then either recognized or rejected. If it is recognized, the Ada net corresponding to the program will have been success-

CH2345-7/86/0000/1072$01.00 © 1986 IEEE

fully constructed. The parts of the net that are generated are linked together into an Ada net that preserves the behavior of the input program. Consider the following fragment from an Ada program (statement numbers are for later reference only):

```
TASK BODY task1 IS          TASK task2 IS
1)  WHILE cond1 LOOP        0)  ENTRY entry2;
2)  task2.entry2;           END;
3)  END LOOP;               TASK BODY task2 IS
END;  -- task1              4)  ACCEPT entry2;
                            END;  -- task2
```

The Ada translation maps the program fragment into the Ada subnet illustrated in Fig. 1a. The dashed places, transitions and arcs represent connections to the parts of the program not shown. Place p6 represents a Loop statement. The Entry call statement 2 is represented by t3, t4, p5, p2, p3, and p4. The Accept statement 4 is represented by t5, t6, p7, p2, p3, and p4. Places p2, p3, and p4 represent the communication channel of entry2 in task2. In general, every entry in any Ada program is represented by three unique places (p_{mep}, p_{send}, p_{ack}) in the corresponding Ada net. Thus, for this example, the three places for entry "entry2" are p2, p3, and p4 respectively. Place p2 is initially marked and belongs to the set of mutual exclusion places. Such a place is referred to as p_{mep}. Execution of statement 2 is represented by firing transition t3. This firing is possible only if p_{mep} of entry 2 (place p2) is marked. When p2 is not marked, some Entry call of entry2 must be currently involved in a rendezvous with an Accept statement of entry2. When t3 successfully fires, places p3 and p5 become marked. This submarking of the Ada subnet represents the situation when the Entry call (statement 2) is trying to rendezvous with an Accept statement of entry2. When the Accept statement (statement 4) is ready to rendezvous, t5 fires and removes the token from p3. Note that if there exists more than one Accept statement for entry2, then p3 serves as an input place to multiple transitions. The place p3 is referred to as a sending place (p_{send}). Firing t6 represents the control of task2 exiting the Accept statement. After the Accept statement ends, place p4 enables transition t4. This means that the control of task1 may exit from the suspended state (due to the Entry call). Place p4 is referred to as an acknowledging place (p_{ack}). When t4 fires, p2 becomes marked. This means that no Entry call is now rendezvousing with any Accept statement of entry2.

The details of translation from an Ada program into an Ada net are fairly complicated and require syntactic analysis as well as some semantic analysis (for example, to distinguish package function calls from task entry calls). The major technique is to generate pieces of the net during each production rule reduction during parsing. We briefly describe the idea on the previous example. Since it is assumed that Ada task specifications precede task bodies, statement 0 is reduced first. Three communication places (p2, p3, and p4) for entry2 in task2 are generated, as illustrated in Fig. 1b. Statement 2 is reduced next. It causes t3, t4, and p5 to be generated and linked with the communication places of entry2, as illustrated in Fig. 1c. The dashed places and arcs are not yet generated. The reduction of the Loop statement in task1 creates the Ada subnet illustrated in Fig. 1d. Place p6 is substituted for both dashed places in Fig. 1c. Finally, statement 4 is reduced. Two new transitions, t5 and t6, are linked to the places p3 and p4 of entry2. The Ada subnet for the program fragment is then complete and is illustrated in Fig. 1a. A detailed discussion of the general Ada translation technique used here is found in [3]. Another approach to automated translation of Ada tasks into Petri net equivalent models is discussed in [4].

3. PROPERTIES OF ADA NETS

3.1. Definitions and Characteristics

Definition: A process-subnet is a state machine that models one of the processes (Ada tasks) in a distributed environment when communication between processes is ignored.

Fig. 2a illustrates two separate process-subnets. Note that each begins with a place and ends with a place. At translation time, we introduce two transitions into the Ada net - t1 (par-begin transition) and t2 (par-end transition) in order to give a cycle in the Petri net graph. These two transitions are connected through a place p1 (cycle place). In Fig. 2b, we have the two process-subnets of Fig. 2a structured as a distributed program with no communication. However, most useful distributed programs are characterized by some communication between the processes. In Ada, processes communicate through communication statements (i.e., Accept statements and Entry call statements). As discussed in Section 2, every communication statement is represented by two transitions and at least four places. Fig. 2c illustrates the distributed program of Fig. 2b with some communication.

A transition that represents a communication statement is referred to as an AE transition (Accept or Entry transition). Since every communication statement is represented by exactly two AE transitions in an Ada net, we will refer to one of them as "first" (e.g., t5 in Fig. 2c) and the other one as "last" (e.g., t6 in Fig. 2c), based on the order of their occurrences in a transition firing sequence starting with t1. A place in a process-subnet that is either an output place of some first AE transition and/or an input place of some last AE transition is referred to as an AE place (e.g., place p7 in Fig. 2c). The places generated by the translation of an entry specification are referred to as communication places (e.g., p2, p3, p4 in Fig. 2c). They are not considered to be a part of any process-subnet. All non-communication places are referred to as sequential places. Any arc connecting an AE transition and a communication place is referred to as a communication arc. A communication arc is not considered to be part of any process-subnet. All non-communication arcs are referred to as sequential arcs. Three communication places, p_{mep}, p_{send}, and p_{ack} (p2, p3, and p4 in Fig. 2c, respectively), are required for token (information) exchange between two or more process-subnets. The places may be thought of as a communication channel between potential "senders" and "receivers". The mutual exclusion place p_{mep} guarantees that only one process-subnet may send a token at any time to the receiving process-subnet. No other process-subnet will send a token to that receiver until the current sender is acknowledged by p_{ack}.

Definition: An Ada net is a strongly connected Petri net, APN, composed of process-subnets possibly interconnected with communications arcs and places. The strong connectiveness of APN is provided by p1 (cycle place), t1 (par-begin transition), and t2 (par-end transition).

3.2. Safeness of Ada Nets

The initial marking M_0 for an Ada net represents the situation when a program modeled by the net is ready to start executing. The marking M_0 consists of places p1 and every p_{mep} initially marked with one token. No other places have tokens initially. M_0 for the Petri net in Fig. 2c is $(1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)^T$.

Theorem 1: An Ada net APN < P, T > is safe for M_0, i.e. $M(p) \leq 1$ for each place p in P and for any marking M reachable from the initial marking M_0.

Proof: Consider first an Ada net APN<P, T> consisting of n process-subnets with no communication involved. Since each process-subnet is a state machine, the sequential places of each process-subnet together with the cycle place p1 constitute the support of an S-invariant having exactly one token. There are n such S-invariants, and each place p in P belongs to one of these n S-invariant supports. Since the token content of each S-invariant support remains the same and is one for any marking M reachable from M_0, we have $M(p) \leq 1$ for each place p in P and for marking M reachable from M_0. Next, when there are communications between process-subnets of APN, by the construction of an Ada net, there are S-invariants whose supports consists of communication places, p_{mep}, p_{send}, p_{ack}, and some sequential places. Each of the supports of these S-invariants contains exactly one token (initially in p_{mep}). Thus every place p in P is in the support of at least one S-invariant having one token. Therefore, $M(p) \leq 1$ for each p in APN and for any marking M reachable from M_0. Thus, APN is safe for M_0.
q.e.d.

4. DETECTION OF STATIC DEADLOCKS IN ADA PROGRAMS

There is no commonly accepted definition of a static deadlock in a distributed programming environment. Traditional deadlock detection techniques (reachability tree or state graph analysis), using exhaustive search, label every potentially reachable deadlock as a "static deadlock". However, such techniques can generate some extra information. For example, a variable index loop containing communication statements always creates a potentially reachable deadlock. Since the number of times the loop executes is determined at execution time, any number of the loop iterations is assumed possible in this paper. Therefore, here we do not consider such a deadlock to be a static deadlock. Since Ada nets do not distinguish between the loops with variable and constant indices, potentially reachable deadlocks caused by Loop statements would not be reported by our static deadlock detection system. We would report any other deadlocks that can be predicted by static analysis of a program's source code. In the following sections, we define two classes of static deadlocks: inconsistency deadlocks and circular deadlocks.

4.1. Inconsistency Deadlocks

Some potentially reachable Ada program static deadlocks can be easily detected using only structural analysis of Ada nets. We call these deadlocks inconsistency deadlocks. In this section, we present an algorithm to detect such deadlocks.

An Ada program is called static executable when its corresponding Ada net has t2 (par-end transition) potentially enabled for M_0. We will refer to a set of linearly-independent T-invariants x of an Ada net such that $x(t) \geq 0$ for each t in the net and $x(t1) = x(t2) = 1$, where t1 is the par-begin transition and t2 is the par-end transition, as a set of Ada T-invariants of the net.

Theorem 2: A necessary condition for an Ada program Ap to be static executable is that its corresponding Ada net APN < P, T> has at least one Ada T-invariant.

The proof is obvious since there is always an Ada T-invariant in APN corresponding to the static execution path of Ap.

By Theorem 2, if an Ada net APN <P, T> modeling an Ada program Ap does not have an Ada T-invariant, then Ap is not static executable. Moreover, if some transition does not belong to the support of the linear combination of all Ada T-invariants of

APN, then the statement of Ap represented by the transition does not belong to any successful execution path. Such a situation indicates that Ap has at least one static deadlock, caused by a communication statement (or a group of communication statements). For example consider the following program fragment:

```
  task1:                      task2:

task2.entry2;          IF cond1
task2.entry1;            THEN ACCEPT entry2;
                         ELSE ACCEPT entry1;
```

The net corresponding to this Ada program would not have Ada T-invariants. Thus, the program is not static executable (it has a static deadlock) as can be easily checked. Note that removal of some communication statements (e.g., "task2.entry 1" and "ACCEPT entry1") could make the fragment static executable, since then every statement in the modified program would belong to at least one static execution path. Then, the support of a linear combination x' of all Ada T-invariants of the Ada net (for the modified program) would cover every transition in the net. Since the detection of these deadlocks depends on a test for T-invariants, we refer to them as inconsistency deadlocks. Now we present an algorithm that detects inconsistency deadlocks in an Ada program Ap (with its corresponding Ada net APN) and reports the minimal number of communication statements that need to be removed in order to remove the deadlock potential.

```
ALGORITHM INCONSISTENCY_DETECT

Input:
A - the incidence matrix of the net APN;
k - the number of communication
    statements in the program Ap;
S - the set of all possible subsets of
    communication statements in Ap.

Output:
The set of communication statements
"causing" inconsistency deadlocks

Method:
1) LOOP1: For i = 0 to k do
2) LOOP2: For each element s of S such
   that |s| = i do
3) A' := A;
4) Update A' to reflect the removal of
   all communication arcs associated
   with communication statements in s;

   For any entry, if the three columns of
   A' corresponding to $p_{mep}$, $p_{send}$, and

   $p_{ack}$ each have all zero entries,

   then remove the columns from A';
   Else if not every column of A' has
   both positive and negative entries,
   then return to Step 2;

   (* Some communication channels have *)
   (* lost all Accepts or Entry calls  *)
   (* -- but not both                  *)
5) Find the support ||x'|| of a linear
   combination of Ada T-invariants of the
   net APN' corresponding to A';
6) If every transition of APN' belongs to
   ||x'||, report the set s and exit.

   End LOOP2;
   End LOOP1.
END ALGORITHM INCONSISTENCY_DETECT
```

As an example, consider a modified version of the classical consumer-producer problem:

```
Producer:               Buffer:
BEGIN                   BEGIN
WHILE cond1 LOOP        WHILE cond2 LOOP
Buffer.produce;         SELECT
Buffer.msg_send;        WHEN
ACCEPT msg_ack;          buffer not full =>
ACCEPT msg_ack;          ACCEPT produce;
END LOOP;                Producer.msg_ack;
END;                     ACCEPT msg_send;
                        OR
Consumer:               WHEN
BEGIN                    buffer not empty =>
WHILE cond3 LOOP         ACCEPT consume;
Buffer.consume;         END SELECT;
END LOOP;               END LOOP;
END;                    END;
```

Note that after Producer supplies an item to Buffer, both tasks intend to exchange messages (msg_ send and msg_ack). Unfortunately, the programmer made a mistake in coding this exchange. The result is such that whenever Producer sends an item to Buffer, the program deadlocks.

The Ada net for the program is shown in Fig.3. The first iteration of Loop1 in the algorithm INCONSISTENCY_DETECT reveals two Ada T-invariants:
x1 = (1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
and
x2 = (1 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0).
Since some transitions do not belong to ||x'||, where x' = x1 + x2, the program has an inconsistency deadlock. During the second iteration of LOOP1 (with |s| = 1), we are trying to remove the deadlock. After we remove the communication arcs associated with the producer's first Accept statement, the algorithm reports three Ada T-invariants:
x1 = (1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0),
x2 = (1 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0),
and
x3 = (1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1).
Let x' = x1 + x2 + x3. Now, ||x'|| includes every transition of the net. Therefore, the Accept statement is reported as a "cause" of the inconsistency deadlock. Presumably, the programmer would then update the program by removing the extra Accept statement. The reader may note that

the new version would still suffer from a deadlock. This deadlock, however, is not an inconsistency deadlock; it is a circular deadlock and will be treated in the next section. The Ada net for the new program is illustrated in Fig. 4.

Algorithm INCONSISTENCY_DETECT is designed to find inconsistency deadlocks. In the best case (i.e., the program has no inconsistency deadlocks), when all the AE transitions belong to the support of a linear combination of the Ada T-invariants, only one execution of LOOP2 (LOOP1) is needed. Thus, the time complexity in this case is bounded by the time complexity of an integer programming algorithm (denote it by $C_{mn,}$ where m and n are the dimensions of the incidence matrix). which has been shown to be polynomial in the number of matrix columns (Petri net places) [6]. In the worst case, when all communication statements must be removed, Step 2 ("LOOP2") may execute upto 2^k times, where k is the number of communication statements. In a general case, when only two or three communication statements are causing the deadlock, the time complexity of INCONSISTENCY_DETECT is $O(k^2 \times C_{mn})$ or $O(k^3 \times C_{mn})$ respectively. Note that a general state enumeration technique would always require $O(n^T)$ time, where n is the number of concurrency related statements and T is the number of Ada tasks [1].

4.2. Circular Deadlocks

Even if an Ada program is free of inconsistency deadlocks, some statements may not belong to any static execution path of the program. In this section we discuss potentially reachable static deadlocks in Ada programs free of inconsistency deadlocks.

Definition: A circular deadlock is a deadlock (in a program free of inconsistency deadlocks) due to a set of communication statements, each in separate tasks, mutually suspending each other and, thus, also the control of flows of their respective tasks.

An Ada net dicircuit (directed circuit) which represents a circular deadlock is referred to as a C-type dicircuit (dicircuit of circular type). For example, the only circular deadlock for the program of Fig. 4 is represented by the C-type dicircuit (t18, t19, t20, t9, t10, t11). A C-type dicircuit possesses the following properties:
1) The dicircuit is composed of process-subnets' segments which start and end with communication transitions (e.g., [t18, t19, t20] and [t9, t10, t11] in Fig. 4. The starting transition of such a segment is referred to as a ready transition, and the ending transition as a last-executable transition. The last-executable transition of one segment is connected to the ready transition of another segment through a communication place (e.g., in Fig. 4, last-executable transitions t20 and t11 are attached to ready transitions t9 and t18, respectively). Thus, a C-type dicircuit has a set of ready transitions and a set of last-executable transitions.
2) A set of communication places representing at least two Ada entries belongs to the C-type dicircuit (e.g., p10 represents entry msg_ack and p7 represents entry msg_send in Fig.4). Otherwise, the dicircuit is called a single-entry dicircuit and is not a C-type dicircuit.
3) If more than one segment of a process-subnet belongs to the same dicircuit, such a dicircuit is called a multi-sequential-part dicircuit and is not considered a C-type dicircuit. It is easy to show that such a dicircuit does not represent any circular deadlock; for if it did, then in order to "enable such a dicircuit", we would need to have a marking which would enable all the ready transitions of the dicircuit. Since one of the process-subnets has more than one ready transition, reachability of such a marking would contradict to the safeness of Ada nets (Theorem 1).

Theorem 3: A sufficient condition for an Ada net to have a C-type dicircuit is that there is a minimal S-invariant y such that the places of its support $||y||$ have no tokens at M_0 and do not form either a single-entry or multi-sequential-part dicircuit.

Proof: If the support $||y||$ does not have any tokens at M_0, then the number of tokens in $||y||$ is 0 for any marking reachable from M_0 [5]. If the places of $||y||$ do not form either a single-entry or a multi-sequential-part dicircuit, then the places in $||y||$ form a C-type dicircuit. q.e.d.

The condition of Theorem 3 can be used to detect every circular deadlock in a program that has only one Entry call and one Accept statement per entry. The example of Fig. 4 shows an Ada net for which reporting of S-invariants provides complete information about is C-type dicircuits.

Finding circular deadlocks in general is a more complicated task. We attempt to detect C-type dicircuits by isolating them from any other possible dicircuits. It is important to first remove the dicircuits representing iterative statements, such as Loop statements. The procedure for that is illustrated in Fig. 5. We remove a dicircuit representing a Loop statement by replacing place p1 (representing the Loop statement) by two places pk and pm.

ALGORITHM CIRCULAR_DETECT

Input:
A - The incidence matrix of net APN.

Output:
The set of minimal supports of submarkings enabling all C-type dicircuits in APN. Communication places are not included in the supports.

```
Method:
1) Disconnect the dicircuits representing
   Loop statements;
2) Remove all initially marked places
   from A. The result is a new matrix A'
   for a new net APN'. Perform procedure
   PRUNE_TREE;
3) Perform procedure ALL_IN_DICIRCUITS;
4) Trace all dicircuits in A' and place
   them into a set C. Remove all single-
   entry and multi-sequential-part di-
   circuits from C. For each dicircuit
   in C, find the support of the minimal
   submarking enabling the dicircuit and
   place it into a set R (if not there).
END ALGORITHM CIRCULAR_DETECT
```

```
PROCEDURE PRUNE_TREE
(* Makes a Petri net connected *)

Repeat the following step until every
place and transition of APN' has at least
one input and at least one output arc;

If any place or transition of APN' has no
input or output arcs, then remove it from
APN'. The new incidence matrix of APN' is
still denoted by A'.
END PROCEDURE PRUNE_TREE
```

```
PROCEDURE ALL_IN_DICIRCUITS
(* Makes a net strongly connected *)

If APN' has n transitions and m places,
create a (m+n)x(m+n) matrix L for all
vertices of APN'. Initially L has all
entries set to 0. If there exists an arc
from a transition t to a place p in APN',
set L(t,p) to 1. By symmetry, an arc from
a place p to a transition t implies that
we set L(p,t) to 1;

For each entry of L, perform Dijkstra's
shortest path algorithm [7] to derive a
new matrix L;

For all vertices vi, if there is no
vertex vj such that L(vi,vj) > 0 and
L(vj,vi) > 0, and APN' has more than four
transitions and four places, then remove
vi from A' (from APN'), perform Procedure
PRUNE_TREE and go to the first step of
this procedure;

Return the resulting Ada net APN'.
END PROCEDURE ALL_IN_DICIRCUITS
```

A detailed stepwise discussion and analysis of the algorithm can be found in [3]. The time complexity of the algorithm can be shown to be polynomial in the number of places of an input Ada net. The following example illustrates the algorithm CIRCULAR_DETECT.

Consider the Ada net illustrated in Fig. 4. Step (1) disconnects Loop statements represented by places p25, p16, and p14. Step (2) removes all original places and transitions of the net, except for the ones with thickly drawn arcs (in Fig. 4) as their inputs and outputs. Step (3) has no effect in this example, since every place (transition) left has at least one input and at least one output thickly drawn arc. Step (4) traces the only C-type dicircuit of the net: (t18, t19, t20, t9, t10, t11). When the algorithm terminates, set R has only one element, the set {p28 and p20} (representing the communication statements that would block if the deadlock is reached).

In general not every circular deadlock is of interest to a programmer. Only the ones which are actually reachable by static execution paths should be detected and reported. Such circular deadlocks are referred to as potentially reachable circular deadlocks. Structural analysis (using structural restrictions and Petri net invariants) is necessary but not sufficient for detecting potentially reachable circular deadlocks. Dynamic (reachability tree) analysis must be used too. Algorithm CIRCULAR_REACHABLE, presented next, reports only the potentially reachable C-type dicircuits of an Ada net.

```
ALGORITHM CIRCULAR_REACHABLE

Input:
Ap  - An Ada program;
APN - The Ada net corresponding to Ap;
X   - The set of Ada T-invariants of APN;
R   - The set of the minimal supports of
      submarkings enabling the C-type di-
      circuit of APN.

Output:
The set R' of minimal supports of reach-
able submarkings enabling the C-type di-
circuits of APN.

Method:
(* T-invariant directed reachability *)
(* graph generation                  *)

If R is empty, report that Ap is cir-
cular deadlock-free and exit.

LOOP1: For each x in X perform
(* Select T-invariant as a firing *)
(* count vector                   *)
M:=$M_0$; x':=x; R':=Ø; STACK:=Ø;

LOOP2: Repeat
(* Perform selective reachability *)
(* graph generation               *)
If no tj with x'(tj) > 0 is enabled at M,
then do
   (* if x'=0 a static execution  *)
   (* path in Ap is found         *)
   If STACK is empty exit LOOP2;
   Retrieve the last saved marking of
   APN, firing count vector, and transi-
   tion in conflict, and store them into
   M, x', and tk, respectively;
```

```
Else if more than one transition is
enabled at M then do
   Assign one of those transitions to tk;
   For every other transition in con-
   flict, push current values of M and x'
   onto STACK;
Else assign the enabled transition of APN
to tk;

Fire transition tk to get new marking M;
Let x'(tk) := x'(tk) - 1;
If the current M was considered before,
then do
   If STACK is empty exit LOOP2;
   Retrieve the last saved marking of
   APN, firing count vector, and tran-
   sition in conflict, and store them
   into M, x', and tk, respectively;
Else if any element of R is a subset of
the support of M and the element does
not belong to R', then add the element
to R';

END LOOP2;
END LOOP1.
END ALGORITHM CIRCULAR_REACHABLE
```

An Ada program that is free of inconsistency deadlocks and free of potentially reachable circular deadlocks is referred to as a static deadlock-free program. Even though the program in Example 1 has a circular deadlock, it is a static deadlock-free program, since the circular deadlock reported by Algorithm CIRCULAR_DETECT is not reachable (as would be reported by Algorithm CIRCULAR_REACHABLE).

Algorithm CIRCULAR_REACHABLE is more efficient than "blind" tracing of the Petri net's reachability graph. CIRCULAR_REACHABLE uses the knowledge of Petri net's Ada T-invariants in order to reduce the time complexity of Petri net dynamic (reachability) analysis. As an example, consider the Ada net illustrated in Fig. 4. As noted earlier, the Ada net is free of inconsistency deadlocks and has three Ada T-invariants.
x1 = (1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0),
x2 = (1 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0),
and
x3 = (1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1).
T-invariant x1 indicates that the program may terminate (i.e., the Petri net may cycle) without any communication between the tasks Producer, Buffer, and Consumer. Ada T-invariants x2 and x3 correspond to the nets shown in Figs. 6a and 6b respectively. Generating the reachability graph for the net in Fig. 4 using x1 or x2 as a guiding firing count vector does not lead to a marking whose support would include both p28 and p20, the places previously identified by Algorithm CIRCULAR_DETECT. On the other hand, when the algorithm generates the reachability graph using x3 as the guiding firing count vector, we eventually reach marking M = (0 1 0 0 0 1 0 0 1 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0). Since M's support includes both p28 and p20, CIRCULAR_REACHABLE reports the potentially reachable C-type dicircuit. Presumably, the user would then eliminate this deadlock by interchanging the order of the statements "Producer. msg_ack" and "ACCEPT msg_send" in task Buffer. If we were to use blind reachability graph analysis for the Ada net in Fig. 4, then we would generate a reachability graph of eighty two vertices. In contrast, our algorithm generates a two-node reachability graph when using x1 as the guiding T-invariant, a nineteen-node graph when using x2 as the guiding T-invariant, and a sixteen-node graph when using x3 as a guiding T-invariant. Thus, the static deadlock detection system is more time and space-efficient than unrestricted reachability graph analysis. In fact, for an Ada program Ap modeled by an Ada net APN, the more Ada T-invariants APN has, the more efficient the analysis of APN by CIRCULAR_REACHABLE. The number of T-invariants of APN depends on the number of conflict statements (IF, Select, and Case) that have nested communication statements.

The analysis can also be considered to be modular since a user interested in some particular statements may not have to deal with the whole reachability graph of the Ada net modeling his system. Only the Ada T-invariants whose supports include the transitions which represent the statements of interest need to be used in the algorithm CIRCULAR_REACHABLE.

5. SUMMARY

In this paper, we described a static deadlock detection method for Ada tasking programs. First, an Ada program is translated into a Petri net, called an Ada net, which properly models the communication patterns and control flow of the source program. Then, the Ada net is analyzed for existence of static deadlocks based on structural analysis and dynamic analysis (using the reachability graph). Here static deadlocks are considered to be either inconsistency or circular deadlocks. Algorithms for detection of both classes were presented and illustrated by examples. Due to the use of both structural and dynamic Petri net techniques, our static deadlock detection method is relatively time(space)-efficient and can support modular analysis of Ada tasks.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation under Grants ECS83-10719 and DMC85-10208.

REFERENCES

[1] R. Taylor, "A General Purpose Algorithm for Analyzing Concurrent Programs", Communications of the ACM, Vol. 26, No. 5, May 1983, pp. 362-376.

[2] S. German, "Monitoring for Deadlocks and Blocking in Ada Tasking", IEEE Transactions on Software Engineering, Vol. SE-10, No.6, November 1984.

[3] B. Shenker, Using Petri Nets for Automated Detection of Static Deadlocks in Ada Programs, Master's Thesis, Department of Electrical Engineering and Computer Science,University of Illinois, Chicago, September 1985.

[4] S. M. Shatz and W. K. Cheng, "An Approach to Automated Static Analysis of Distributed Software", Proceedings of the First International Conference on Supercomputing Systems, Dec. 1985, pp. 377-385.

[5] T. Murata, "Modeling and Analysis of Concurrent Systems", Handbook of Software Engineering, C.R. Vick and C.V. Ramamoorthy (eds.), Chapter 3, Van Nostrand Reinhold, New York, 1983.

[6] N.K. Karmarkar, "A New Polynomial-time Algorithm for Linear Programming", Proceedings of 16th Annual ACM Symposium of the Theory of Computing, Washington DC, 1984.

[7] E. Horowitz and S. Sahni, Fundamentals of Data Structures, Computer Science Press, Potomac, Md., 1977.

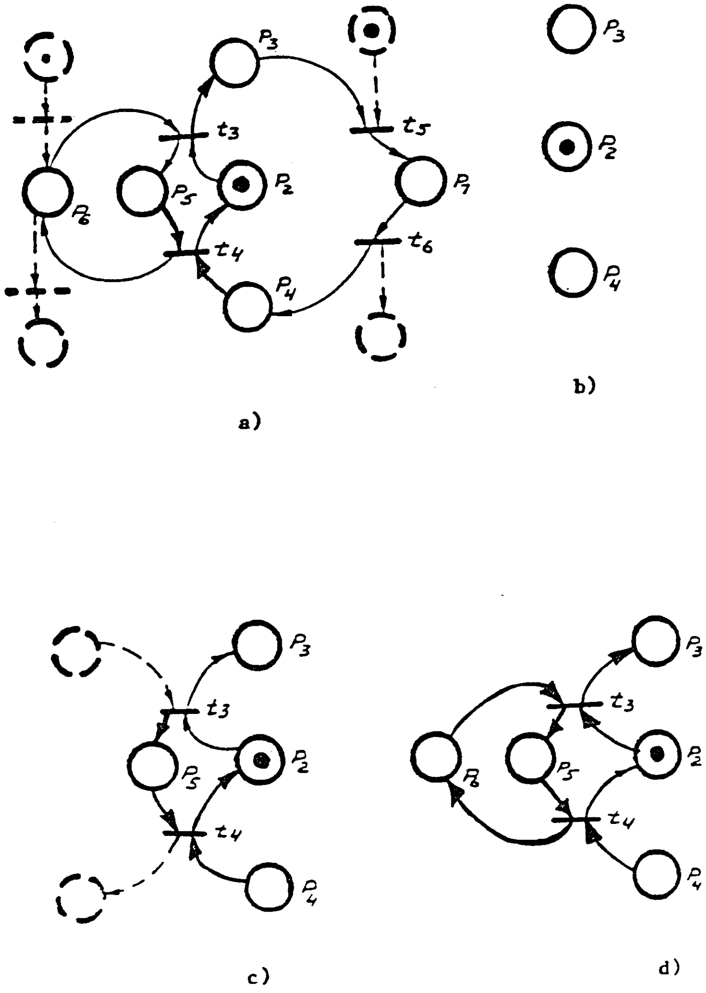


Fig. 1. Stepwise Translation of an Ada Program Into an Ada Net

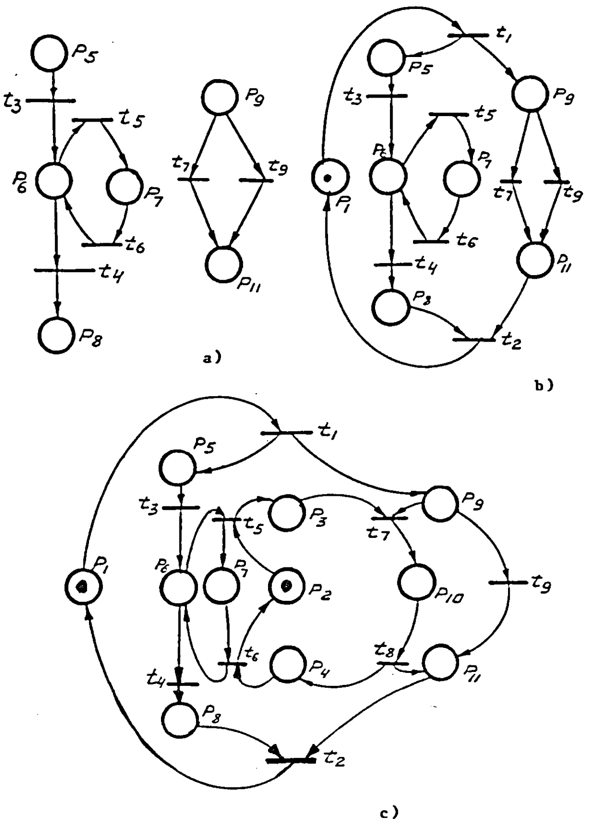


Fig. 2. Process Subnets in a Distributed Environment

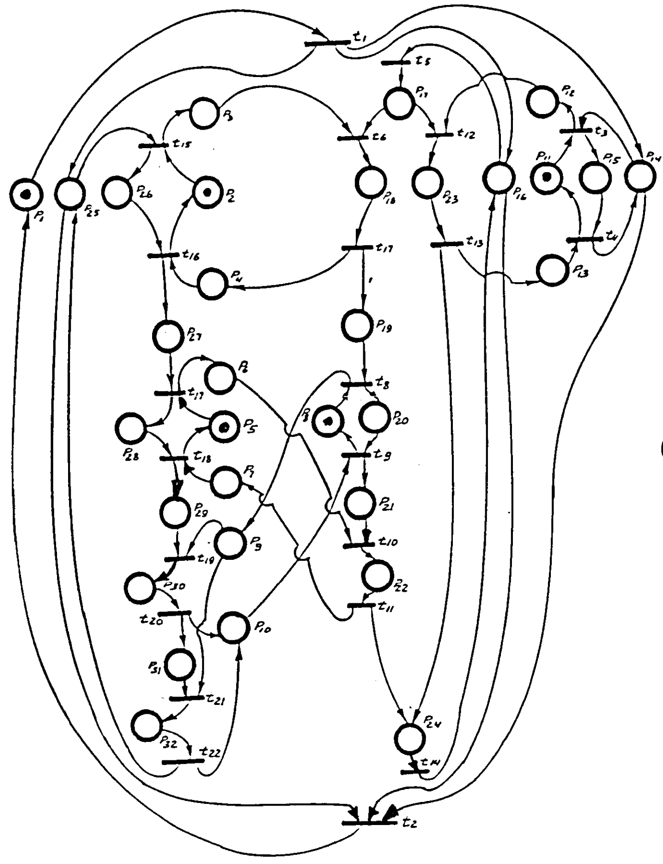


Fig. 3. Ada Net for a Producer-Consumer Program
With Inconsistency Deadlocks

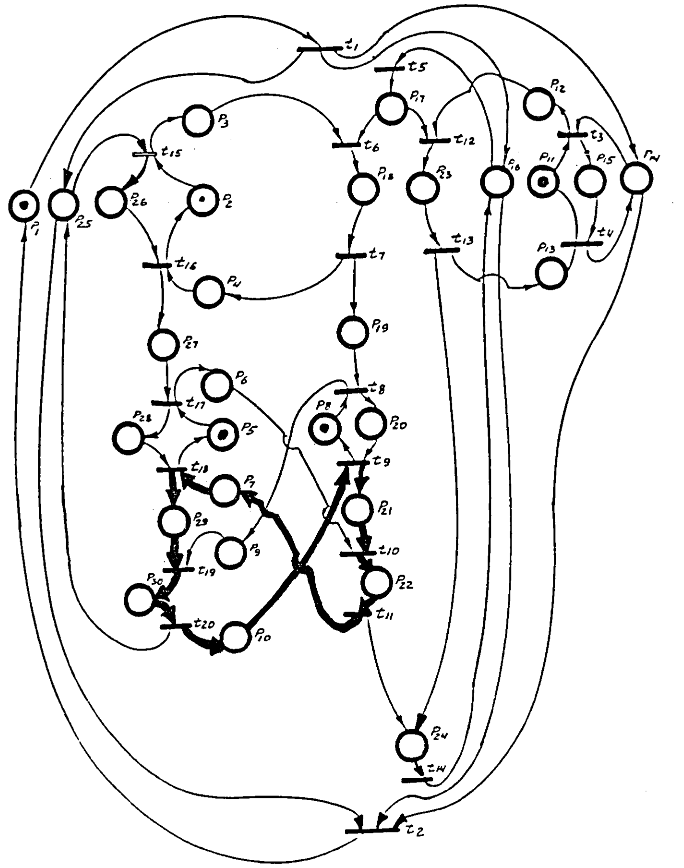


Fig. 4. Ada Net for Producer-Consumer Program
With No Inconsistency Deadlocks

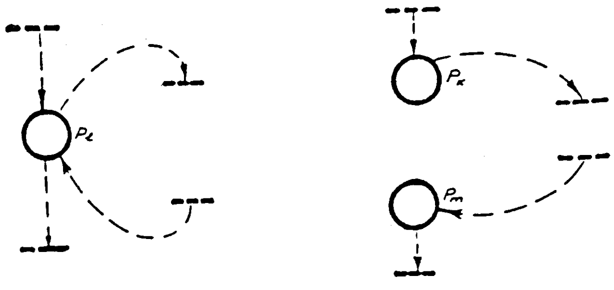
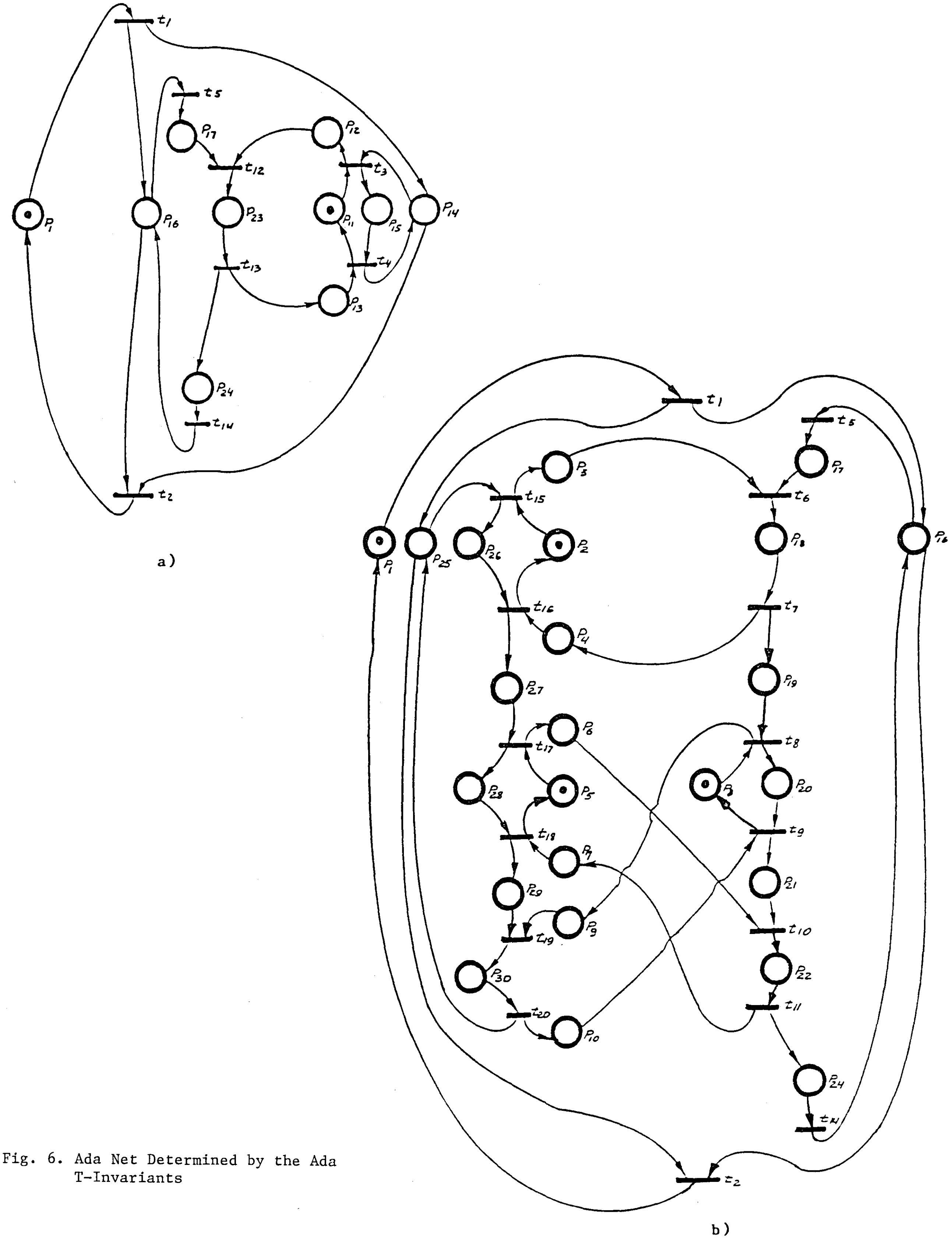


Fig. 5. Loop Transformation



a)

b)

Fig. 6. Ada Net Determined by the Ada T-Invariants

A CAD Tool for Stochastic Petri Nets

by
Michael K. Molloy
Department of Computer Science
4212 Wean Hall
Carnegie-Mellon University
Pittsburgh, PA, 15213

Abstract

This paper provides a description of a new CAD tool for Stochastic Petri Nets. The primary philosophy for the software is to maintain a flexible design environment while providing support tools to quickly and easily evaluate different aspects of the design. Several components of the software are innovative and are described in some detail. The user interface is graphical with highly interactive layout and editing features. The most significant feature is the inherent support for hierarchical modeling of systems.

Introduction

Over the last two decades, Petri Net [12] models have been used in a wide area of applications. More recently, Petri Net models have been extended to include the concept of time. These models fall into two basic classes, Timed Petri Nets (TPN) [14, 15, 18] and Stochastic Petri Nets (*SPN*) [7, 11, 8]. The TPN model has the concept of an underlying clock that clicks off time and can have synchronous transition firings. The Stochastic Petri Net model has the underlying notion of time as a random variable where events are asynchronous. The bridge between the two models is the discrete time Stochastic Petri Net model [9] with unit probabilities.

Several application areas have adopted one or more of these models. Research in flexible manufacturing, PERT analysis, computer architectures, system reliability, and network protocols are the most familiar examples.

There have been several modifications of these basic models, in addition to the the various semantic interpretations of firing rules. First, the Generalized Stochastic Petri Nets [6] extend Stochastic Petri Nets by adding the concept of instantaneous transitions. Second, the Extended Stochastic Petri Nets [2] add features to relax the exponential assumption on the transition firing rates. Third, the Generalized Timed Petri Net [5] adds a selection probability to groups of transitions but maintains the discrete time concept. Fourth, Stochastic Petri Nets with 'new better than used' distributions for firing times are being used to specify regenerative simulation models [4]. All of these models have had analysis software written to support their application.

All of these software packages provide similar features for general Markov analysis which were previously available for queueing network models, without the need to directly specify all of the possible states and state transitions. These include, analytical solutions where possible, and simulations when necessary. Since many new design problems are difficult or impossible to cast into a queueing network model, the need for user friendly Markov analysis tools, based upon a *SPN* representation, is clear. Unfortunately, the widespread use of such tools will require an improved user interface.

Stochastic Petri Nets

Recall the formal description of Petri Nets [13] where the model PN has places P, transitions T, input and output arcs A and an initial marking M.

$$\begin{aligned} PN &\triangleq (P,T,A,M) \\ P &= \{p_1,p_2,\ldots,p_n\} \\ T &= \{t_1,t_2,\ldots,t_m\} \\ A &\subset \{P\times T\}\cup\{T\times P\} \\ M &= \{\mu_1,\mu_2,\ldots,\mu_n\} \end{aligned}$$

The marking may be viewed as a mapping from the set of places P to the natural numbers N.

$$M:P\rightarrow N \text{ where } M(p_i)=\mu_i \text{ for } i=1,2,\ldots,n \tag{1}$$

Define for a Petri Net PN the set function I of input places for a transition t.

$$I(t)\triangleq\{p\,|\,(p,t)\in A\} \tag{2}$$

Define for a Petri Net PN the set function O of output places for a transition t.

$$O(t)\triangleq\{p\,|\,(t,p)\in A\} \tag{3}$$

As is common in practice, a Petri Net can be drawn using circles to represent places and bars (or boxes) to represent transitions. Tokens, to denote a marking, are represented as dots inside the circles (places). A five place, five transition Petri Net could look like the one shown in figure 1.

The continuous time stochastic Petri Net $SPN \triangleq (P,T,A,M,\lambda)$ is extended from the Petri Net $PN \triangleq (P,T,A,M)$ by adding the set of average, possibly marking dependent, transition rates $\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_m\}$ for the exponentially distributed transition firing times. These models are equivalent to continuous time homogeneous Markov chains. So, the *SPN* model may be considered as a concise representation, or generator, of the Markov representation a system.

The *SPN* model, by the nature of its being a generator, is very useful in the verification of the state space of the system. It has all of the formal mechanisms to verify that the model correctly represents the the system, before the performance analysis is attempted. Since the entire verification and analysis phase is automated, no errors are introduced as would be if a new model was described for the performance analysis phase.

Tool Description

This section describes the tool in some detail. The basic model of the Stochastic Petri is augmented with a new transition, called a subnet transition. This transition is used to represent an entire subnet defined (or about to be defined) in the library of nets on disk. The basic goal of the hierarchical model is to keep the size of the current net under design (representing some component at some level of abstraction) to a manageable size. We anticipate that a *SPN* model would be made up of a very high level description with lots of subnet transitions. Then, at the next level of abstraction, a successive refinement of those transitions would associate another net structure to that transition.

The following subsections describe each of the three major features of the tool that are unique and innovative. The first subsection describes the user interface, its philosophy, function, and use. The second subsection describes some of the special design features of the analysis components. The last subsection describes the current approach to hierarchical modeling along with some of the problems encountered with the approach.

The User Interface

The basic philosophy of the user interface is simple. The user interface should be flexible enough to let the designer be creative (even artistic) while removing any tedium from the activity. This is accomplished by adopting the following goals.

1. The user interface should provide very rapid, easily selectable options.

2. Anything that is repetitious or algorithmic should be done by the tool.

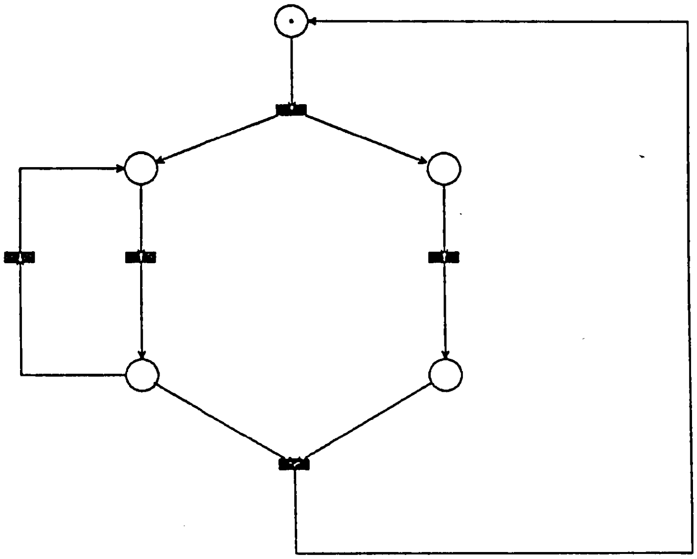


Figure 1.

3. All of the rules should be gently enforced by the tool.

4. Do not try to be smarter than the designer.

5. Anything that can be done, can be undone.

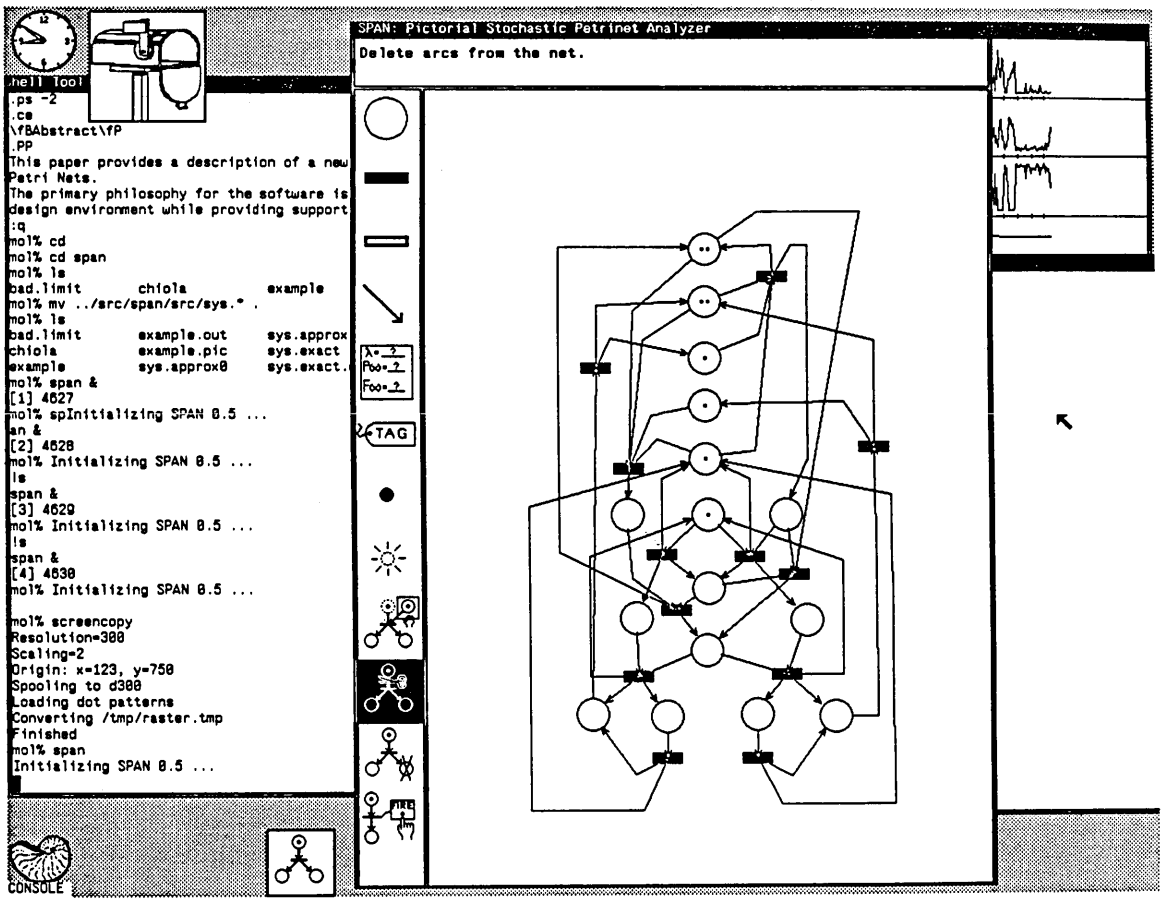
6. Options should be split into logical categories with different access for each.

7. Graphical representations of options are the best for the visually oriented human.

8. Both a top down and bottom up approach must be supported.

The user interface is constructed as a tool under the SUNTOOLS ® window management software. The tool has all of the features of a normal window. It can be moved, modified in size or shape, closed into icon form, and exist in multiple copies on the screen. The mouse is the primary input device. All drawing, menu selection, and analysis activities are initiated by mouse key clicks. The keyboard is used only in those cases when arbitrary text input is needed. The basic layout of the tool, the icons, and an example drawing can be seen in figure 2. The tool has three basic subwindows which make up three of the five functional user interface components.

® SUNTOOLS is a registered trademark of Sun Microsystems Inc.



Delete arcs from the net.

```
.ps -2
.ce
\fBAbstract\fP
.PP
This paper provides a description of a new
Petri Nets.
The primary philosophy for the software is
design environment while providing support
:q
mol% cd
mol% cd span
mol% ls
bad.limit         chiola              example
mol% mv ../src/span/src/sys.* .
mol% ls
bad.limit         example.out         sys.approx
chiola            example.pic         sys.exact
example           sys.approx0         sys.exact.
mol% span &
[1] 4627
mol% spInitializing SPAN 0.5 ...
an &
[2] 4628
mol% Initializing SPAN 0.5 ...
ls
span &
[3] 4629
mol% Initializing SPAN 0.5 ...
ls
span &
[4] 4630
mol% Initializing SPAN 0.5 ...

mol% screencopy
Resolution=300
Scaling=2
Origin: x=123, y=750
Spooling to d300
Loading dot patterns
Converting /tmp/raster.tmp
Finished
mol% span
Initializing SPAN 0.5 ...
```

Figure 2.

1. The menu subwindow (it can be on the left or right hand side) is a collection of icons representing drawing and editing features.

   a. Create a Place

   b. Create a Transition

   c. Create a Subnet Transition (a transition representing a subnet)

   d. Create an Arc

   e. Annotate a Transition (specify firing rate, etc.)

   f. Tag a Node (Place, Transition, or Subnet) with an ASCII string

   g. Add a Token

   h. Delete a Token (add an 'anti-token')

   i. Move a portion of the Net

   j. Delete an Arc

   k. Delete a Node

   l. Manually Fire a Transition or Subnet

2. The message subwindow on the top of the tool window which is used for error messages, comments, and text input to selections.

3. The canvas subwindow where all the actual drawing is done. The canvas subwindow represents a window into the drawing space. It may not show all of a particular *SPN*, but can be moved around or scaled to do so.

The basic idea of the tool is similar to the popular MacPaint for the MacIntosh computer. The menu icons are used to select an action. The selection is performed by pointing at the icon, and clicking the left mouse button. Once the selection is made, the cursor can be moved back to the drawing canvas, where it changes to represent the action selected. In the case of creating places, transitions, and subnet transitions, the cursor looks like the object to be created. Once a location is determined (pointed to by the cursor) the left mouse button is clicked to create an instantiation of the object. In a similar fashion, tokens can be added or removed from places, ASCII tags can be modified, and transition firing rates can be changed. In all cases, the user selects an option from the menu and activates the option by pointing and clicking the left mouse button. An English description of the option is displayed in the message subwindow and the icon is displayed in reverse video.

Since many drawing options are not semantically correct for Stochastic Petri Nets (such as connecting transitions to transitions), the tool enforces the rules. The user gets an error message in the message subwindow when an illegal action is taken, and the action is not be allowed to occur. There are several such rules.

1. An object can not be placed on top of another object.
2. Transitions and subnet transitions can only be connected to places.
3. Deleting a place, transition, or subnet transition deletes all the arcs associated with that node.
4. All locations are lined up with an invisible grid so that being close to lining up becomes exactly lined up.
5. Tokens and Anti-tokens can only be put into places.
6. Anti-tokens have no effect when there are no tokens present.
7. Only transitions and subnet transitions can be annotated with firing rates.

There are two additional functional components of the tool. These can be seen in figure 3. First, a popup menu (textual), activated by the right mouse button, provides options for the manipulation of the drawing environment. These include several commands (again activated by pointing).

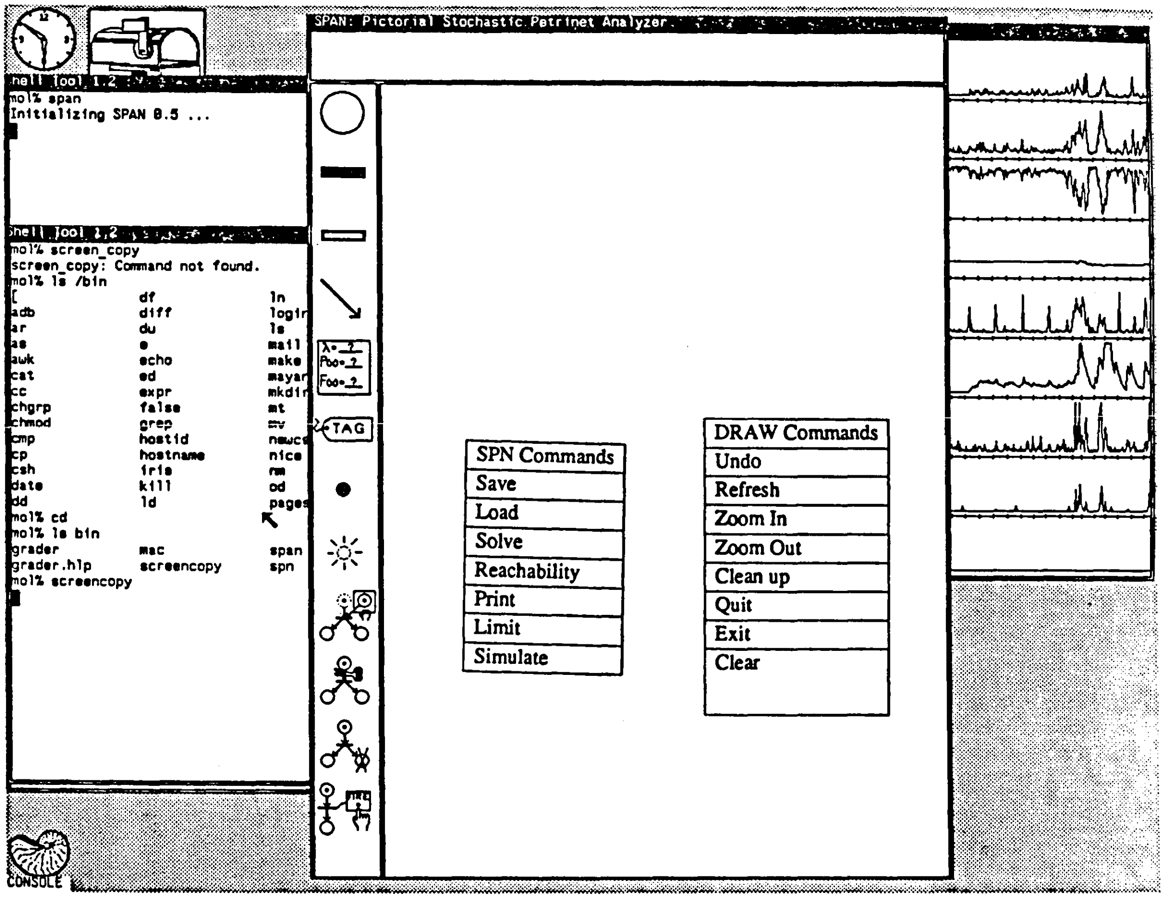
1. Undo - undo the last drawing action (icon menu only)
2. Refresh - clear and redisplay the net.
3. Zoom In - magnify the image in the canvas subwindow with the cursor location as the new center.
4. Zoom Out - reduce the image in the canvas subwindow with the cursor location as the new center.
5. Clean Up - align a coarse drawing onto the imaginary grid points. (This is for files generated by other programs.)
6. Quit - abort the current session. A confirmation is required if any modifications have been made.
7. Exit - end the current session writing out the current net to the edit file defined when the tool was started.
8. Clear - clear the screen and destroy the current net. A confirmation is required if any modifications have been made.

Second, a popup menu (textual), activated by the middle mouse button, provides analysis options for studying the network currently designed.

1. Save - save the current design in a disk file. The file format is PIC compatible for inclusion in DITROFF documents.
2. Load - load the current design from a disk file.
3. Solve - generate the reachability set, construct the Markov matrix, solve the matrix for the steady state solution, and display the average number of tokens in each place.
4. Reachability - generate the reachability set, and send a printable copy of the reachability tree to the standard output.
5. Print - generate the reachability set, construct the Markov matrix, solve the matrix for the steady state solution, and display the average number of tokens in each place. Send a copy of the token probability densities for each place to the standard output.
6. Limit - solve for bounds on the throughput of the *SPN* using the bottleneck analysis technique [10].
7. Simulate - run a Monte Carlo simulation to animate the display of the *SPN* using a mouse controlled throttle for the clock rate.

Special Data Structures

Because maintaining information about the graphical representation needs to be flexible for ease of editing, it is not appropriate for analysis. Since analysis requires speed and compact size, an entirely different structure was used for nets when the analysis phase is initiated. Each of the basic structures are described below. The result of carefully selecting each of these data structures is the superior performance of the tool. Most of the design time went into selecting and coding these structures. All of the structures are allocated dynamically to increase the range of the tool. The allocation of the reachability set and Markovian structures is only done when an analysis is requested, and deallocated when it is complete. This philosophy has made it possible for this system to draw, edit and analyze *SPN* with a large number of nodes and at least 5000 states in the reachability set, in an interactive environment.



```
mol% span
Initializing SPAN 0.5 ...
```

```
mol% screen_copy
screen_copy: Command not found.
mol% ls /bin
[               df              ln
adb             diff            logir
ar              du              ls
as              e               mail
awk             echo            make
cat             ed              mayar
cc              expr            mkdir
chgrp           false           mt
chmod           grep            mv
cmp             hostid          newcs
cp              hostname        nice
csh             iris            rm
date            kill            od
dd              ld              pages
mol% cd
mol% ls bin
grader          mac             span
grader.hlp      screencopy      spn
mol% screencopy
```

Figure 3.

Graphical Data Structures

The data structures for drawing are object oriented. A *SPN* is simply a collection of pointers to lists of objects. The lists of objects include, places, transitions, subnet transitions, arcs, and compound objects. Each element in the lists are data structures of a type for that list. All structures for places have information about the location of the place, its ASCII label, the number of tokens, etc. All structures for transitions have information about the location of the transition, its ASCII label, its firing rate, etc. All structures for subnet transitions have information about the location of the subnet transition, its ASCII label, its firing rate, the filename where the complete subnet description is (if any), etc. All structures for arcs have information about the sequence of points (arcs can be segmented) which make up the arc, the direction of the arc (to or from a place) and pointers to the specific place, transition, or subnet transition the arc connects. In addition to all of these, there is a backup net structure to save deletions for the undo of a deletion.

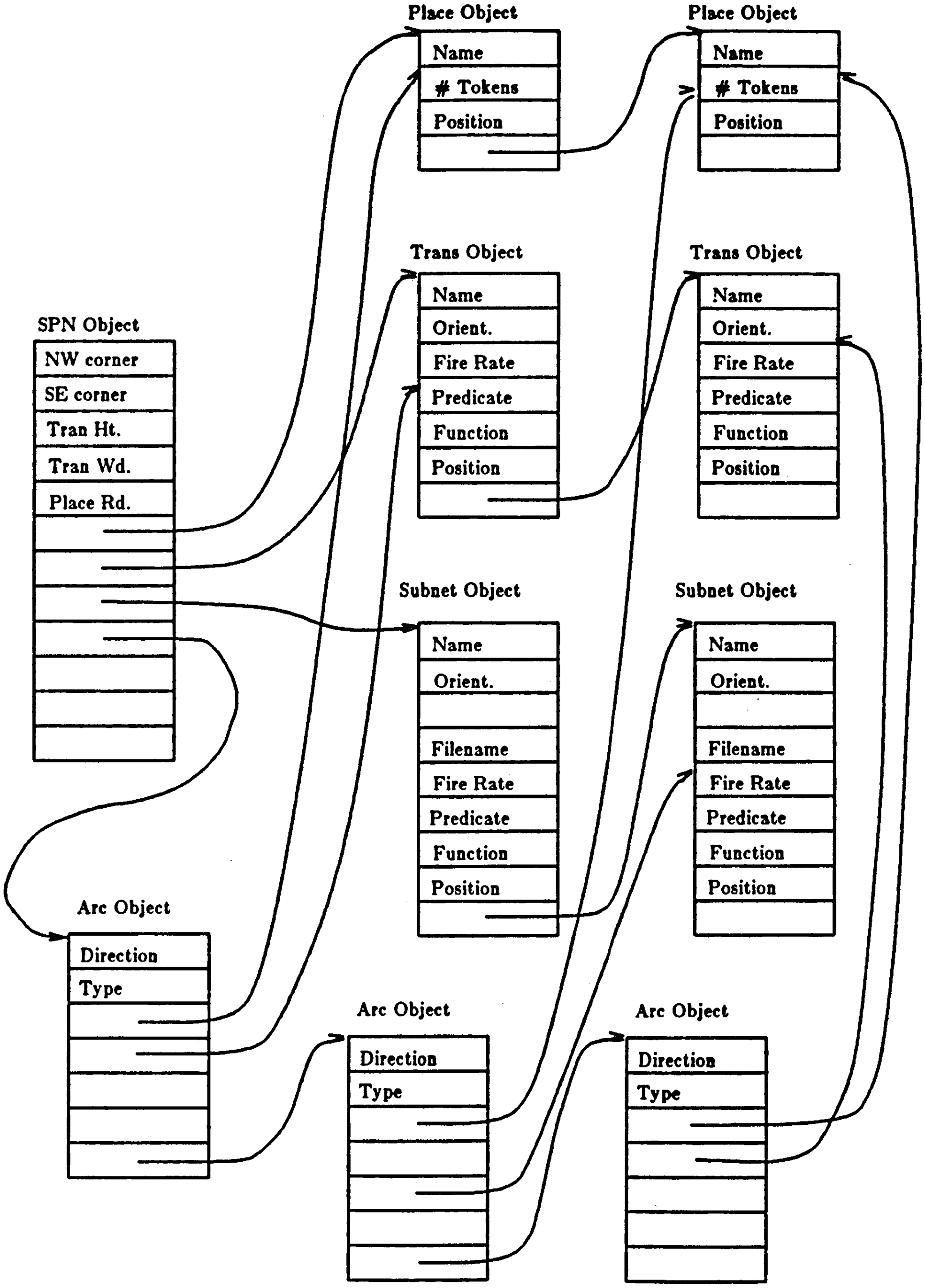
Reachability Data Structures

To speed up the response time for the analysis phase, a flexible structure for the net was not selected. Instead, the object oriented structure is translated into a packed structure for the generation of the reachability set.

The reachability set is maintained as a tree structure following the normal algorithms [13], but because of the significant lookup time in this graph structure, a second data structure was added. After the generation of a new marking from a current leaf in the reachability graph, the new marking is dropped into a binary search tree (which only contains pointers) to check for duplication. If it is found, it is not included in the reachability set. If it is not found, it is added to both the binary search tree and the reachability set.

Markov Data Structure

The end result of the analysis is the generation of the equivalent Markov chain. Since this tool is currently restricted to continuous time Stochastic Petri Nets, the resulting matrix tends to be very sparse. Therefore, the data structure used for the matrix is the uncompressed pointer storage structure defined for the Yale Sparse Matrix software package [3].


Place Object
Name
# Tokens
Position
Place Object
Name
# Tokens
Position
Trans Object
Name
Orient.
Fire Rate
Predicate
Function
Position
Trans Object
Name
Orient.
Fire Rate
Predicate
Function
Position
SPN Object
NW corner
SE corner
Tran Ht.
Tran Wd.
Place Rd.
Subnet Object
Name
Orient.
Filename
Fire Rate
Predicate
Function
Position
Subnet Object
Name
Orient.
Filename
Fire Rate
Predicate
Function
Position
Arc Object
Direction
Type
Arc Object
Direction
Type
Arc Object
Direction
Type

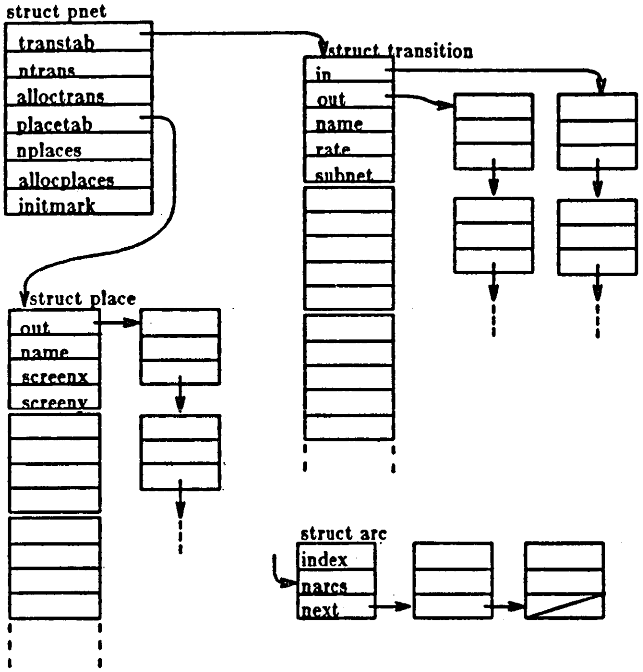


Fig.5 Packed Data Structure for a SPN

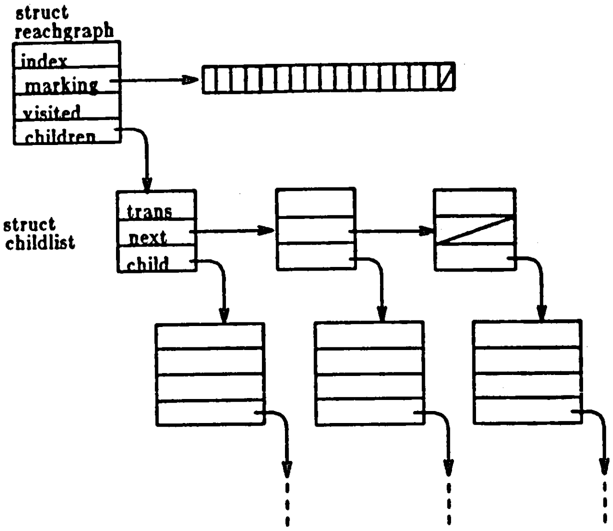


Fig. 6Reachability Graph Data Structure

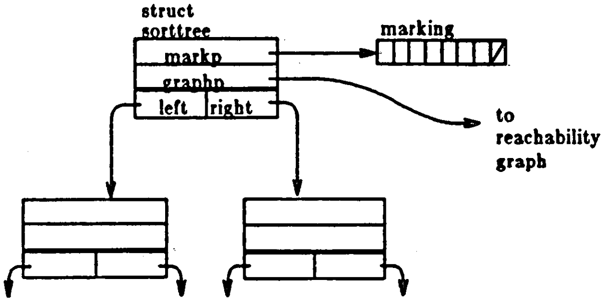


Fig. 7 Binary Tree for Lookup

Approximations for Hierarchical Models

In order to facilitate the top-down and bottom-up design concepts, a hierarchical model is needed. It is clear that the state space explosion makes large models intractable. The idea is a simple one, and a heuristic. The addition of a subnet transition allows us to postpone the description of a detailed section of the net. This section is then described and analyzed separately. The load dependent behavior of the subsystem is then included as the behavior of the subnet transition in the higher level model.

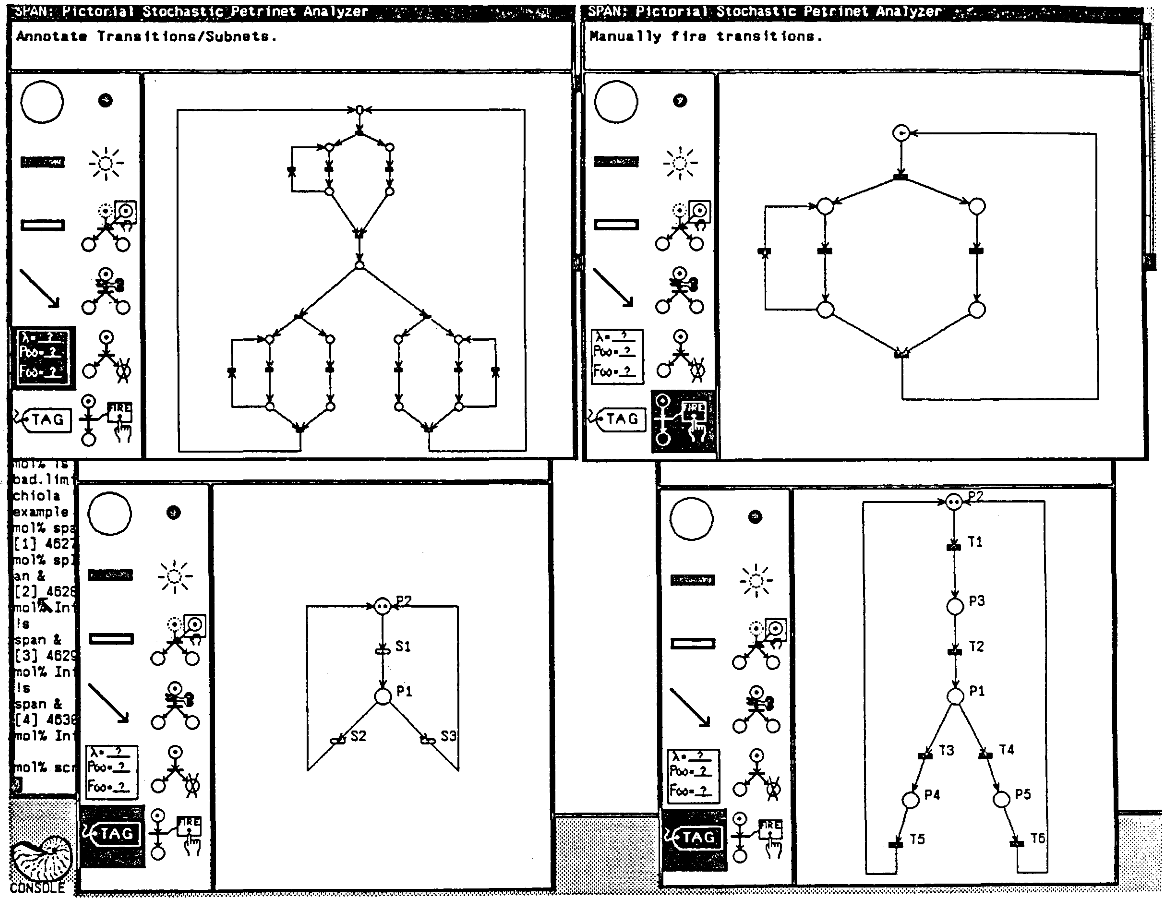
In figure 8, we see an example of such a model. The drawing in the upper left is the complete model without a hierarchical description. The drawing in the lower left is the high level model for this complicated system, where each subnet transition represents the behavior of the subsystem models, one of which is shown in the upper right hand corner. The solution would not be even close if we just approximated the throughput of the subnet as a single transition. There are several reasons for this. First, the routing probabilities in a *SPN* depend on the relative firing rates of the contending transitions. Second, the throughput of a *SPN* is load dependent. Therefore, a pair of transitions and a holding place are used to model the subsystem in this approximation. The first transition fires at precisely the same rate as the selection transition would in the exact model. That means that the transition at the entry to the subsystem is simply duplicated. The second transition models the load dependent behavior of the actual subsystem. The firing rate of the that transition depends on how many tokens are in the subsystem (equivalently, the holding place).

Although this is clearly a heuristic approximation, it is similar to the techniques used in Non-Product Form Queuing networks [16]. The results so far are very promising, and we will continue to investigate this approximation, as well as others.

An Example - A Token Ring

In order to clearly demonstrate the use of the SPAN tool, a token ring was analyzed as an example. Token rings have been analyzed by others [1, 17] using other techniques, so this example can be compared to the other alternatives.

A token ring is a network for local communication which has a physical topology in the shape of a ring. The perimeter of the ring may follow some funny path rather than a direct line to the next station, but is still a direct physical connection. The access (i.e. permission to transmit) is controlled by circulating a single token which enables the station who possesses it to transmit. This gives access to the communication medium to stations in a round-robin fashion. Several token ring designs are currently available or soon to be available. IBM Corporation and Proteon Corporation have token ring designs for local area networks.



SPAN: Pictorial Stochastic Petrinet Analyzer

Annotate Transitions/Subnets.

SPAN: Pictorial Stochastic Petrinet Analyzer

Manually fire transitions.

Figure 8

One of the performance measures for a token ring is the token latency time. That is the time it takes the token to return to a station since it was last forwarded. This latency has an fixed upper bound, since in heavy load, at most one message can be sent by each station. What the solution gives is the average value of the latency for a system under various, non-saturating loads.

The load parameters for this model are the same as used in the analysis given in Sethi's paper [17]. The time to pass a token is 50 microseconds. The time to send a packet is exponentially distributed with a mean of 400 microseconds. Messages arrive as a Poisson process if the station is idle with means 50, 100, 150, 175, 200, 300, and 1000 messages per second. Messages are assumed to be rescheduled if they arrive to a station with a message waiting.

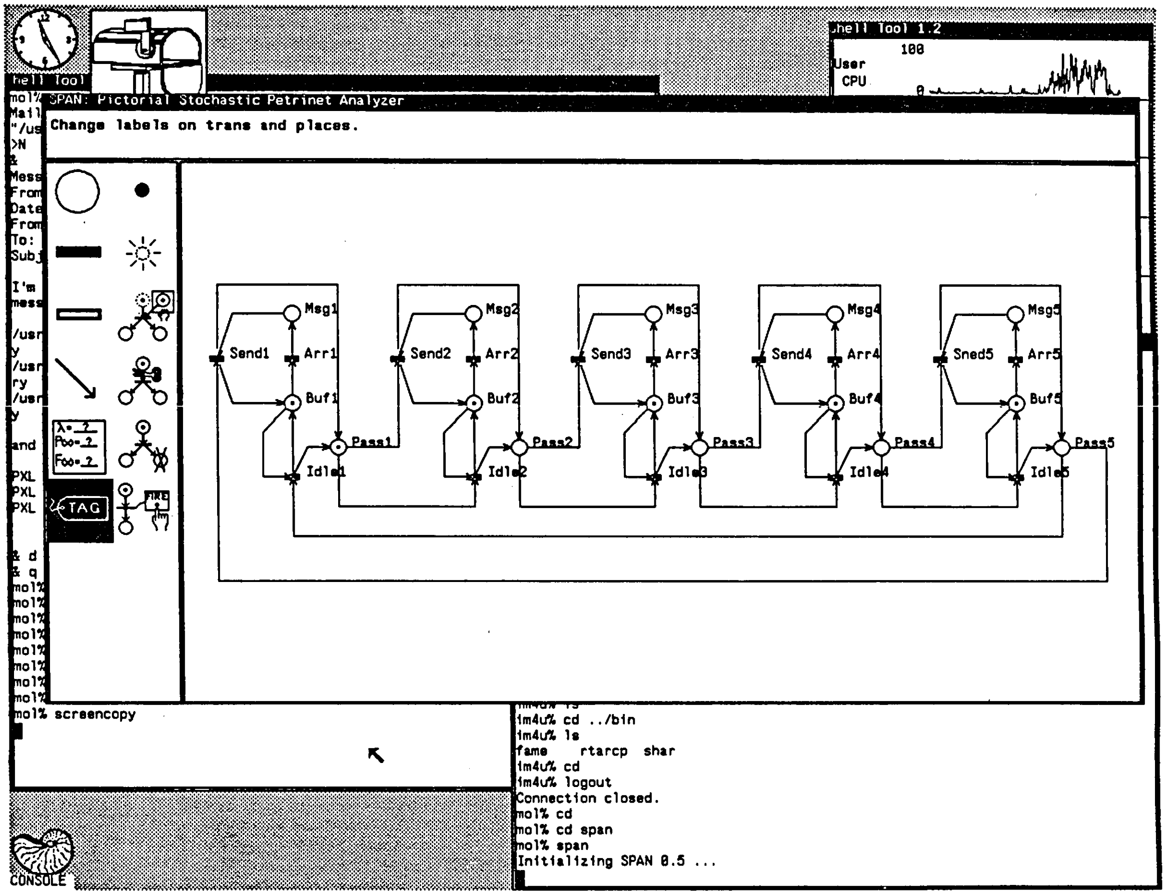
The Stochastic Petri Net model for the token ring is shown in figure 9. The various places represent whether a message is waiting to be sent (Msg), the message buffer is available (Buf), or the token is being passed to the next station (Pass). The various transitions represent the events of forwarding a token (Idle), sending a message followed by the token (Send), or having a message arrive (Arr). This particular model is for five stations spaced around the ring.

The model is then solved by SPAN to give the steady state probabilities of the number of tokens in each place. That gives the probability that each station is either idle or has a message to send. Knowing the average time to send a message or token, the average token latency (rotation time) can be calculated. In addition, since the Poisson arrival process is effectively turned off during the period of time a message is waiting, the actual offered load is the original rate times the probability the station is in the idle state.

The results of the analysis are shown graphically in figure 10. The curve starts at the minimum time to circulate a token around the five stations and increases as the rate of message arrivals increase. The curve shows the expected convexity, and would show the abrupt upper-bound at 2 milliseconds, if the system were pushed that far.

Conclusions

The CAD tool, SPAN, for the analysis of Stochastic Petri Models has proven to be very useful. The design goals of a highly interactive, user friendly, and high performance design aid have been attained. The system can handle large nets by breaking them down in a hierarchical manner and can solve medium size nets in a relatively short time (given the fact that the SUN 2/120 ® is poor at floating point). A summary of the timings for the standard example net with increasing reachability set size is given below.



```
Change labels on trans and places.
```

```
mol% screencopy
```

```
im4u% cd ../bin
im4u% ls
fame    rtarcp  shar
im4u% cd
im4u% logout
Connection closed.
mol% cd
mol% cd span
mol% span
Initializing SPAN 0.5 ...
```

Figure 9

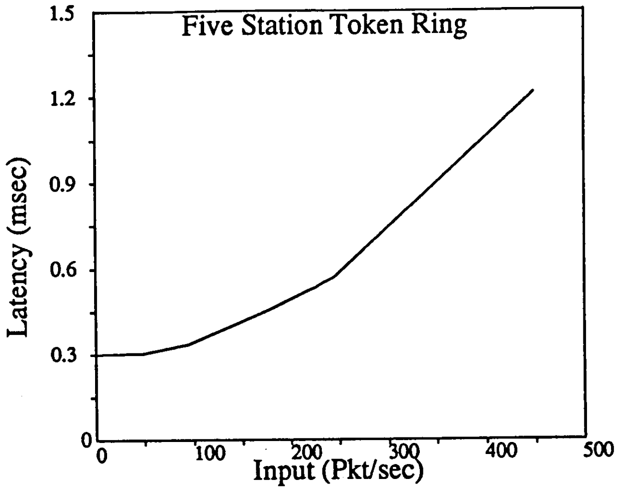


Figure 10

The sudden increase in processing time for the cases above 500 states is not completely understood. At this time, we believe that at 500 states, paging becomes a significant factor on our SUN 2/120 ® workstation since we have only 2meg of memory. This belief is supported by the timings on a 3meg SUN 2/120 ® which are shown in parentheses. The lack of data for the time on the Markovian analysis for large systems is due to the fact that those routines are currently in FORTRAN and can not do dynamic memory allocation. Therefore, if SPAN does not allocate enough space for the Yale package, the program aborts the Markovian analysis. This is currently under conversion to avoid the problem.

Tokens (#)	States (#)	Reachability (sec)	Markovian (sec)
1	5	<1	<1
2	14	<1	<1
3	30	<1	1
4	55	<1	4
5	91	1	6
6	140	3	19
7	204	6	45
8	285	10	110
9	385	18 (14)	283
10	506	27 (21)	---
11	2212	270 (130)	---
12	5322	1300 (862)	---

® SUN is a registered trademark of Sun Microsystmes Inc.

It is clear from even the small amount of use the package has seen, that several changes are needed. The human interface for reachability analysis must be changed to support some type of query-response dialogue rather than a long listing. In a similar vein, the Markovian results need to be represented differently. It would be much better to point at an arc, and get the throughput for that arc. Similarly, pointing at a transition would trigger the display of the utility of that transition. Finally, pointing at a place could give you the mean time until a token returns to that place.

Some of the drawing features need to updated. It would be nice to select and duplicate sections of a *SPN* in a fashion similar to the way we move sections of a net. The addition of a mouse activated keypad would further minimize the number of keyboard actions required.

Acknowledgements

This software could not have been written without the help of two outstanding individuals. First, a graduate student, Supoj Sutanthavibul, spearheaded all of our efforts by creating several tools on the SUN with such great style, that several sections of the SPAN code still bear his name. Second, Prentiss Riddle actually wrote half of the code with the author and provided exactly the right balance for the completion of this project in only 3 man-months.

References

[1] Bux, W. "Local-Area Subnetworks: A Performance Comparison" *IEEE Transactions on Communications*, Vol. COM-29, No. 10, 1465-1473, Oct. 1981

[2] Dugan, J.B.; Trivedi, K.S.; Geist, R.M.; Nicola, V.F. "Extended Stochastic Petri Nets: Applications and Analysis" *Performance 84*, pp. 507-519, Paris France, Dec. 1984

[3] Eisenstat, S.C.; Schultz, M.H.; Sherman, A.H. "Considerations in the Design of Software for Sparse Gaussian Elimination" *Sparse Matrix Computations* Ed. J.F. Bunch, D.J. Rose, pp. 263-273, Academic Press 1976. also in Proceedings of Symposium on Spares Matrix Computations at Argonne National Laboratory, Sept 1975

[4] Haas, P.; Shedler, G. "Regenerative Simulation of Stochastic Petri Nets" *Proceedings of the Workshop on Timed Petri Nets*, Torino, Italy, July 1985

[5] Holliday, M.A.; Vernon, M.K. "A Generalized Timed Petri Net Model for Performance Analysis" *Proceedings of the Workshop on Timed Petri Nets*, Torino, Italy, July 1985

[6] Marsan, M.; Balbo, G.; Conte, G. "A Class of Generalized Stochastic Petri Nets" *ACM Transactions on Computer Systems* Vol 2, pp. 93-122, May 1984

[7] Molloy, M.K. "On the Integration of Throughput and Delay Measures in Distributed Processing Models" Report CSD-810921, University of California, Los Angeles, 1981

[8] Molloy, M.K. "Performance Analysis Using Stochastic Petri Nets" *IEEE Transactions on Computers* Vol. C-31, No. 9, Sept. 1982, pp. 913-917

[9] Molloy, M.K. "Discrete Time Stochastic Petri Nets" *IEEE Transactions on Software Engineering* Vol. SE-11, No. 4, April 1985, pp 417-423

[10] Molloy, M.K. "Fast Bounds for Stochastic Petri Nets" *Proceedings of The Workshop on Timed Petri Nets*, Torino Italy, July 1985

[11] Natkin, S.O. "Les Reseaux de Petri Stochastiques et leur Application a L'Evaluation des Systemes Informatiques" Doctoral Thesis, Conseratoire National des Arts et Metiers, 1980

[12] Petri, C.A. "Communication with Automata" PhD Thesis, Translated by C.F. Green, Information System Theory Project, Applied Data Research Inc., Princeton N.J., 1966

[13] Peterson, J.L. "Petri Nets" *Computing Surveys* ACM Sept 1977 Vol 9 No 3 pp 223-252

[14] Ramchandani, C. "Analysis of Asynchronous Concurrent systems by Timed Petri Nets" PhD Thesis, MIT 1974 Project Mac report #MAC-TR-120

[15] Ramamoorthy, C.V.; Ho, G.S. "Performance Evaluation of Asynchronous Concurrent Systems using Petri Nets" *IEEE Transactions on Software Engineering* Vol SE-6 No. 5 Sept 1980 pp. 440-449

[16] Sauer, C.H.; Chandy, K.M. *Computer Systems Performance Modeling* Prentice Hall, 1981

[17] Sethi, A.S.; Saydam, T. "Performance Analysis of Token Ring Local Area Networks" *Proceedings of the 9th Conference on Local Computer Networks* Oct. 1985, pp. 26-31

[18] Zuberek, W.M. "Timed Petri Nets and Preliminary Performance Evaluation" *Proceedings of the 7th annual Symposium on Computer Architecture* 1980, pp. 88-96

petri - A UNIX[1] Tool for the Analysis of Petri Nets

Ira R. Forman

Microelectronics and Computer Technology Corporation
Austin, Texas 78759

ABSTRACT

An analyzer of Petri nets is described. Petri nets are a tool for modeling concurrent systems. The program supports three forms of analysis: simulation, reachability analysis, and invariant analysis. Simulation allows one to follow the behavior of the net in order to develop intuitions and correct the net. Reachability analysis computes all possible simulations and thus allows one to verify net properties. Reachability analysis is specialized for three forms of Petri nets (i.e., safe, bounded, and place/transition). Invariant analysis derives properties of nets by analyzing the incidence matrix, which implies larger nets can be analyzed. The combination of these forms of analysis yields an engineering tool that should be useful in many software engineering environments.

1. Introduction

The modeling of concurrent systems is becoming increasingly important; Petri nets are a excellent way of doing such modeling. The reader who is not familiar with Petri nets should consult Peterson [13] or Agerwala [1]. Greater detail is available in the texts by Brauer [5], Peterson [14], or Reisig [16]. Once the model is built, computer analysis is important in order to insure that desired system properties are achieved. This is especially true in those environments where the model must be modified intermittantly during development or maintenance in response to changing customer or market requirements. Computer analysis uncovers errors due to modification quickly and at a lower cost.

petri is a tool for manipulating and analyzing Petri nets. *petri* has a flexible interface that makes it usable in learning situations. In addition, *petri* provides reasonably rapid computations and has no storage limitations (other than operating system limits). Thus *petri* is a tool for doing serious analyses. Other tools for the display and analysis of Petri nets are also finding their way into industry and the classroom [3,9,12,15,17].

There are three modes of analysis provided by *petri:* simulation, reachability, and invariant. One can simulate the net with the aid of a two column display that shows the current marking and the firable transitions. One can compute the reachability graph of the net and display information about the reachability graph. One can compute the S- and T-invariants of the net and have them displayed.

For reachability analysis *petri's* preferred mode of operation is safe place/transition nets. In this case *petri* uses one bit per place to store markings, leading to a very efficient computation. *petri* also has modes for bounded nets and unrestricted nets.

2. A Description of the User Interface

This section describes the interface to *petri*. An evaluation of *petri* is given in Section 5.

2.1 Invoking petri

The *petri* program is invoked with the following UNIX command

```
petri [-s] [-b] [-g] [-q] infiles
```

If infiles are given, *petri* first reads commands from the infiles and then prompts the user. The four command line flags have the following meanings:

-s means that the user believes the net to be safe; *petri* uses this information to compute the reachability graph more efficiently. One bit is allocated per place to store markings for reachability computations. If the net is not safe, *petri* discovers this fact and terminates with an error message that contains a path (i.e., place/transition sequence) to the unsafe marking. The -s, -b, and -g flags are mutually exclusive with -s as the default.

-b means that the user believes the net to be bounded; in this mode *petri* does not do the ω search. If the user is mistaken, i.e., the net is not bounded, the computation of the reachability graph does not necessarily terminate.

[1] UNIX is a trademark of AT&T Bell Laboratories.

-g means that no assumptions are made about the net. In this case, the computation of the reachability graph does the ω search, which is a very expensive computation. (Formally speaking we are computing the coverability graph in this case; see Jantzen and Valk [8]).

-q implies the quiet mode; *petri* does not prompt the user after reading the file of commands. This flag is used for background analyses.

When used in interactive mode, the *petri* program prompts the user for commands. Output from a command may be redirected by adding a filename to the command to create a new file or by adding ">> filename" to the command to append to the file.

The *petri* commands can be partitioned into four groups; net manipulation, net display, interface support, and analysis. The following four subsections contain descriptions of the first three groups; the analysis commands are covered in Section 3.

2.2 Net Manipulation

Figure 1 depicts an example net that is used throughout this paper. The net represents two processes that share a resource. Each process requires exclusive access to the resource. Figure 2 contains the commands to build the net depicted in Figure 1. The meaning of the commands is given below.

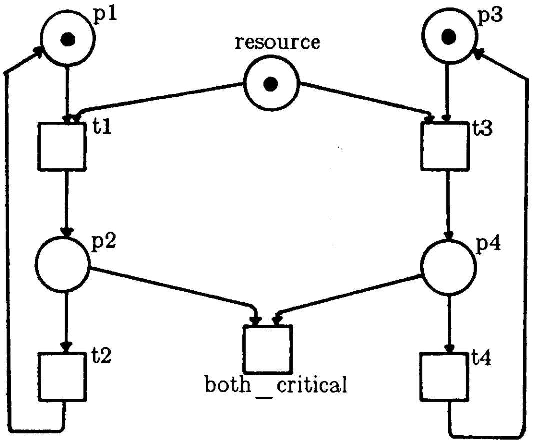


Figure 1. A sample Petri net.

add place *list_of_places_to_add*
Creates one new place for each item in the comma-separated list of the place names. The name of a place or transition is an identifier (a letter followed a sequence of letters, numbers, and underscores).

add transition *list_of_transitions_to_add*
Creates one new transition for each item in the comma-separated list of the transition names.

add arc *list_of_arc_groups_to_add*
Creates new arcs specified by each arc group in the comma-separated list of arc groups. An arc group is denoted as follows: node_a{successor1, successor2, ...}, which specifies an arc from node_a to each of successor1, successor2, ... The term "node" refers to either a place or a transition.

mark *list_of_places_to_mark*
Allows one to change the current marking of the net. The *list_of_places_to_mark* is a comma separated list of place names followed by an equal sign and a number. For example, place_a = 2, place_b = 0. If a place is mentioned alone (without an equal sign and number), this means "set the number of tokens to 1."

```
add place p1, p2, p3, p4, resource
add transition t1, t2, t3, t4, both_critical
add arc p1{t1}, t1{p2}, p2{t2}, t2{p1}
add arc p3{t3}, t3{p4}, p4{t4}, t4{p3}
add arc t2{resource}, t4{resource}, resource{t1, t3}
add arc p2{both_critical}, p4{both_critical}
mark p1, p3, resource
```

Figure 2. Commands for building the net in Figure 1.

It is not sufficient in an interface just to be able to build nets. It is also important to be able to compose them in such a way that the nets can be built by automatically translating from some language for distributed programming. For this purpose, the interface also contains the **merge, split, rename, reduce,** and **retain** commands, which are described below. The important point is that the commands facilitate the building of a net in pieces that can be subsequently composed. The **merge** command is the most important in this category. It should be noted that *petri* supports nodes that have multiple names, any of which are valid identifiers of the node. The result of the **merge** or **reduce** command is a node with multiple names.

merge *list_of_nodes*
Merges a list of transitions(places) into one transition(place) whose set of names is that of all the elements of the list. The predecessors of the resulting transition(place) is the union of the predecessors of the transitions(places) in the list. If a predecessor has arcs to multiple elements of the list, the resulting transition has multiple arcs from the predecessor. Similar actions hold for successors.

split *transition_name* **in** *number_of_parts*
Splits a transition into *number_of_parts* transitions; each transition has "#<an_integer>" appended to it beginning with 0. Each transition has the same set of

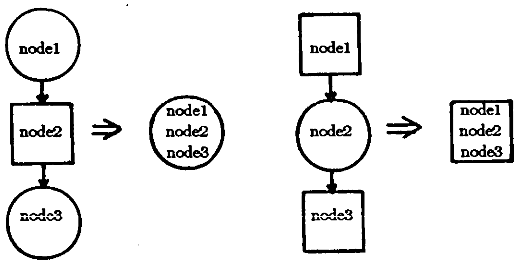
predecessor and successor places as the original transition.

rename *node_name* **to** *identifier*

Renames the node with name *node_name* with new name *identifier*.

reduce

Executes the following net reductions until there are no reductions possible.



Note that node1 has exactly one successor and node3 has exactly one predecessor. A list of the reductions is sent to the terminal or to a file with use of the same syntactic conventions as the **display** commands.

retain *list_of_nodes*

Sets a flag in each transition (place) in the list to indicate to the **reduce** command may not eliminate the transition (place). (The **reduce** command resets the flag.) This command is necessary to prevent the **reduce** command from removing the node in which interest lies when doing analysis.

2.3 Net Display

The five commands for displaying a net are described below. These commands are usually used for debugging a net (except for **display statistics**, which can contain reachability information).

display net

Displays a depth-first spanning tree of the net. States are enclosed in braces with the number of tokens (e.g., {place_p = 1}). Events are enclosed in brackets (e.g., [transition_t]).

display frame *node*

Displays the node and all its neighboring nodes.

display marking

Displays the names of the places with nonzero tokens and the number of tokens if greater than zero.

display statistics

Displays interesting statistics about the net. These are: the number of places, the number of transitions, the number of arcs. If the reachability graph has been computed already, the number of nodes, the number of terminal nodes, and the number of back arcs of the reachability graph are displayed also.

display firable

Displays the current list of the firable transitions.

2.4 Interface Support Commands

There are several commands to help interface with *petri* and UNIX. **save** *file* copies the net on the *file* in the form of *petri* commands. **source** *file* reads *file* as a source of commands. **alias** *id1 id2* allows *id1* to be used as an alias for *id2*. The purpose of this command is to allow the user to define short forms for the keywords. **!** is the shell escape. **help** displays a synopsis of the command set.

3. Types of Analyses

There are three forms of analysis provided by *petri:* reachability graph analysis, simulation, and invariant analysis.

3.1 Reachability Graph Analysis

The **compute rgraph** command computes the reachability graph and stores it in memory with no displayed output. The hitting of the interrupt character during the computation of the reachability graph causes the computation of the reachability graph to stop. In this case, some nodes are marked as type *frontier*, which means that these nodes may have successors that may not have been computed because the computation was stopped. The computation can be resumed with the **compute rgraph** command (there is a small possibility that the count of the number of arcs in the reachability graph might be off by 1 when doing this).

Once the reachability graph is computed, the **display bounds**, **display conflicts**, **display concurrent**, **display terminals**, and **display path** commands can be used to obtain information about the reachability graph. The reachability graph is automatically computed for these commands if it has not already been done.

The **display bounds** command lists the greatest number of tokens that may appear in any place. An ω indicates that the place has no finite bound. The **display concurrent** command lists pairs of transitions for which there is a reachable marking in which the pair of transitions are concurrent. The **display conflicts** command lists pairs of transitions for which there is a reachable marking in which the pair of transitions conflict. The **display terminals** command lists the terminal nodes of the reachability graph, that is, the reachable markings at which no transition is firable. The **display path to terminals** command lists for each terminal node in the reachability graph, one minimum length path from the initial marking to the terminal node. The **display path to (** *marking* **)** command lists one minimum length path from the initial marking to the marking in the reachability graph. The **display path to cover of (** *marking* **)** command lists one minimum length path from the initial marking to a cover of the marking in the reachability graph. The **display dead** command lists those transitions that can never be fired from the current marking. One analysis technique is to add to the net transitions that represent events that erroneous conditions. Then one can use the **display dead** command to verify that the erroneous transitions are never enabled. In Figure 1 *both_critical* is such a transition.

The set of commands is also intended to support the design process. The reachability computation implies an interleaving semantic model. In order to really verify the design one must show the noninterference of concurrent transitions. The **display concurrent** command exists for this purpose. In designing systems with nets, because conflict resolution is nondeterministic, the resolution is actually a choice that the designer leaves to the implementer. Designers should be able to easily enquire about these choices. The **display conflicts** command exists for this purpose.

It should be noted that once the reachability graph is computed, the **display statistics** command reports for the reachability graph on the numbers of nodes and arcs, the number of terminal nodes, and the number of back arcs. These numbers can be useful in the analysis of a net. For example, a back arc is an arc that completes a circuit. If the net is not supposed to have any cyclic behavior (as in checking for the absence of livelock), then the number of back arcs should be zero.

No matter how well displayed, a listing of the entire reachability graph is not an effective analysis tool because there is too much information. The **display** commands are provided to obtain useful information about the reachability graph. Figure 3 depicts sample outputs for some display commands for the Petri net in Figure 1. However, the number of possible display commands is endless and a method of forming ad hoc queries is needed. The answer to this problem is to view the reachability graph as a database about the behavior of the net. With this view one can see that a database query language would be useful. Now because we are interested in the behavior of the net, which occurs over time, temporal logic is an ideal query language. For this reason we have incorporated the interface to the Extended Model Checker, which enables one to form queries in temporal logic about the reachability graph. The reader should consult the paper by Clarke, Emerson, and Sistla for more information [6].

The **display emc** command displays the reachability graph as a LISP S-expression for input into the Extended Model Checker. The Extended Model Checker is a package with which one can make queries in temporal logic in terms of certain atomic propositions. The set of atomic propositions that we chose for the queries about the behavior of a Petri net are:

- the set of place names, where if *x* is a place name, then the atomic proposition *x* means "*x* is marked",
- the set of transition names, where if *x* is a transition name, then the atomic proposition *x* means "*x* is firable".

In analyzing the net in Figure 1 one could compose the temporal statement "sometimes p2 is marked and p4 is marked" and expect that the Extended Model Checker will reply false. This allows us to test the desired property of mutual exclusion directly (i.e., without going to the effort of adding the *both_critical* transition and checking that it is dead). Note that "x fires" is not an atomic proposition and one cannot enquiry about it directly. To make such a query, one may construct the nets so that the transition of interest has a unique post-state.

```
petri: display statistics
number of places = 5
number of transitions = 5
number of arcs = 14
number of rgraph nodes = 3
number of rgraph arcs = 4
number of rgraph terminals = 0
number of rgraph back arcs = 1
petri: display bounds
        1       resource
        1       p4
        1       p3
        1       p2
        1       p1
petri: display dead
both_critical
petri: display conflicts
(t1,t3)
```

Figure 3. Example of analysis commands for the net in Figure 1.

We mention above that nodes do not have just one name but a set of names. The extra names come from the use of the **merge** and **reduce** commands. The **display emc** command outputs all of the names of a node. (The other commands display only one of the names of a node.) This feature allows one to reduce the net and still make queries in terms of the original net.

3.2 Simulation

The **simulate** command presents the user with a two column display for simulation. The left column lists the marked places and the number of tokens in each place. The right column lists the firable transitions. The following keys are available as commands when looking at the simulation display.

k move the cursor up.

j move the cursor down.

f fires the transition named to the right of the cursor. The display is updated with the new marking and new set of firable transitions. (Note: currently the display shows only 22 marked places and 22 firable transitions.)

u undoes the firing of the last transition. (Repeated use undoes the entire simulation.)

p outputs the firing sequence to file *sim.path*.

q terminates the simulation with the last marking as the new net marking.

x terminates the simulation with no change to the net.

The command **simulate restart** continues the last simulation from where it was terminated.

The simulate command proved most useful when debugging nets that reachability analysis showed to have unsatisfactory behavior. The procedure was to simulate the net while moving tokens about a net picture. The advantage of having the program was that one could fire multiple transitions and the marking would be accurately kept. This was especially true when one wanted to back up and try another firing sequence. One opportunity that was missed in the implementation is the capability to execute the reachability graph analysis commands on the firing sequence of the simulation.

3.3 Invariant Analysis

The command **display invariants** lists the S- and T-invariants of the net. A T-invariant is an algebraic formula on a firing sequence that leaves the marking unchanged. An S-invariant is a algebraic formula on a subset of the places of a net such that the value of the formula remains constant for all firing sequences.

Invariants can be used to prove properties of nets. In Figure 1 the S-invariants are:

(1) p1 + p2
(2) p2 + p4 + resource
(3) p3 + p4

where the name of a place represents the number of tokens that the place contains. Based on the initial marking, it is easy to see that

$$p2 + p4 + resource = 1$$

for all reachable markings. This implies $p2 + p4 \leq 1$, which implies that mutual exclusion of p2 and p4 is a property of the net.

S- and T-invariants are explained in [11]. The algorithm of Martinez and Silva [10] is used; this algorithm produces the set of minimal support invariants. [4] contains an extensive explanation of how S- and T-invariants may be used to prove properties of protocols.

The advantage of invariant analysis is that it can be used on nets for which the reachability graph is too large to build. The disadvantage is that there is an additional step of proving net properties once the invariants are found. This can be quite difficult. Because *petri* can build very large reachability graphs, the direct route of using the display commands or the Extended Model Checker has been the one used in all problems. We have not done anything other than trivial examples with invariant analysis.

4. An Overview of Implementation

petri runs on Berkeley 4.2bsd UNIX on either a Sun Microsystems workstation, a Pyramid mainframe, or a DEC VAX. It is built with approximately 5100 lines of code that is either C, *yacc*, or *lex*. (*yacc* is a parser generator and *lex* is a scanner generator.) The program is structured like a compiler in that the input commands are parsed by a *yacc/lex* generated parser. This technique and the UNIX facilities for redirecting input and output account for our ability to achieve high capability with little software.

Figure 4 depicts a net schema that is used for a performance benchmark. This schema is chosen because it generates very large reachability graphs for small nets. Table 1 gives the time and space requirements for each of the three options for building reachability graphs when run on a Pyramid 90x. Time is given in hours, minutes, and seconds. Space is given in kilobytes. From this table one can see the importance of space to the construction of the reachability graph. In dealing with safe nets the storage consumption is measured in megabytes/minute. The use of a word instead of a bit for the marking vector (and the associated programming techniques) totally accounts for the difference between the performance for safe and bounded nets. The ω search totally accounts for the difference between the performance for bounded and general nets.

The high performance of the reachability algorithm is obtained by the following implementation techniques.

1. A reachability graph is computed rather than a reachability tree. This greatly reduces the amount of storage required to hold the structure. The number of nodes in the reachability tree is equal to the number of arcs in the reachability graph minus one. From the table above it is clear that for larger nets, the reachability graph provides large savings.
2. When a new marking is generated, the search for a duplicate (i.e., a node with an equal marking) is a costly part the computation. This search of the reachability graph is done by using a hash table with 10007 entries.
3. On the less intuitive side, the test for the firability of a transition is a very costly operation because it must be done for every transition at every reachable marking. The algorithm for computing the reachability graph uses the net itself to keep track of the current marking during the computation. This allows a quick check of firability because the predecessor places of a transition can be quickly accessed and usually the first predecessor has zero tokens because the transition is probably not firable.
4. The algorithm was specialized to compute the reachability graph for safe and bounded nets. This was done in two ways.
   a. Markings for safe nets are stored with one bit per state instead of one word per state. In turn, this allowed other optimizations such as increasing the speed of the equality test of marking vectors.
   b. The search of the frontier to discover ω's is omitted, because ω's cannot occur in safe or bounded nets.
5. All the memory for the storage of the net and its reachability graph is dynamically allocated. (For this reason there are no limits on the size of entities other than the operating system limit on storage allocated to a job.)
6. All of the display commands for reachability graph

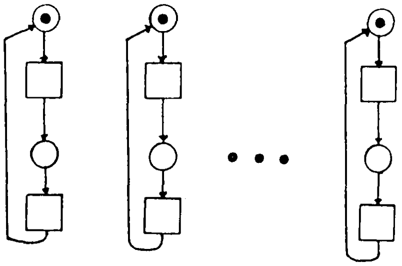


Figure 4. Net schema for performance benchmark.

Processes	Rgraph		-s		-b		-g	
	Nodes	Arcs	Time	Space	Time	Space	Time	Space
09	512	4608	2	296	4	328	11	328
10	1024	10240	4	424	10	504	38	504
11	2048	22528	9	712	25	873	2:25	873
12	4096	49152	22	1306	1:01	1674	9:14	1674
13	8192	106496	50	2556	2:21	3358	36:47	3358
14	16384	229376	1:50	5220	5:31	6952	2:28:53	6952
15	32768	491520	4:05	10863	12:56	14614	9:52:45	14614

Table 1. Performance statistics for a Pyramid 90x.

analysis involve a search of the graph. Usually searches are done with a marking bit at each node to indicate whether or not the node has been visited. This leads to the problem of resetting the marking bit at the beginning of each search, which implies two passes over the graph for each search. To avoid this problem, a global count of the number of searches is kept and for each node the number of times it is visited is kept in the node. At the beginning of a search the global count is incremented; when a node is visited during a search the local count is set equal to the global count. Therefore, unvisited nodes have a local count less than the global count. Although this technique requires more storage it is preferable to resetting bits because it yields faster searches.

5. Experience

The experience described was accumulated over by the author previous to his employment at MCC with a different net analyzer (which he also built). The analyzer described to this point is the one with a new architecture, while the experience is with the old analyzer. Although we have not yet had the opportunity to use the new analyzer, the experience with the old one is valid for discussion.

5.1 Synergy in the Tool Set

Originally net entry was conceived to be an interactive process. Net entry was never interactive; it didn't prove to be useful In dealing with larger nets. (In fact, a command was implemented to delete nodes and arcs, but it was never used.) Instead synergy among the available tools ran rampant during use. The synergy between *petri* and *emc* is just one example. The most used combinations were *petri* and tools for building nets like the *m4* macro processor.

One mode of using *petri* was as follows. One would build macros to generate pieces of a net that are repeated in a problem. For example, if one were modeling the dining philosophers problem, a philosopher macro and a fork macro would be written. This would be done carefully so that the generated net pieces could be merged with the **merge** command. The net was generated by writting the proper macro invocations and merge commands.

The second mode of use was to generate the net from a programming-like language. A language based on Holt's Role/Activity Diagrams [7] was designed and a translator to Petri nets was built. The **rename, merge,** and **split** commands were indispensable in this translation process. There are many reductions possible for Petri nets; however, for the analysis of the nets produced from this programming language, only the reduction that is described for the **reduce** command was found to be useful (which was why it was the only one implemented).

5.2 Reachability Graphs and Debugging Net Designs

The interactive analysis of a net is a very useful feature when showing that a net model satisfies behavioral requirements. When the analysis shows that the requirements are not met, a debugging process then proceeds. There are many subtlies of the command set that facilitate this process.

Let us consider how we might tackle one type of erroneous net. Imagine that we are designing with the intent of having a safe net. If the net is not safe, the analyzer reports on the shortest firing sequence to the unsafe marking (i.e., the shortest that exists in the partially built reachability graph when the unsafe marking is found). The use of the simulation facility with this information may be enough to correct the net. But if it is not, then the computation may be done with either the -s or -g option. This takes more time, but one can interrupt the computation and analyze the partially built reachability graph. This can be done until one has a reachability graph with enough information to correct the problem.

5.3 Actual Use

Despite the power of *petri,* experience shows that its use requires study and creativity. When modeling a system, it is easy to abstract away the properties in which one is truly interested. The model must be constructed accurately. The other side of the coin is that one can easily overwhelm the analyzer (remember the size of the reachability graph can be exponential in the size of the net). One must try to abstract as much a possible. This gives us a trade-off between two conflicting goals.

Two protocols were analyzed with *petri*. Let us call them A and B; space does not permit us to describe them other than to say that protocol A was for the run-time support of a distributed programming language and protocol B was for a packet switching telephony system. In both cases errors were found in the protocol. In the case of A, the protocol was abandoned for other techniques. In the case of B, the designers corrected the protocol. It is significant to note that the second analysis was not done by the author [2].

6. Conclusion

An analyzer for Petri nets was built and used on two protocols. This proved successful in that errors in the protocols were found that might have gone undetected until after the software was released. The analyzer tool fit well with other available tools leading to increased synergy in the tool set.

Another useful aspect of having an analyzer is that it supplies the impetus for people to learn about Petri nets. To an organization involved in the design of software, the benefit is a model of concurrency that is independent of any programming language.

Acknowledgement

I would like to thank Jim Peterson for his comments on this paper.

References

1. Agerwala, T., "Putting Petri Nets to Work," *IEEE Computer Magazine* **12**(12) pp. 85-94 (December, 1979).

2. Ashok, E., *private communication,* ITT, 1 Research Drive, Shelton, CT 06484 (1986).

3. Beaudouin-Lafon, M., "Petripote: a graphic system for Petri-Nets design and simulation," 4th European Workshop on Applications and Theory of Petri Nets, Toulouse, France, September, 1983, pp. 20-31

4. Berthelot, G. and Terrat, R., "Petri Nets Theory for the Correctness of Protocols," *IEEE Trans. on Communications* **Com-30**(12) pp. 2497-2505 (December, 1982).

5. Brauer (ed.), W., *Net Theory and Applications,* Springer-Verlag, New York (1980).

6. Clarke, E. M., Emerson, E. A., and Sistla, A. P., "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications: A Practical Approach," Conf. Rec. Tenth Ann. ACM Symp. on Principles of Programming Languages, Austin, Texas, January 24-26, 1983, pp. 117-126

7. Holt, A. W., Ramsey, H. R., and Grimes, J. D., "Coordination System Technology as the Basis for a Programming Environment," *Electrical Communications* **57**(4) pp. 307-314 (1983).

8. Jantzen, M. and Valk, R., "Formal Properties of Place/Transition Nets," pp. 165-211 in *Net Theory and Applications,* ed. W. Brauer,Springer-Verlag, New York (1980).

9. Jensen, K., Huber, P., Larsen, M. N., and Martinsen, I., "Petri Net Package User's Manual," DAIMI MD-46, Computer Science Department, Aarhus University, Aarhus, Denmark (September, 1983).

10. Martinez, J. and Silva, M., "A Simple and Fast Algorithm to Obtain All Invariants of a Generalised Petri Net," pp. 301-310 in *Application and Theory of Petri Nets,* ed. C. Girault and W. Reisig,Springer-Verlag, Berlin (1982).

11. Memmi, G. and Roucairol, G., "Linear Algebra in Net Theory," pp. 213-224 in *Net Theory and Applications,* ed. W. Brauer,Springer Verlag, Berlin (1980).

12. Montel, B., Grissault, D., Le Mer, E., Robert, C., Sivet, A., Ayache, J. M., Azema, P., Bachmann, S., Berthomieu, G., Chezalviel-Pradin, B., Courtiat, J. P., Diaz, M., and Dufan, J., "OVIDE: A Software Package for Verifying and Validating Petri Nets," Proceedings of Softfair, Arlington, Virginia, July 25-28, 1983, pp. 86-92

13. Peterson, J. L., "Petri Nets," *ACM Computing Surveys* **9**(3) pp. 223-252 (September, 1977).

14. Peterson, J. L., *Petri Net Theory and the Modeling of Systems,* Prentice-Hall, Englewood-Cliffs, N.J. (1981).

15. Razouk, R. R. and Hirshberg, D. S., "Tools for Efficient Analysis of Concurrent Software Systems," *Softfair,* pp. 192-198 (1985).

16. Reisig, W., *Petri Nets,* Springer-Verlag, Berlin (1985).

17. Vuong, S. T. and Cowan, D. D., "Reachability Analysis of Protocols with FIFO Channels," SIGCOMM '83 Symp. on Communications Architectures and Protocols, Austin, Texas, March 8-9, 1983, pp. 49-57

The GTPN Analyzer: Numerical Methods and User Interface

Mark A. Holliday
Department of Computer Science
Duke University
Durham, NC 27706

Mary K. Vernon
Computer Sciences Department
University of Wisconsin — Madison
Madison, WI 53706

Abstract

The GTPN (Generalized Timed Petri Net) is a performance model based on Petri Nets. The state space for a model of the system is automatically constructed and analyzed using results from Markov chain theory. We address some of the key computational issues involved in the Markov chain theory approach. In particular, we discuss two types of algorithms. The first type compute the *absorption probabilities* and *mean time to absorption.* The second type compute the steady state probability distribution for a single, possibly periodic, recurrent Markov chain class. We also describe the GTPN's user interface for input of the model description, execution of the tool, and the output of the performance results.

Section 1. Introduction

Petri Nets are a formal graphical model of asynchronous parallel computation [PET62, PET81]. Modifying Petri Nets so that time is represented has recently been an active research area [MOL81, NAT80, ZUB80, AJM84, HOL85a]. The goal of these models is to analyze system performance and/or reliability as an extension of the reachability analysis. Our model, Generalized Timed Petri Nets (GTPN) [HOL85a], associates firing frequencies and deterministic firing times with each transition in the net. For the purposes of performance analysis, we view the GTPN as a stochastic process. The time-in-state is a deterministic function for each state of the net. However, a probability distribution is defined over the possible next states based on the firing frequencies. Analysis of the embedded discrete-parameter Markov Chain yields performance estimates. [HOL85b and VER86] contain results for multiprocessor performance issues obtained using the GTPN analyzer.

We have used our GTPN analyzer to solve models with up to 45,000 states. An important issue is the computational efficiency of the performance analysis. An earlier paper [HOL85a] describes efficient methods we developed for building the state space for GTPN models. States in the automatically generated Markov Chain are grouped into *transient* and *recurrent* classes. This paper focuses on numerical methods we use for solving the very large (typically $20,000 \times 20,000$), but very sparse (typically 2-3 non-zero entries per row) matrix equations to get the absorption probabilities and equilibrium state probabilities for the recurrent classes in the Markov Chain. For computing absorption probabilities, we use the key observation that the classes in the Markov Chain form a directed acyclic graph (DAG). For computing equilibrium state probabilities, we use the Power Method, with some shifting and scaling of the probability matrix needed for periodic Markov Chains. A second important issue we address is the user interface of the tool.

The organization of the remainder of this paper is as follows. Section 2 presents an overview of the GTPN and outlines the equations which must be solved. Section 3 considers efficient methods of computing the absorption probabilities and mean times to absorption for the recurrent classes. Section 4 considers methods for computing the steady state probability distribution for each recurrent class. Section 5 describes the current user interface of the GTPN. Section 6 summarizes our experiences.

2. The GTPN Model

2.1. The Net

A GTPN is a Petri net which includes: 1) a deterministic firing duration associated with each transition, 2) a mechanism for specifying next state probabilities for conflicting transitions, and 3) a set of named resources associated with transitions and/or places which are used to calculate performance estimates. We plan to add transition *enabling times* [RAZ83] to support modeling behavior such as timeouts in network protocols. This will require some minor changes in the model definition and how the reachability graph is built.

Letting S denote the set of reachable states, $\Re^+$ denote the positive reals, and $\mathcal{P}$ denote the power set, the current GTPN is formally defined by the following tuple:

$$GTPN = (P, T, A, M_0, D, F, C, R)$$

where

$P = \{p_1, p_2, \ldots, p_n\}$	(places)
$T = \{t_1, t_2, \ldots, t_m\}$	(transitions)
$A: \{P \times T\} \cup \{T \times P\} \rightarrow \{0, 1, 2, ...\}$	(directed arcs)
$M_0: P \rightarrow \{0, 1, \ldots\}$	(initial marking)
$D: T \times S \rightarrow \Re^+ \cup \{0\}$	(firing durations)
$F: T \times S \rightarrow \Re^+ \cup \{0\}$	(firing frequencies)
$C: T \rightarrow \{yes, no\}$	(CntComb flags)
$R: P \cup T \rightarrow \mathcal{P}(\{r_1, r_2, \ldots, r_k\})$	(resources)

The first four components of the tuple are identical to the constructs in an untimed Petri Net (see [PET81]). The remaining four components are described below.

Figure 2.1 shows an example GTPN including the initial state distribution of tokens. Each transition is labeled with, from left to right, its firing duration expression, its frequency expression, its CntComb boolean flag, and its list of resources. There are no resources associated with places in this model.

This research was partially supported by the National Science Foundation under grants DCR-8402680 and DCR-8451405, and by grants from IBM and Cray Research, Inc.

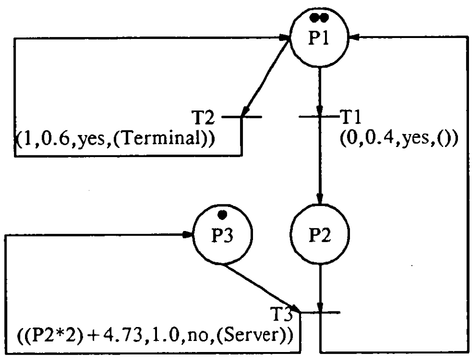


Figure 2.1. Example of a GTPN net

A transition's firing duration and frequency are expressions containing immediate values (real and integer), names of places (which represents the number of tokens in that place in the current state), names of transitions (which represents the value one if at least one firing of that transition is in progress in the current state and is otherwise zero), and arithmetic, relational, and logical operators. Thus, a transition's firing duration and frequency can be state-dependent, but for a given state they are deterministic.

The example in Figure 2.1 models users at terminals who, with a geometric think time, generate requests for a server. There is one token on place P1 for each user. Transitions T1 and T2 implement the think time. Note that the GTPN can represent geometric, as well as constant, holding times. Transition T3 implements a load-dependent server with a firing duration that depends on the number of tokens on P2.

2.2. The Reachability Graph

The *multiplicity* of an input place is the number of arcs from that place to that transition. An output place's multiplicity is defined analogously. An enabled transition ***starts firing*** by removing from each input place a number of tokens equal to its multiplicity. After start firing, the firing is ***in progress*** until ***end firing***. While the firing is in progress, the time to end firing, called the ***remaining firing time*** (RFT), decreases from the transition's firing duration to zero. At end firing a number of tokens equal to its multiplicity is put on each output place.

A marking together with the set of current RFT's defines a *state* of the net. State transitions in the GTPN are defined by maximal sets [1] of start firing or end firing events which occur at the same time. A state which has at least one transition enabled (e.g. the initial state in Figure 3.1), has zero duration, and leads to a set of next states defined by all possible combinations of start-firing events that can occur simultaneously. The new RFT's are set to their transitions' durations. If there are no enabled transitions, but there are some firings in progress, then there is one next state generated by the end firing of all firings in progress with the smallest RFT (*Tmin*). The time-in-state value in this case is *Tmin*. If there are no enabled transitions and no firings in progress, then the net remains in the current state forever.

For each state, a probability distribution is defined over the set of next states. In the nontrivial case, we need to assign a probability to each maximal set of transitions that can start firing together. Calculation of the next state probabilities for start firing events is complex, due to the possibility of conflicting transitions. We only outline our approach and quote the relevant formulas in this paper. The reader is referred to [HOL85a] for a more complete discussion.

Two transitions whose sets of input places intersect are in the same ***conflict set***. The transitive closure of the property of intersecting input places defines an equivalence relation on the set of transitions. This equivalence relation partitions the set of transitions into disjoint sets called ***generalized conflict sets***. A maximal set of start firing events which comprise a state transition is the union of a set of independent ***local maximals***, one from each generalized conflict set. The probability for the maximal is the product of the probabilities for the associated local maximals (since the local maximals are independent). Let *LocalMax[j,i]* denote the *j*th local maximal of the *i*th generalized conflict set. Let *NumComb* denote the ***number of combinations***, or the number of ways tokens can be removed from input places, in order to implement a local maximal. Our formula for *Pr{LocalMax[j,i]}* is

$$Pr\{LocalMax[j,i]\} = \frac{J * \prod_{\{k:k\in LocalMax[j,i]\}} f_k}{\sum_{\{m:m=1,\ldots,M\}} M * \prod_{\{k:k\in LocalMax[m,i]\}} f_k}$$

J is NumComb[LocalMax[j,i]] if *NumComb* is used and zero otherwise. M is NumComb[LocalMax[m,i]] if *NumComb* is used and zero otherwise. In some cases, the value *NumComb* is needed in order to derive an intuitively reasonable probability. The boolean flag CntComb (Count Combinations) associated with each transition specifies whether this should be done. Only if the flag is ***yes*** for all transitions in the maximal, is *NumComb* used.

In Figure 2.2 and Table 2.1 we show the reachability graph for the net in Figure 2.1, assuming there is only one user (i.e. one initial token in P1). The labels on the edges of the graph are the next state probabilities. The labels on the vertices of the graph are the values for time-in-state. The marking vectors

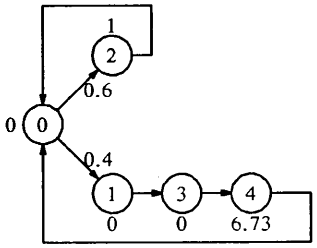


Figure 2.2. Reachability Graph for example

Table 2.1. Reachable States for example

States	Marking			RFT Set	Resources
	P1	P2	P3		
0	1	0	1	{}	{}
1	0	0	1	{(T1,0.0)}	{}
2	0	0	1	{(T2,1.0)}	{Terminal(1)}
3	0	1	1	{}	{}
4	0	0	0	{(T3,6.73)}	{Server(1)}

[1] a set with property α is a ***maximal*** set with property α if it is not a proper subset of any other set with property α.

are shown in the table. The RFT sets are shown as a list of pairs with one pair per in progress firing of a transition. The first component of each pair is the name of the transition. The second component is the remaining firing time. The resources used and their number of uses are also shown in the table.

Section 2.3. Analysis of the Embedded Markov Chain

Our analysis is based on the key observation that the times at which state changes occur form an embedded, discrete-time, finite state Markov Chain. Consequently, we need to sketch some of the relevant terminology and theorems of Markov Chain theory. We are only concerned currently with analyzing models with finite state spaces. The strongly connected components of the state space (when viewed as a directed graph) are the *classes* of the Markov Chain. The condensed graph (one vertex for each class) is a directed acyclic graph with one root. In the case of a finite state space, the interior vertices of the condensed graph are called *transient* classes. The leaf vertices of the condensed graph are called *recurrent* classes. In a typical evolution of the system being modeled, the system starts in a state in the root of this condensed graph. It then filters through the transient classes until it is absorbed by one of the recurrent classes. Once absorbed it stays in that recurrent class permanently.

Since, in the GTPN, we are interested in long run behavior, there are two characteristics of the movement through the transient classes that are of interest. First, is the *absorption probabilities*, the probabilities of being absorbed by each leaf. Second, is the *mean time to absorption.*

The primary approach to computing the absorption probabilities is based on First Step Analysis [TAY84]. Consider a particular initial state, i. On the first state change, the process will move from state i to a state j that is in a transient class or in a recurrent class. If j is in class R, the future probability of being absorbed by class R is one. If j is in another recurrent class, the future probability of being absorbed by class R is zero. If j is in a transient class, then, by the memoryless property, the probability of being absorbed by class R is the same as if j were the initial state.

More formally, suppose that the states in all the transient classes are numbered $0, ..., n-1$ and consider a fixed recurrent class R and fixed initial state i. Let $U_{iR} = Pr\{\text{Absorption in class } R | X_0 = i\}$ for $0 \leq i \leq n-1$.

$$U_{iR} = \sum_{\{j \in R\}} P_{ij} + \sum_{j=0}^{n-1} P_{ij} U_{jR}, \qquad i = 0, 1, ..., n-1$$

This equation cannot be solved in isolation. However, if we consider all possible transient initial states, then we have a system of linear equations that can be solved for the U_{iR}'s.

Once absorbed in a particular recurrent class, the probability distribution over the states in the long run needs to be computed. This probability distribution exists for any recurrent class R in a finite state space as long as it is interpreted as the long run expected fraction of visits in each state. This stationary probability distribution is easy to find since it is the unique solution to the set of equations

$$\pi_R = \pi_R P_R \qquad \text{and} \qquad \sum_{j \in R} \pi_j = 1$$

The matrix P_R is $P_R = \{P_{ij} | i, j \in R\}$.

Section 2.4. Computing Performance Estimates

The embedded Markov Chain has allowed us to obtain, for each recurrent class, the long run expected fraction of visits in each state. We still need to obtain, for each recurrent class, the performance estimate for each resource. This is done in two steps. First, we obtain, for each recurrent class, the long run expected fraction of time, $RelTime(S_1)$, spent in each state S_1, (i.e. count time-in-state, instead of visits). Let S be the set of states in R. As shown in [HOL85a], $RelTime(S_1)$ can be computed by:

$$RelTime(S_1) = \frac{TimeInState(S_1)\pi_{S_1}}{\sum_{k \in S} TimeInState(k)\pi_k}$$

Second, the performance estimate for a resource in a given recurrent class is the long run expected number of usages of that resource in that class. Consequently, we simply take the expectation of the random variable ResUsages (again S is the set of states in R).

$$E[ResUsages] = \sum_{k \in S} ResUsages(k) Pr\{statek\}$$
$$= \sum_{k \in S} ResUsages(k) RelTime(k)$$

The probability distribution of the random variable ResUsages with respect to the number of usages of the resource is also computed. Computing this distribution is straightforward. We know the long run probability distribution of time over the states in the recurrent class. We also know the number of usages of the resource in question for each state. Simply summing the probabilities of the states that use the resource the same number of times generates the desired distribution. This distribution is very useful for reliability prediction (e.g. 90% of the time will at least 3 processors be working?).

Section 3. Absorption Probabilities and Mean Time to Absorption

As discussed in section 2.3, two of the characteristics of the system's performance that we need to determine are the absorption probabilities and mean time to absorption. First Step Analysis was proposed as the method for computing these characteristics. In this section we discuss the computational issues involved in determining those characteristics. Section 3.1 discusses some of the efficiency issues involved in implementing First Step Analysis for computing the absorption probabilities. Section 3.2 describes an optimization which can significantly accelerate solution of the First Step Analysis equations. Section 3.3 covers the analogous material for computing the mean time to absorption. In an important special case that arises frequently in GTPN models, an alternative to First Step Analysis can be used to compute absorption probabilities. Section 3.4 describes this still more efficient method.

Section 3.1. First Step Analysis

Direct application of First Step Analysis implies that r systems of n linear equations are solved where r is the number of recurrent classes, and n is the number of transient states in the Markov Chain. Solving one system of equations determines the absorption probability of interest for one recurrent class. Since the GTPN is intended to be a practical tool, an efficient solution method is important.

We write the systems of linear equations in matrix form as follows:

$$U_R = P_{TR} + P_T U_R, \tag{3.1}$$

where U_R is the $n \times 1$ vector of absorption probabilities for recurrent class R from transient states $0, 1, ..., n-1$, P_{TR} is the $n \times 1$ vector of one-step transition probabilities from transient state i to any state in R, and P_T is the $n \times n$ one-step transition probability matrix for the transient states.

The standard form for a system of linear equations is $Ax = b$ where A is $n \times n$, x is $n \times 1$, and b is $n \times 1$. Our form maps into the standard form by letting $A = (P_T - I)$, $x = U_R$, and $b = -P_{TR}$:

$$(P_T - I)U_R = -P_{TR}. \tag{3.2}$$

There are many methods of solving systems of linear equations. Gaussian elimination is the primary direct method. Iterative methods, such as the Gauss-Seidel method, are much more efficient for large matrices. Before selecting a solution method, however, we will discuss an important optimization that can be applied to First Step Analysis.

Section 3.2 Optimization of First Step Analysis

An optimization exists that can significantly accelerate solution of the linear systems of equations for First Step Analysis. This optimization is based on a key observation: by permuting the rows and columns of the matrix P_T, P_T can be put into block upper triangular form, where each diagonal block represents the transitions within one transient class of the Markov Chain. To see this, recall that the Markov Chain classes are the strongly connected components in the reachability graph. They thus form a directed acyclic graph (DAG), the condensed graph. As with any DAG, the vertices of the condensed graph can be numbered via a topological sort so that the number assigned to a vertex is always less than the numbers assigned to its children. This numbering defines the permutation that generates the block upper triangular form. We find the strongly connected components using Tarjan's O(n) algorithm, and perform the topological sort on the DAG using another linear-time depth-first search [SED83].

After the permutation, equation 3.2 is of the following form:

$$\begin{pmatrix} A_{11} & A_{21} & \cdots & A_{1N} \\ & A_{22} & & A_{2N} \\ & & \ddots & \vdots \\ & & & A_{NN} \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ \vdots \\ U_n \end{pmatrix} = \begin{pmatrix} -P_{1R} \\ -P_{2R} \\ \vdots \\ -P_{nR} \end{pmatrix} \tag{3.3}$$

where the elements are matrices, each diagonal block A_{ii} is square of order n_i and

$$\sum_{i=1}^{N} n_i = n.$$

The system (3.3) can be solved as a sequence of N smaller problems. Problem i is of order n_i and the matrix of coefficients is $A_{ii}, i = 1, 2, ..., N$. The procedure is as follows [PIS84]:

(i) Solve the last subsystem, with A_{NN} as the matrix of coefficients, for the last n_N unknowns. Compute the vector x_N of order n_N.

(ii) Subtract the products $A_{jN}x_N$ from the right-hand side for $j = 1, ..., N-1$. A block upper triangular matrix of order $N-1$ is obtained, and the procedure is repeated until the complete solution is obtained.

Note that the assumption must be made that the diagonal blocks in Equation (3.2) are nonsingular. The solution of each subsystem can be done by any method for solving linear equations, such as Gaussian elimination or the Gauss-Seidel iteration.

Section 3.3. Computing Mean Time to Absorption

The mean time to absorption can also be computed using a first-step analysis. One system of n linear equations is solved where n is the number of transient states. Each transient state i has one equation. Starting in state i, the mean time to absorption is the time-in-state for state i plus zero if the next state is in a recurrent class and plus P_{ij} times the mean time to absorption for state j if state j is transient. Note that the Markov property is again being used. Formally, each equation is of the form:

$$U_i = TimeInState(i) + \sum_{j=0}^{n-1} P_{ij}U_j, \qquad i = 0, 1, ..., n-1$$

If the integer 1 replaces the time-in-state, then the mean number of visits to transient states before absorption can be computed. The mean number of visits to a given set of transient states before absorption can be computed if the time-in-state is replaced by the integer 1 when visiting a state in the given set and is replaced by the integer 0, otherwise.

The discussion in Section 3.2 applies here also. In particular, the system of equations can be changed into the form $Ax = b$, permuted into block upper triangular form, and then solved as outlined.

Section 3.4. More Efficient Solution of an Important Special Case

In a special case that occurs frequently in GTPN models, an efficient alternative approach based on the condensed reachability graph (i.e. one vertex per strongly connected component) can be used to compute absorption probabilities. In this alternative, probabilities are assigned to the edges leaving each vertex in the condensed graph A depth first search is then done. During the depth first search the probability along each path to each leaf is determined by taking the product of the edge probabilities along the path. The absorption probability for a given leaf is then simply the sum of the path probabilities terminating at that leaf.

The difficulty with this approach is with determining the edge probabilities leaving a vertex in the condensed graph. If all the exit edges originate at the same vertex V_1 within the strongly connected component, then the probabilities are easily computed. Find all the edges of which V_1 is the parent that are exit edges. Sum their probabilities as edges in the original graph. The probability of each edge in the condensed graph is its probability in the original graph normalized by this sum.

The problem is when more than one vertex, say V_1 and V_2, in the original graph are parents of exit edges. Determining the probability that each of these parent vertices is the vertex from which exiting occurs is dependent on the detailed structure of the strongly connected component. Our current implementation uses the condensed graph approach when the special case of one parent vertex in the strongly connected component is met. When the special case is not met, first-step analysis with Gauss-Seidel iteration is used. We plan to implement the optimization described in section 3.2 with subsystem solution by Gauss-Seidel iteration.

Section 4. Stationary Probability Distributions

In this section we discuss the method we use to calculate the stationary probability distribution for each recurrent class in the Markov Chain. Let R denote a recurrent class with states $j = 0, 1, ..., n$, and let π_j represent the long run expected fraction of visits to state j, given that the modeled system is absorbed in class R.

Recall that the vector $\pi_R = (\pi_0, \pi_1, ..., \pi_n)$ is uniquely determined by the following equations:

$$\pi_R = \pi_R P_R \qquad \text{and} \qquad \sum_{j=1}^{n} \pi_j = 1, \tag{4.1}$$

where P_R is the $n \times n$ state transition probability matrix for R.

Because the matrix P_R is sparse, but potentially very large, iterative methods are more practical than direct methods. One of the most widely used iterative methods, the Power Method [JOH82], views equation 4.1 as an eigenvalue problem. In particular, π_R is the eigenvector associated with the unit eigenvalue of P_R. Applying the Power Method to iteratively solve for π_R, is done as follows:

$$\pi_R^{k+1} = \pi_R^k P_R.$$

Since our current implementation uses the Power Method, the remainder of this section focuses on issues related to it and, in particular, with ensuring convergence.

Section 4.1. Spectral Distribution

An eigenvalue, λ, of a real $n \times n$ matrix A, is *strictly dominant*, if its modulus is strictly greater than those of all the other eigenvalues of A. Direct application of the Power Method converges to an eigenvector corresponding to the dominant eigenvalue, if and only if the matrix has a simple, strictly dominant eigenvalue.

This constraint on the direct application of the Power Method can be expressed in terms of the periodicity of the recurrent class in the Markov Chain. In order to make that connection between the spectral distribution and the periodicity of the Markov Chain class, we need the theorems of Perron and Frobenius [CIN75,SEN84]. These theorems state the following:

> An irreducible non-negative matrix, A, has a real, positive, and simple eigenvalue, α, which is *greater than or equal to* all other eigenvalues of A in modulus. The eigenvector corresponding to the eigenvalue α is strictly positive. If A is *aperiodic*, (i.e. $A^k > 0$ for some k), then α is *strictly greater than* all other eigenvalues of A. A periodic matrix with period δ, has exactly δ eigenvalues with absolute values equal to α. These eigenvalues are all distinct and are given by:
>
> $$\lambda_k = \alpha[e^{2\pi i/\delta}]^{k-1}, k = 1, 2, ..., \delta, i = \sqrt{-1}$$

If A is a stochastic matrix (i.e. all rows sum to one), then $\alpha = 1$. An aperiodic stochastic matrix thus has a simple unit eigenvalue and all other eigenvalues are of strictly smaller modulus. A periodic stochastic matrix with period δ has exactly δ eigenvalues of unit modulus all of which are simple. These eigenvalues can be regarded as a set of points around the unit circle in the complex plane, which goes over into itself under a rotation of the plane by the angle $2\pi/\delta$.

Section 4.2. Ensuring Convergence

According to the above discussion, the Power Method can only be applied directly to find the stationary probability distribution when the recurrent class is aperiodic. To handle the case of a periodic recurrent class, the following theorem [STE74] concerning shifting and scaling the matrix becomes important:

Theorem: Let A be a complex $n \times n$ matrix, and let λ be an eigenvalue of A with eigenvector x. Then:

1. $a\lambda$ is an eigenvalue of aA with eigenvector x.
2. $\lambda - b$ is an eigenvalue of $A - bI$ with eigenvector x.

This theorem allows us to transform the matrix P_R to ensure that the unit eigenvalue is strictly greater than all other eigenvalues in modulus. In particular, we make the following transformations on P_R:

$$P_R' = \epsilon(P_R - I) + I = \epsilon P_R + (1 - \epsilon)I \qquad 0 < \epsilon < 1.$$

The first transformation, subtracting I, shifts all eigenvalues of P_R to the left by one in the complex plane. The eigenvector π_R now corresponds to the eigenvalue zero. The second transformation, multiplication by ϵ, shrinks all of the eigenvalues except the zero eigenvalue corresponding to π_R (the circle is now centered at $-\epsilon$ and is of ϵ radius). The third transformation, adding I, shifts all eigenvalues to the right by one, creating a unit eigenvalue corresponding to π_R whose modulus is strictly greater than that of all other eigenvalues. Thus, application of the Power Method using P_R' always converges.

Section 4.3. Convergence Rate

The convergence rate of this method is essentially the rate at which $\lambda_2'^k$ converges to zero, where λ_2' is the second largest eigenvalue of the matrix P_R' [JOH82]. The value of ϵ indirectly influences the rate of convergence by scaling all of the eigenvalues. Wallace and Rosenberg [WAL66] define a suitable value for ϵ to be $0.99 \times [max(|P_{ii} - 1|)]^{-1}$. We have used this value, which equals 0.99 for most GTPN models, in our implementation.

In general, the value of λ_2' depends on the size and structure of the particular GTPN model constructed, and convergence rates vary considerably. In particular, the convergence rate may be extremely slow. A recent paper by Stewart and Goyal [STE85], suggests that successive overrelaxation is a better method for solving steady-state equations for continuous time Markov chains. We plan to investigate whether this approach would also be more efficient for the GTPN.

Section 5. The GTPN Analyzer User Interface

This section describes the current user interface to the GTPN tool. This interface has three parts: the format of the model description input to the tool, the process of running the tool, and the format of the output of the performance results. We should note that the nature of this interface is largely independent of the rest of the tool.

Section 5.1. The Input Format

The model description is input in textual form. This text is processed by table-driven lexical and syntactic analyzers. In particular, we use two tools, Scangen and LLgen, that have been developed by others at the University of Wisconsin—Madison. Scangen accepts descriptions of tokens written as regular expressions and generates tables to drive a lexical analyzer. LLgen accepts a context-free grammar specification and generates tables for parsing sentences of the specified language. It generates tables for any LL(1) grammar. During execution of the GTPN tool, the tables generated by Scangen and LLgen are used in conjunction with the parser and semantic routines to construct the internal tables describing the net and its initial state.

```
NET
     P1(1) -- > P2(1): 0, 0.4, yes, ;
     P1(1) -- > P1(1): 1, 0.6, yes, Trans2;
     P2(1), P3(1) -- > P1(1), P3(1):
               (P2*2)+4.73, 1.0, no, Trans3;
RESOURCES
     Terminal: Trans2;
     Server: Trans3;
INITSTATE
     P1(2), P3(1)
END
```

Figure 5.1. Input Format for Example

The input format is quite simple. As an example, in Figure 5.1 is the actual input for the net which is shown graphically in Figure 2.1. There are four reserved words: NET, RESOURCES, INITSTATE, and END. These four words, in that order, divide the input into three sections.

The NET section is a sequence of entries, one for each transition. A transition's entry first lists the transition's input places, the token -- >, and the transition's output places. Each input place has a number in parenthesis which is the number of tokens removed by a single firing. Each output place has a number in parenthesis which is the number of tokens added by a single firing. After the colon, the transition has four remaining fields: its duration expression, its frequency expression, the CntComb flag, and an optional name. A transition needs a name if it is to be referenced in a duration or frequency expression or in the RESOURCES section.

The RESOURCES section has one entry for each resource (performance measure). A resource's entry names the resource and then lists a sequence of names of places and transitions. In a given state the number of usages of a resource are determined by these place and transition names. A place has as many resource usages as there are tokens on it. A transition has as many resource usages as there are firings in progress. A more general specification of performance measures would allow expressions containing arithmetic, logical and relational operators as well as place and transition names. We have not yet implemented this capability, but it appears straightforward.

The INITSTATE section lists the places which contain at least one token in the initial state. Each such place has the number of its initial tokens in parenthesis.

There will be some minor changes to this format when we introduce enabling times.

Section 5.2. Running the GTPN

The GTPN tool is a single executable file. It is invoked by the command line:

```
gtpna -[a-z] file
```

The net description comes from *file*. The results go to standard output. Various optional flags display aspects of the internal state of the tool and are useful for debugging.

Section 5.3. Output Format

The output is also textual. First, general information is listed. This includes the number of places, transitions, resources, and states. Second, for each leaf, the performance results for each resource are given as well as the number of iterations needed for the Power Method to converge. Third, the absorption probability for each leaf is listed. Fourth, for each leaf and for each resource, the probability distribution over the number of resource usages is given.

6. Conclusions

A careful study of implementation issues is essential to the successful development of a modeling tool. We have discussed some of the most important implementation issues in the Generalized Timed Petri Net modeling approach. The first general issue concerns using results from numerical linear algebra to aid in efficient analysis of the state space.

The two important transient characteristics of the state space are the absorption probabilities and mean time to absorption. We proposed using First Step Analysis to compute these characteristics. In this method several systems of linear equations are solved. Each system could be solved immediately by Gauss-Seidel. Alternatively, we suggested a more efficient solution based on permuting the matrix into block upper triangular form and then solving a sequence of smaller problems. In an important special case a still more efficient method based on the graph of strongly connected components can be used.

The important steady state characteristic is the steady state probability distribution within each recurrent class. This distribution is determined by another system of linear equations. We currently solve this system of equations by treating it as an eigenvalue problem and using the Power Method. A direct application of the Power Method, unfortunately, does not converge if the recurrent class is periodic. We discuss how scaling and shifting of the transition probability matrix can ensure convergence without changing the desired eigenvector. We briefly discuss the convergence rate. We plan to investigate whether other methods may be superior to the Power Method.

The second issue is the user interface to the GTPN tool. The most interesting aspect of this interface is the use of a table-driven scanner and a table-driven parser has greatly aided in the processing of the textual form of the input description.

Acknowledgements

Many people have made helpful comments and suggestions during our research. We would like to especially thank Klaus Hollig and Diane O'Leary.

References

[AJM84] M.A. Marsan, G. Balbo, and G. Conte, "A Class of Generalized Stochastic Petri Nets," *ACM Trans. on Computer Systems*, vol. 2, pp. 93-122, May 1984.

[CIN75] E. Cinlar, *Introduction to Stochastic Processes*, Prentice Hall, Englewood Cliffs, NJ, 1975.

[HOL85a] M.A. Holliday and M.K. Vernon, "A Generalized Timed Petri Net Model for Performance Analysis (extended version)," Tech. Rep. #593, Computer Sciences Dept., UW-Madison, May 1985, to appear in *IEEE Trans. on Software Engineering.*

[HOL85b] M.A. Holliday and M.K. Vernon, "Exact Performance Estimates for Multiprocessor Memory and Bus Interference," to appear in *IEEE Trans. on Computers*, also Tech. Rep. #594, Computer Sciences Dept., UW-Madison, May 1985

[JOH82] L.W. Johnson and R.D. Riess, *Numerical Analysis, Second Edition*, Addison-Wesley, Reading, MA, 1982.

[MOL81] M.K. Molloy, "On the integration of delay and throughput measures in distributed processing models," Ph.D. dissertation, Univ. California, Los Angeles, 1981.

[NAT80] S. Natkin, "Reseaux de Petri stochastiques", these de Docteur Ingenieur, CNAM-Paris, June 1980.

[PET81] J.L. Peterson, *Petri Net Theory and the Modeling of Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[PET62] C.A. Petri, "Kommunikation mit Automaten," *Schriften des Rheinisch- Westfalischen Institute fur Instrumentelle Mathematik an der Universitat Bonn,* Heft 2, Bonn, W. Germany, 1962; translation: C.F. Greene, Supplement 1 to Tech. Report RADC-TR-65-337, Vol. 1, Rome Air Development Center, Griffiss Air Force Base, NY 1965.

[PIS84] S. Pissanetzky, *Sparse Matrix Technology.* Academic Press, Orlando, Florida, 1984.

[RAZ83] R.R. Razouk and C.V. Phelps, "Performance Analysis Using Timed Petri Nets," in *Proc. 1984 Int. Conf. on Parallel Processing,* pp. 126-129, August, 1984.

[SED83] R. Sedgewick, *Algorithms.* pp. 428-430, Reading, MA: Addison Wesley, 1983.

[SEN81] E. Seneta,, *Non-negative Matrices and Markov Chains; Second Edition,* Springer-Verlag, New York, NY, 1981.

[STE73] G.W. Stewart, *Introduction to Matrix Computations,* Academic Press, New York, NY, 1973.

[STE78] W.J. Stewart, "A Comparison of Numerical Techniques in Markov Modeling," in *Communications of the ACM,* Volume 21, No.2, February 1978, pp. 144-152.

[STE85] W.J. Stewart and A. Goyal, "Matrix Methods in Large Dependability Models," Tech. Rep. RC 11485, November 1985, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[TAY84] H.M. Taylor and S. Karlin, *An Introduction to Stochastic Modeling,* Academic Press, Orlando, Florida, 1984.

[WAL66] V.L. Wallace and R.S. Rosenberg, "The Recursive Queue Analyzer," Systems Engineering Dept., Tech. Report #2, Univ. of Michigan, Ann Arbor, MI, 1966.

[VER86] M.K. Vernon and M.A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets," to appear in *Performance '86 and ACM SIGMETRICS '86 Joint Conference on Computer Performance Modeling, Measurement and Evaluation,* also Technical Report #618, November 1985, Computer Sciences Dept., UW-Madison.

[ZUB80] W.M. Zuberek, "Timed Petri nets and preliminary performance evaluation," in *Proc. 7th Annual Symp. on Computer Architecture,* pp. 88-96, 1980.

SECURITY AND PRIVACY REQUIREMENTS IN COMPUTING

Rein Turn

California State University, Northridge
Northridge, CA 91330

ABSTRACT

Advances in computer technology are profoundly affecting and changing the functioning of societies worldwide, and are generating a variety of nontechnical concerns about computer applications. Among these are information security, privacy protection, and societal resiliency. This paper briefly describes the security and privacy requirements in system design and use, and surveys the current efforts for meeting these requirements. The paper points out that it is important to consider security and privacy issues in all phases of a system's life cycle, as well as to familiarize computer professionals with the societal impacts of computer uses.

I. INTRODUCTION

Recent advances in computer technology are making available, at acceptable cost, computer systems with virtually unlimited processing power, storage capacity, and capability for data communication. The very large scale integration (VLSI) technology permits placing on single chips tens of thousands of logic circuits or millions of bits of memory, and to mass-produce these for near-negligible cost[1]. It is economical, therefore, to maximize the use of computer technology in any system. Such use produces old products and services in new, digital form, and engenders new services or products. Examples include the digital telephone, expert systems[2], various electronic funds transfer systems (EFTS), smart cards[3], embedded microprocessors for controlling other systems, and computer vision[4].

The new computer technology and its applications are profoundly affecting and changing the functioning of societies worldwide. Dependence on computers is increasing, and they are becoming critical components in systems that will dirctly affect our future as well as the daily life. This means that the traditional computer system design requirements, such as high performance and reliability, software portability, system interoperability, and easy maintainability must be strengthened and augmented by new requirements such as system safety, system and data security, privacy protection for personal information, and preservation of societal resiliency. The purpose of this paper is to provide an overview of these requirements, and to discuss the design issues they raise.

II. SECURITY AND SAFETY REQUIREMENTS

A general concern of the users of any system, or of people directly affected by the operation of a system, is its trustworthiness: Is it available when needed? Will it function correctly and reliably? Is it safe to use? Is it protected against misuse or tampering? For a simple system such as a chain saw, it may be possible to answer these questions by inspection at the time of use. For complex systems, such as an air traffic control system, the designers and operators must be trusted to have implemented the appropriate safeguards and requirements.

The safety of computer-controlled real-time systems is receiving increased attention[5]. The principal goal is to prevent physical harm to the users of the system, to anyone or any property within the system's domain of operation, or to the system itself and its operators. Concerns here go beyond the reliability of the controlling computer system alone -- while the latter is recovering, the controlled system must not be unsafe. Thus, system safety encompasses the total system, including its human operators and users, and even other systems that may be encountered. Safety preservation techniques must be developed and integrated into all phases of the system's life cycle, especially the life cycles of the system's software, data bases, and interfaces with users or other systems.

Security requirements.

Computer security is usually defined as the: "protection of the system and the data stored therein against unauthorized access, modification, destruction or use, and against actions or situations that deny authorized access or use of the system." While accidental events that threaten security are included, the emphasis is on deliberate attempts to weaken security. The need for computer system security should be self-evident in view of the large body of evidence that attacks against computer systems or the information therein have occurred[6,7].

In general, the following are the principal reasons for providing access control and security in computer systems and applications[8]:

Protection of resources: The computer sytem, and more importantly, the data stored and processed therein, tend to be critically valuable

CH2345-7/86/0000/1106$01.00 © 1986 IEEE

to an organization's functioning. Some data in the system (e.g., financial accounts) directly represent tangible value and must be protected against unauthorized alterations.

Mandated by law: Several countries have enacted laws that require protection of certain categories of information, such as national defense information or personal information; other laws may require that the integrity of financial data be ensured.

Maintain management control: A goal for any organization is to be in full control of the computer resources and their use.

Ensure safety and integrity: Security is an essential prerequisite for the system safety and for the integrity of the data bases and processes being used (e.g., in computer-aided design or in computer-based modeling).

Operational advantages or economies: A secure computer system is important in organizations whose business depends on their customers' trust (e.g., financial institutions). It can increase the organization's advantage over competitors with less secure systems. It can also reduce operating costs such as insurance premiums.

Despite of the convincing rationale, many systems employ only a minimal set of security safeguards. One reason for this is that the operating system software provided by vendors is still not capable of preventing unauthorized access -- security has not been a design objective. While operating systems generally perform their intended functions correctly, experience has shown that they contain design or implementation flaws and shortcuts that permit any existing access control mechanisms to be bypassed and, thus, cannot prevent unauthorized access and use[9].

The theory of designing truly secure operating systems appears now to be well in hand and meaningful design criteria exist[10], but applying these in practice is a different matter. Secure operating systems will require radical changes in the currently used hardware and software architectures, and it will be very difficult to maintain compatibility with the existing systems and software. Vendors and their customers are reluctant to develop new systems which, even if they provide much better security, require expensive conversions of the existing software and data bases. Many users would accept the uncertainty of suffering large losses in the future due to lack of security rather than making even relatively modest expenditures for security now.

Managers who do want to improve security find a sparse market place: a few access control software packages for managing passwords[11], some devices for controlling dial-up access to thwart hackers, and various anchoring devices for personal computers. With a few exceptions (e.g., Honeywell's MULTICS and SCOMP[12]), there are no secure operating systems available.

Security Risks.

Not all systems require a high level of security, and different approaches can be taken to achieving a desired degree of secureness. The determination of security requirements for a given system, and the selection of appropriate security mechanisms are a part of the risk management activity. The basic steps are[13]: value and criticality analysis, vulnerability analysis, threat identification, risk analysis, risk assessment, security safeguards selection and implementation, development of contingency plans, and effectiveness reviews. Again, these steps may be difficult to apply in practice. For example, it is difficult to place value on information, discover all vulnerabilities, or develop threat scenarios for exploiting these vulnerabilities.

Security risk analysis requires establishing for each threat its probability of occurrence over a specified time period (e.g. a year) and the amount of loss that would be incurred. These quantities can be multiplied to obtain the annual loss expectancy (ALE), as described in[14]. However, since very little actuarial information is available about threats to computer systems, computations of the ALE's is usually based on the analyst's subjective judgement. A thorough review of risk analysis techniques is found in[16].

The operational environment is also taken into account. Thus, in general, security risks are higher in systems where: (1) the "distance" is large between the highest information sensitivity or processing criticality level in the system, and the lowest trustworthiness level of the users; (2) user capabilities in the system include assembly language programming rather than being restricted to higher-level language (HOL) programming only (systems where users are limited to predefined transactions are even less risky); (3) system architecture is complex, such as in the case of multiprocessors or in distributed data processing systems (the risk is even higher in computer networks); and (4) there is considerable uncertainty over the trustworthiness of the system software developers (the situation worsens when hardware developers' trustworthiness is also in doubt).

The result of risk analysis is a list of threats ordered by some severeness measure (such as ALE). In the risk assessment step, decisions are made about each threat: either to accept the risk due to that threat or to reduce the risk by selecting and implementing a set of additional security mechanisms. However, in practice the granularity of strengthening security is much more coarse and, correspondingly, security cost increases are also in much larger increments that might be expected. In addition, tradeoffs may need to be made with system design objectives or requirements other than cost, such as functionality, performance, resource sharing, reliability, and user-friendliness.

Security Policies.

The conceptual elements of a security system are: (1) a security policy which establishes the security framework for the system; (2) prevention mechanisms which provide isolation, identification and authentication, and access control; (3) active protection mechanisms which perform monitoring and response functions; (4) integrity assurance procedures for the security mechanisms; (5) backup and recovery mechanisms; and (6) deterrence provisions (e.g., the existing computer crime laws).

In general, a security policy of an organization or a system defines the data protection requirements, establishes the information sensitivity levels and access categories, defines access criteria and protection requirements, and establishes authorization and accountability systems. A formal model of the access control rules defined by the policy is developed to permit formal proofs of the correctness of the design, implementation, and enforcement of the policy in the computer system software[16]. This model is called the "reference monitor", and it is implemented in the operating system as the "security kernel"[17].

The security policy model of the U.S. Department of Defense (DoD) for protecting national defense information can be used as an illustration[18]. The objective is to permit multilevel secure (MLS) operation where the system permits concurrent storage and processing of information with different degrees of sensitivity (security levels and categories), and permits concurrent access by users with different authorizations (security clearances and categories), but prevents any unauthorized accesses. In this security mode, full resource sharing is possible and the system can be used efficiently. The cost is in developing a secure and trusted operating system. The DoD multilevel security model may be stated as follows.

The system contains protected entities, called "objects",0, and active entities that seek access to the objects, called "subjects", S. Each object is assigned a "security level", SL(0), and membership in a set of access control categories, CAT(0). Each subject has been granted a "clearance level", CL(S), and access to a set of categories CAT(S). The security and clearance levels form a hierarchical system. Top secret, secret, confidential, and unclassified (TS, S, C, and U, respectively) are used in the DoD. The access categories are unordered and may exist at any security level. Two main access modes of subjects to objects are defined: read (observation but no modification), and write (no observation, appending of new information only).

The policy has a mandatory (nondiscretionary) part, and a discretionary part. The latter is a need-to-know access control policy enforcement which applies after the mandatory policy requirements have been satisfied. A model of discretionary access control is based on the access control matrix and on access rules which may be implemented by using "capabilities" or "access control lists". The mandatory (nondiscretionary) security policy rules for granting access are: (1) the simple security condition -- a subject is granted "read" access to an object if and only if $CL(S) \geq SL(0)$ and $CAT(S) \supseteq CAT(0)$; and (2) the "star-property" (*-property) -- a subject is granted "write" access (without being able to read) if and only if $CL(S) \leq SL(0)$ and $CAT(S) \subseteq CAT(0)$. The purpose of the *-property is to prevent copying of information from objects at a higher SL into objects at a lower SL by untrusted software or users.

The mandatory access rules must be invoked by the reference monitor mechanism in the system each and every time access is sought to an object. In order to compare the security and clearance levels, every object and subject in the system must be labelled. Labels must be unforgeable and the labelling and label transfer mechanisms must be proven correct -- they must be trusted. An additional security requirement not explicit in the above model is "confinement" -- prevention of information leakage to unauthorized subjects via unconventional, "covert" information flow channels.

Security policies in the private sector are less precisely defined and tend to emphasize data integrity and denial of services rather than data disclosure. However, the DoD security policy and systems that implement it may also be suitable for use in commercial applications[8,19].

A recent development which may strongly influence security policies and practice in the private sector is the National Security Decision Directive 145 issued in 1984 by the President. This directive appoints the National Security Agency to be the lead agency for computer security on a national scale, and broadens the mission of the National Computer Security Center to provide security training and education, risk analysis assistance, and access to trusted system technology also to the private sector. It also defines two new sensitivity categories: "sensitive but unclassified government or government-derived information, and "sensitive non-government information."

Security Evaluation Criteria.

In 1983, the U.S. DoD Computer Security Center published the trusted computer system evaluation criteria[10] which have become a de facto framework for evaluating system security, especially the operating system software security. The criteria emphasize policy enforcement, users' accountability, correctness assurance, and system documentation. Based on these, the following system of security divisions is defined:

Division D, -- minimal protection: Systems in this class have been evaluated, but fail to meet any security requirements.

Division C, -- discretionary protection: Systems in this division provide discretionary protection and accountability of the subjects for their actions.

Division B, -- mandatory protection: In this division, all systems contain a Trusted Computer Base (TCB) that preserves the integrity of security labels and uses them to enforce the mandatory access control policy. The implementation of the reference monitor concept must be demonstrated.

Division A, -- verified design: Systems in this division have been subjected to formal verification to assure that the mandatory and discretionary security controls in the system can effectively protect sensitive information.

The concept of a "trusted computing base" is defined in the criteria document as follows: "TCB is the totality of protection mechanisms within a computer system -- including hardware, firmware, and software -- the combination of which is responsible for enforcing a security policy. It creates a basic protection environment and provides additional user services required for a trusted computer system. The ability of a TCB to correctly enforce the security policy depends solely on the mechanisms within the TCB and on the correct input by the system administrative personnel of parameters (e.g., a user's clearance level) related to the security policy."

Security Mechanisms.

Security policies can be implemented in several ways, using combinations of physical isolation, logical isolation in operating system design and in hardware architecture, or administrative procedures. This yields a hierarchy of systems that provide different levels of security at different levels of confidence as specified, for example, in the DoD trusted systems evaluation criteria. The standard modes of secure operation are: dedicated system, system-high security mode (all users are cleared to the highest security level of data or processing in the system), and multilevel secure (MLS) mode. In an MLS system, the operating system software is proven to be correctly designed and implemented to handle security and clearance level labels correctly and reliably, and to perform the reference monitor function correctly and reliably, such that no unauthorized accesses, overt or covert, can occur. The state of the art of operating system security, especially verification of design and implementation correctness, has not yet progressed to the point where general purpose MLS operating systems are available[20].

Security mechanisms are hardware or software features which implement the trusted computing base by providing isolation, identification and authentication, and access controls. In conventional operating systems they provide password validation, memory protection, memory management, and user/supervisor domains separation. In the future trusted operating systems they will implement the reference monitor model and features necessary for enforcing the system's security policy[21]: unforgable security labels, TCB protection (isolation of the TCB from subjects by use of privileged operating modes of the and virtualizing the system's resources), confinement of covert channels, and maintenance of security audit trails.

While the above are not all the architectural features necessary for the implementation secure operating systems, as per DoD security evaluation criteria, they do form an important set of design requirements that must be incorporated. In more complex architectures such as distributed systems, there are additional concerns regarding the security of interprocessor communications (typically using a local area network or LAN). Applicable security mechanisms include encryption[22], and architectural approaches for achieving MLS operation based on the use of a secure LAN and processors dedicated to specific security levels[23]. Security measures for computer networks are also based on the use of encryption[24] and secure protofor the various communications functions[25].

Finally, one must not overlook the interactions of security requirements and mechanisms with the other system design requirements. Of special interest is fault-tolerance[26], where certain software-based techniques for "corrective" redundancy may be incompatible with security requirements. In using corrective redundancy, errors in computation due to a hardware fault are corrected by repeating the computation. However, it may not be possible to correct a security violation in this manner.

III. PRIVACY PROTECTION

Privacy is a concept with many meanings. In this paper it is defined as the rights of individuals regarding the collection, storage, processing and use in decision making of personal information about themselves. It becomes an issue in computer applications in the context of computerized personal information record-keeping systems. While privacy was also a problem in manual systems, modern computer-communication technology makes it economical to store and process large volumes of data, permits complex correlations at high speed, allows high-speed access from distant locations and, thus, makes technically feasible for physically decentralized systems to become centralized "logically".

Centralization, whether physical or logical, lays the groundwork for integration of data records and assembly of personal information dossiers on individuals. This is viewed by the public as a threat to their liberties. There are other problems, too. Since information in stored and processsed in computers is not directly readable by humans, they cannot determine without the services of the record keeper what information about them is stored. Further, in the computer system, undetected hardware and software errors can cause information distortions, and information can be altered without detection by accident or deliberately.

Privacy Protection Principles.

Privacy protection is a societal policy and value which must be balanced with other policies and values and, in system design, with other requirements. It is clear that record-keeping on individuals is necessary when privileges are granted (such as the driver's license) or qualification for some benefits is determined. In these cases, the individual voluntariliy foregoes some of his privacy in order to receive the privilege or benefit. This is in the best interest of the society. On the other hand, the society must also protect the individual against excesses of the record keepers and against unfair decisions.

Since the government at its various levels from national to local is a major record-keeping entity, much of the concern about privacy protection centers on its activities. However, in the private sector, too, are large collections of personal information on education, health, employment, financial status, purchases, and life style[27,28]. On the international scale, operations of multinational corporations and international data processing service bureaus has resulted in transborder data flows (TDF) of personal information[29,30].

The principal mechanisms for ensuring privacy protection to individual data subjects are legislative and administrative, rather than technical. Often this distinction is misunderstood -- threats to individual privacy arise mainly from authorized users of the system, rather than from unauthorized users. Thus, privacy protection legislation is needed to keep the authorized users from making unauthorized use of personal information.

National level privacy protection legislation (or data protection, as it is commonly called in Europe) is now in force in a dozen countries: Austria, Canada, Denmark, Federal Republic of Germany, France, Israel, Luxenbourg, New Zealand, Norway, Sweden, United Kingdom, and the United States. Legislation is pending in several other countries (e.g., Australia, Belgium, Finland, Italy, Japan, the Netherlands, Portugal, Spain and Switzerland). Although the implementation approaches and protection scope tend to vary from country to country reflecting different cultural environments and legal traditions, the privacy rights granted are remarkably similar.

The principles for privacy protection have evolved over the last decade, beginning with several national studies, advancing with the early national privacy or data protection legislation (as in West German province Hessen and in Sweden), and arriving to the current form in the OECD Guidelines[31]. The Code of Fair Information Practices formulated by a U.S. Government advisory committee on privacy[32] was the initial foundation for these principles: (1) openness (no secret record-keeping systems, uses, or practices); (2) individual access (the right of individuals to know what data are kept about them, and how they are used); (3) individual participation (the right to correct or amend erroneous records); (4) collection limitation (restrictions on data types that may be collected, and on the collection methods); (5) use limitation (restrictions on the use of data for unannounced purposes); (6) disclosure limitation (restrictions on any external circulation of personal data); (7) information management (requirements to maintain data quality and confidentiaity); and (8) accountability (clearly fixed responsibility for compliance with privacy protection requirements).

Privacy protection efforts in the United States have developed along three lines: the federal government, state and local governments, and the private sector. Federal-level privacy laws, especially the Privacy Act of 1974, apply to the federal government agencies, but with law enforcement and intelligence communities exempted. They also apply to the private sector in financial credit reporting, to educational institutions that receive federal support, and to government access to individuals' banking transaction records. The states have enacted numerous privacy protection laws that cover government agencies and also some private sector business, addressing one or more of the following: employment records, financial credit reporting, insurance and medical records, law enforcement and criminal justice records, EFTS and cable television, and the use of polygraphs.

The future trend in the U.S. is for more extensive protection, even though it is not likely that any new federal-level privacy protection law will be enacted soon; states are likely to be more active. Thus, the private sector record-keeping systems are likely to remain unregulated on the federal level for some time, despite the Privacy Protection Study Commission's recommendations[33], and the general perception of the U.S. public that the threat to privacy is increasing. There is one exeption, however. The Electronic Communication Privacy Act of 1985 is likely to be enacted in 1986. This law would protect messages in electronic mail systems, and would limit the use of various electronic surveillance devices (e.g., as discussed in[34]).

The Impact of New Technologies.

The public concern over threats to privacy is further supported by the emergence of so-called "new technologies" based on the use of computers: computer networks, electronic mail, office automation, electronic funds transfer systems (EFTS), smart cards, interactive home services, and embedded microprocessors as controllers in other systems. Collectively, these applications tend to have a set of common attributes or modes of operations which may increase their potential for adverse impacts on privacy protection, since they potentially support: (1) automated services that generate large volumes of transactions involving individuals, and keep records on them; (2) automated techniques and systems for collecting and transmitting computer readable personal information; (3) direct or indirect integration of systems which handle personal information; (4) applications and services that allow inferring personal information; (5) automated decision-making based on personal information about individuals; (6) physical or information surveillance of individuals; and (7) overt or covert commercial markets for personal information.

The above features of the new computer technology applications set the stage for potential privacy protection problems. For example, computers are being connected into networks, and networks into super-networks, at a rapid rate. The benefits of this for data communication are obvious, even though the resulting systems contain multitudes of complex, hard-to-trace communication paths which contribute to problems in providing security, access control, and message integrity and authenticity.

From the privacy protection point of view, computer networks where personal information data bases are on-line can support de facto integration of record-keeping systems and, thus, the capability for "virtual dossiers" and extralegal exchanges of personal information. Networking will also enhance matching of personal information files in different systems for investigative purposes[35,36], increase the difficulty in monitoring compliance with privacy protection requirements, and render misuse of personal data bases more difficult to detect. Similar privacy protection problems arise in other applications of the new technology.

Privacy Protection and TDF.

Privacy protection is also an important transborder data flow (TDF) issue. Some countries with data protection laws are concerned that less privacy protection will be given to personal infomation about their citizens when transmitted abroad, especially to countries with less comprehensive data protection laws than their own. However, some countries that provide international data processing services or operate private networks tend to view these concerns as of little merit and, instead, promote the principle of "free flow of information". These countries, mainly the United States, are also aginst the concept of providing privacy protection to information about legal persons, such as corporations.

Standardization of privacy protection requirements, such as acceptance and implementation of the OECD privacy protection guidelines, is one approach to resolving these differences. The Council of Europe convention of privacy protection[37] is another effort in this durection among the European countries.

Technical Implications of Privacy Protection.

A technical consequence of the privacy protection requirements in the design and operation of computerized record-keeping systems is the incorporation of new functions not normally needed. These include: (1) preparing notifications of the system's functions and procedures in using personal procedures for inspections, challenges, reviews, information; (2) providing facilities and and submission of corrections or rebuttals by individuals; (3) accounting for, and auditing of the collection, use, and disclosure of personal information, and interactions with the data subjects; (4) maintaining data quality, confidentiality and security; and (5) demonstrating compliance with protection requirements.

Collectively, these technical requirements imply more computational tasks to be performed, and more data storage resources to be used. For example, the Privacy Act of 1974 requires that "...agencies shall maintain all records with such accuracy, relevance, timeliness, and completeness as is reasonably required to assure fairness to individuals in determinations". This calls for the following policy decisions: selection of data items to be used (relevance), the level of detail of information items (precision), the retention time (timeliness), and criteria for verifying accuracy of factual and evaluative information. In addition, mechanisms must be provided for assuring authenticity of the data items, for access authorization, and for revalidation or purging of data items.

The above, in turn, call for error control in data collection and entry, reliable identification of individuals, maintaining data integrity in the system, providing additional data fields in records for privacy protection purposes, operating privacy protection related audit trails, implementing in the systems data security safeguards and access control mechanisms, and adequate provisions for system backup and recovery.

Data security requirements in national data protection laws, and in international agreements, provide another example. The Council of Europe convention provides that: Appropriate security measures shall be taken " ... against accidental or unauthorized destruction or accidental loss, as well as against unauthorized access, alteration or dissemination". In addition, the convention requires that specific security measures be provided for every file; that the degree of vulnerability, need to restrict access, and requirement for long-term storage be considered; and that the current state-of-the-art security measures, methods, techniques be used.

Concluding this section, it may be observed that privacy protection continues as a concern in many countries, and that privacy protection principles are well-formulated and implementable. Since the technical aspects of privacy protection requirements are substantial, they must be considered early in the system's design phase and maintained throughout its life cycle.

IV. SOCIETAL RESILIENCY

In an information society, production and distribution of information is central to the economic, political, and social life. The benefits that accrue through the availability of information drive the society to obtain even more information. This leads to extensive automation of the information collection, processing, and dissemination

tion collection, processing, and dissemination functions. Examples are the decision support systems in business firms, computer-aided design and manufacturing systems, automated process control, office automation, electronic funds transfers, military command and control systems, and many others. It would be very difficult to operate in a modern society without computerized information systems.

As early as in the mid-1960s when the total computer population of the world was a few tens of thousands, concerns were voiced over the increasing dependence on computers[38]. Now, when the computer population of world is in tens of millions, and extensive computer networks exist, they are beginning to be regarded as a new, potentially serious technological vulnerability of the society.

In 1979, Sweden released a report of its Committee on the Vulnerability of Computer Systems (SARK)[39], which concluded that "vulnerability is unacceptably high in today's computerized society". Responding to this, the OECD held a workshop in 1981 on computer vulnerability[40]. Its conclusion was that there are sufficient reasons to justify concerns. In the United States, the American Federation of Information Processing Societies (AFIPS), reacted to the Swedish report by establishing a panel to examine the applicability of SARK conclusions to the United States. The panel observed that the SARK findings were not representative of the situation in the U.S., and that this country still appears to be adequately resilient[41].

An analysis of societal vulnerabilities and potential hazards due to massive use of computers is similar to performing a technology assessment or a risk analysis. Vulnerabilities and threats are identified and their effects are postulated. Among the effects identified by the AFIPS panel as sufficiently severe to cause society-wide problems were: (1) severe disruption of the national economy, and large losses; (2) large decline in the productive capability of an important industrial sector, or massive wasting of scarce resources; (3) severe erosion of citizens' rights and freedoms, (4) sharply increased dependence on foreign powers in economy, finance, or politics, and (5) a coup d'etat, conventional war, or a nuclear conflict.

If it is not possible for computer systems in a given country to directly cause, or to contribute to the occurrence of such harmful effects as listed above, then it would appear that there are no serious computer-related vulnerabilities in that country. On the other hand, some computer application may contribute indirectly by creating potential vulnerabilities which then may cause harmful societal effects. Examples are command and control systems, real time process control systems, systems for distributing goods and services, personal information record-keeping systems, elevctronic funds transfer systems, and management information systems.

Through a sudden unavailability of a sufficient number of computer systems a societally paralyzing chain of events may be triggered. For example, a coordinated attack on computer systems may be launched by terrorists, anti-social elements may embark on a long-term effort of subversion or sabotage, or computers may be damaged through some natural or man-made event (e.g., radiation or electromagnetic pulse from a nuclear accident in earth orbit[42]). A paralyzing situation may also develop gradually, such as the accumulation in data bases of erroneous information which results in wrong decisions and, over time, renders important computer systems unusable. Deliberate contamination of data bases or software could result in similar problems.

Resiliency is the ability to absorb disruptions and damages without suffering long-lasting or irreversible ill effects. In the present context, it is the ability to recover from computer system failures or from their misuse without society-wide harms. Resiliency may be due to certain intrinsic attributes of the society in question, or it may be achieved by deliberate actions of the users, the system designers, and the governmental agencies which may be involved. The following factors appear to affect resiliency: (1) geographic and demographic aspects of the society (large countries are likely to be more resilient); (2) degree of multiplicity and redundancy in providing critical services to the society; (3) degree of service-level and society-level preparedness, contingency planning, and backup; (4) extent of preservation of the "corporate memory" of how functions were performed and supported prior to automation; and (5) existence of legal safeguards against misuses of computer systems (e.g., such as privacy violations or computer crime).

Some of the factors listed above are depend on the size, location, culture, history, and the geopolitical situation of a country. Others are controllable and achievable by appropriate policies. Thus, a computerized country is not necessarily a vulnerable one. On the other hand, resiliency cannot be expected to be a permanent condition. Deliberate efforts are required by all sectors of the society to preserve resiliency. For example, public awareness of the capabilities as well as limitations of computer technology must be increased, industry associations must assume a role in maintaining resiliency in their specific functional areas, and the government must promote resiliency as a part of its national information policy. Individual organizations that are dependent on automated systems must plan for quality control in their data processing operations, provide for adequate security, establish and test contingency plans, and take steps to maintain the corporate memory of how to perform critical functions without computer support.

V. CONCLUDING REMARKS

This paper has striven to show that the traditional technical requirements must be augmented with nontechnical requirements in the development of computer systems for applications that impact public safety and individual rights. Information security and protection of individual privacy are

Information security and protection of individual privacy are two such requirements. Providing these protections involves a number of technical and administrative measures which may reduce the computer system's performance, limit user access, and add to the overhead in general. However, this is the price that must be paid in designing, implementing and operating systems in a societally responsible way. Warnings about the increased societal vulnerability due to massive computerization must also not be taken lightly, since the potential for highly disruptive threats is increasing. For example, international terrorism is likely to be directed against computer systems in the future at a much greater scale than in the past.

REFERENCES

[1] Christiansen, D. (Ed.), " Technology '85", IEEE Spectrum, January 1985, pp. 34-95.

[2] Feigenbaum, E.A. and P. McCorduck, The Fifth Generation: Artificial Intelligence and Japans Computer Challenge to the World, Addison-Wesley, Reading, MA, 1982.

[3] Weinstein, S.B., "Smart Credit Cards: The Answer to Cashless Shopping", IEEE Spectrum, February 1984, pp. 43-49.

[4] Fu, K.-S. and A. Rosenfeld, "Pattern Recognition and Computer Vision", IEEE Computer, October 1984, pp. 274-282.

[5] Leveson, N., "Software System Safety", Proceedings, 1985 NATO Advanced Study Institute, Durham, U.K., Springer Verlag, New York, 1986.

[6] Parker, D.B., Fighting Computer Crime, Charles Scribner Sons, New York, 1983.

[7] Norman, A.R.D., Computer Insecurity, Chapman and Hall, London, 1983.

[8] Turn, R., "Private Sector Needs for Trusted/ Secure Computer Systems", AFIPS Conference Proceedings, Vol. 51: 1982 National Computer Conferencen, AFIPS Press, Reston, VA, 1982, pp. 449-460.

[9] Linde, R.R., "Operating System Penetration", AFIPS Conference Proceedings, Vol. 44: 1975 National Computer Conference, AFIPS Press, Reston, VA, June 1975, pp. 361-368.

[10] Department of Defense Trusted Computer System Evaluation Criteria, CSC-STD-001-83, National Computer Security Center, Ft. Meade, MD, 15 August 1983.

[11] Eloff, J.H.P., "Selection Process for Security Packages", Computers & Security, November 1983 pp. 256-260.

[12] Fraim, L.J., "SCOMP: A Solution to the Multilevel Security Problem", IEEE Computer, July 1983, pp. 26-34.

[13] Campbell, R.P., and G.A. Sands, "A Modular Approach to Computer Security Risk Assessment" AFIPS Conference Proceedings, Vol. 48: 1979 National Computer Conference, AFIPS Press, Reston, VA, June 1979, pp. 293-303.

[14] Guidelines for Automatic Data Processing Risk Analysis, FIPSPUB 65, U.S. National Bureau of Standards, Washington, DC, August 1979.

[15] Neugent, W., et al., Technology Assessment: Methods for Measuring the Level of Computer Security, NBS SP 500-133, U.S. National Bureau of Standards, Washington, DC, October 1985.

[16] Cheheyl, M.H., et al., "Verifying Security", ACM Computing Surveys, September 1981, pp. 279-339.

[17] Ames, S.R., Jr., M. Gasser, and R.R. Schell, "Security Kernel Design and Implementation: An Introduction", IEEE Computer, July 1983, pp. 14-22.

[18] Landwehr, C.E., "Formal Models for Computer Security", ACM Computing Surveys, September 1981, pp. 247-278.

[19] Lipner, S.B., "Non-Discretionary Controls for Commercial Applications", Proceedings, 1982 Symposium on Security and Privacy, IEEE Computer Society, 1982.

[20] Landwehr, C.E., "The Best Available Technologies for Computer Security", IEEE Computer, July 1983, pp. 89-100.

[21] Landwehr, C.E., and J.M. Carroll, "Hardware Requirements for Secure Computer Systems: A Framework", Proceedings, 1984 Symposium on Security and Privacy, IEEE Computer Society, pp. 34-40.

[22] Meyer, C.H., and S.M. Matyas, Cryptography -- A New Dimension in Computer Data Security, John Wiley & Sons, New York, NY, 1982.

[23] Rushby, J., and B. Randell, "A Distributed Secure System", IEEE Computer, July 1983, pp. 55-67.

[24] Davies, D.W., and W.L. Price, Security for Computer Networks, John Wiley & Sons, New York NY, 1984.

[25] Voydock, V.L., and S.T. Kent, "Security Mechanisms in High-Level Network Protocols", ACM Computing Surveys, June 1983, pp. 135-171.

[26] Avizienis, A.A., "Fault-Tolerance: The Survival Attribute of Digital Systems", Proceedings of the IEEE, Oct. 1978.

[27] Westin, A.F., Computers, Health Records, and Citizens' Rights, NBS Monograph 157, Government Printing Office, Washington, DC, December 1976.

[28] Westin, A.F., Computers, Personnel Administration, and Citizens' Rights, NBS SP 500-50, Government Printing Office, Washington, DC, July 1979.

[29] Turn, R. (Ed.), Transborder Data Flows: Concerns in Privacy Protection and Free Flow of Information, AFIPS Press, Reston, VA, 1979.

[30] Transborder Data Flows and the Protection of Privacy, ICCP-1, OECD, Paris, 1979.

[31] Guidelines on the Protection of Privacy and Transborder Flows of Personal Data, OECD, Paris, 1981.

[32] Records, Computers, and the Rights of Citizens, Government Printing Office, Washington, DC, July 1973.

[33] Personal Privacy in an Information Society -- Report of the U.S. Privacy Protection Study Commission, Government Printing Office, Washington, DC, July 1977.

[34] Electronic Surveillance and Civil Liberties, Office of Technology Assessment, Congress of the United States, Washington, DC, October 1985.

[35] Shattuck, J., "Computer Matching Is Serious Threat to Individual Rights", Communications of the ACM, June 1984, pp. 538-541.

[36] Kusserow, R.P., "The Government Needs Computer Matching to Root Out Waste and Fraud", Communications of the ACM, June 1984, pp. 542-545.

[37] Convention on Protection of Individuals with Regard to Automatic Processing of Personal Data, Council of Europe, Strasbourg, France, 1981.

[38] "Is the Computer Running Wild?", U.S. News and World Report,, February 24, 1964.

[39] The Vulnerability of Computerized Society: Considerations and Proposals, Ministry of Defense, Stockholm, Sweden, December 1979.

[40] "Workshop Stresses Dependence on Computers", Transnational Data Report, July/August 1981.

[41] Turn, R., and E. Novotny, "Resiliency of the Computerized Society", AFIPS Conference Proceedings, Vol. 53, 1983 National Computer Conference, AFIPS Press, Reston, VA, June 1983, pp. 341-349.

[42] Lerner, E.J., "Electromagnetic Pulses: Potential Crippler", IEEE Spectrum, May 1981.

Analyzing the Security of an Existing Computer System

Matt Bishop

Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

ABSTRACT

Most work concerning secure computer systems has dealt with the design, verification, and implementation of provably secure computer systems, or has explored ways of making existing computer systems more secure. The problem of locating security holes in existing systems has received considerably less attention; methods generally rely on "thought experiments" as a critical step in the procedure. The difficulty is that such experiments require that a large amount of information be available in a format that makes correlating the details of various programs straightforward. This paper describes a method of providing such a basis for the "thought experiment" by writing a special manual for parts of the operating system, system programs, and library subroutines.

Introduction

Published work in the security of computer systems tends to take one of two directions. The work may center on a new, secure (possibly provably so) computer system, and discuss its design, implementation, and verification, or the techniques used to do any (or all) of these steps. Less commonly, the work may report ways of improving the security of an existing system by discussing the known problems and methods to counter these threats. Only a few papers[1] deal with how to analyze an existing computer system in order to locate security problems

At this point, we should remind ourselves what we are trying to do. Users who have legitimate access to the system are *authorized users.* If the permissions on the system are set to allow someone to perform an action, that action is an *authorized action*; if the action is performed in the absence of such permission, it is an *unauthorized action.* A *secure* system is a system which allows only authorized users to perform only authorized actions. For example, if a user is not known to the system administrator (by an entry in the password file), he is not an authorized user and hence should not be able to access the system. Similarly, a *breach of security* occurs whenever an authorized user performs an unauthorized action, or when an unauthorized user obtains access to the system.

There are several reasons to check existing programs. The most important is that the design, implementation, and verification of new code takes quite some time, during which the new code could not be used. When one realizes that most operating systems were not designed with security as the primary consideration, the magnitude of such a task becomes apparent. Existing code, on the other hand, could be used for increasingly privileged tasks as it is examined in stages for security flaws. Second, given that there is already enough existing code to keep a nonsecure system functioning, it may be more cost-effective to check the code for security holes rather than rewriting it completely. Finally, once it is written, new software can be treated like existing software.

Unfortunately, lack of formal verification poses problems. The best way to reduce the number of security problems is "to use formal security verification methods to assure that the mandatory ... security controls employed in the system can effectively protect ... sensitive information stored or processed by the system."[2] To do this, the developers must state their security policy, the axioms used to implement the security policy, and using these axioms present a mathematical proof that the system satisfies the security policy. Then, they must show the implementation of the operating system conforms to the design. (LOCUS[3] and PSOS[4] are examples of proposed operating systems for which mechanisms of formal verification have been described.) Throughout this procedure is an assumption that the system is designed with this type of verification in mind. To submit an existing system to this procedure, one must first decide on a security policy, and then model the system mathematically and show that the system not only satisfies the security policy, but also is accurately represented by the mathematical model. Abstraction of a mathematical description from the operating system is far more difficult than implementing the operating system from the mathematical description.

It is important to realize that no method will provide the same degree of security as formally verifying a system; however, less rigorous methods can reveal security flaws, and make the writing and checking of secure system software easier and less prone to error.

The Starting Point

Given that mathematical verification is not suitable, let us look at other methods of testing, and improving, system security. The most obvious is an *ad hoc* approach of trying types of attacks that have proven successful on this, or

Work reported here was supported by the National Aeronautics and Space Administration under contract NAS2-11530.

CH2345-7/86/0000/1115$01.00 © 1986 IEEE

other, operating systems in the past. Although doing so is very effective in discovering specific security problems, it does not provide a broad, systematic approach for discovering flaws in the security of computer systems, or for testing new components.

A generalization of this method will provide a foundation for analyzing security problems. One technique for penetrating operating systems involves a formal strategy called the "Flaw Hypothesis Methodology."[5] It consists of four parts: knowing how the target operating system interacts with users, hypothesizing a flaw in that interaction, confirming that the flaw exists (through "thought experiments" and actual testing), and generalizing the flaw, and similar flaws, to a design or implementation deficiency in that operating system. Clearly, the most difficult part is taking the first step, from the knowledge of the operating system to the supposition of flaws.

Before discussing ways to make this easier, let us try to categorize the main areas in which problems arise, to gain some insight about where to look. Bisbey, Carlstedt, and Hollingsworth at the University of Southern California's Information Sciences Institute have identified several categories of system flaws which can produce security violations. The following list summarizes them by listing main areas, each broken into sub-areas:*

(1) Improper protection (initialization and enforcement):

(1a) improper choice of initial protection domain; for example, an incorrect choice of a protection domain or security partition leading to a user being able to access and change an audit trail;

(1b) improper isolation of implementation detail; for example, allowing users to bypass operating system controls and write to absolute input/output addresses;

(1c) improper change; for example, allowing data to be inconsistent while still in use, by letting one process change a database file while another, different process is accessing that file;

(1d) improper naming; for example, allowing two different programs to have the same name;

(1e) improper deallocation or deletion; for example, leaving old data in memory deallocated by one process and reallocated to another process, enabling the second process to access the information used by the first;

(2) Improper validation; for example, not checking critical conditions and parameters, leading to a process' addressing memory not in its memory space by referencing through an out-of-bounds pointer value;

(3) Improper synchronization:

(3a) improper indivisibility; for example, interrupting atomic operations such as locking;

(3b) improper sequencing; such as allowing race conditions among processes vying for resources;

(4) Improper choice of operand or operation; such as using unfair scheduling algorithms that block certain processes or users from running.

Although certainly not complete, this list provides a means of classifying most security problems, and is quite suitable as an outline of areas in which problems of security will arise.

Now that we have guidelines on where to look, we must consider how to go about looking. Unfortunately, there is no way to do this other than by trial and error. (There has been some discussion of problems leading to, and attacks taking advantage of, security flaws in operating systems generally[4,6], as well as discussions of the security of specific operating systems[7,8,9].) Such methods may be made more effective if the trials are done systematically rather than at random. One technique to systematize the search is to use a dependency graph of the control objects in the operating system to study their interaction and look for possible problems that may enable an attacker to breach security. Among the difficulties with this are the generation of the graph, and its being understood by those not familiar with the layout of the graph.

Before examining another technique, let us analyze the problem of finding security holes a bit further.

Laying the Groundwork

The key point in looking for security flaws is recognizing that the security problems we are dealing with arise from interactions between the user and the operating system. Specifically, the user creates a condition using one or more programs and then executes another program or programs which cause the operating system to ignore specific protections. For example, to copy a protected file, the user must force the operating system to ignore or override this protection (for example, by running a program at a level of privilege sufficient to cause file protections to be ineffective.)

Unfortunately, any list of methods to do this will contain only a subset of all possible methods, since any new system program would add many new ways to evade protections. Even if such a list could be made, it would be very different for each operating system, because each operating system has its own design and implementation philosophy, and the latter often differ in ways that affect very subtle points of interaction. Similarly, programs perform different tasks, and the work needed to catalogue all of the possible jobs programs may do will be endless. Indeed, the required level of security differs, too; programs executed with special powers (such as *root* or *operator* privileges) must be checked for security violations that need not be looked for in other nonprivileged programs.

But the problems that arise come from the interaction of users with the operating system, as we have said. The only two ways for a user to interact with the operating system are through programs (software) and through equipment attached to the computer (hardware), in the latter case the interface being the kernel. So, in order to examine the way users interact with the operating system, we must study how the programs interact with the operating system, and the device drivers and other routines through which the equipment interacts with the kernel.

Let us deal with individual programs first. To study how they interact with the kernel, we shall try to abstract the functionality of the program from the actual code. This will have two effects. First, it will separate security problems introduced by the coding of the program from those introduced by the design of the program. Then, the design of the program can be checked, both for internal security problems

* This organization is from Peter Neumann[6].

and for security problems arising from interaction with other programs. Once this is done, the implementation can be examined to ensure that it does not introduce other security problems.

The first step, therefore, is to figure out what the program does, and how it goes about doing it. For the first part, system documentation will provide some guidance, but because documentation very often is incorrect, incomplete, or imprecise, it is not always good to rely on it; hence, for both learning what the program does, and how it does it, one must go through the program code. Second, one must document all interaction with the operating system (such as the files looked at, and how the program uses them.) In particular, one must document all error checking and recovery (or the lack of it.)

As an outline, the following organization for this document would be appropriate:

Name

This is the name of the program. If the program may be invoked by any of several names, all should be listed.

Actions

Although similar to a specification, this section should conform to the code and not to what the program is supposed to do. This section requires that the implementation be examined and written out in such detail that someone not familiar with the code could understand not only the action of the program, but how it works, and what side effects it has. If library routines or programs already documented in this fashion are used, it is often useful to refer to the appropriate pages rather than recapitulate the actions of those routines or programs.

Apparent Assumptions

This presents any inherent assumptions. For example, if a file is assumed to be in a specific format, this should be noted here. If an assumption about the meaning of an error condition is made, list it here.

Files Used

This section names the system and user files used. It also contains a short description of each, any assumptions made about format, and the system calls used to access each.

System Calls

This lists all the system calls used.

Execution Modes

This is most useful for programs; it describes who may execute the program and with what privileges the program executes.

Known Bugs

Any known security problems are listed here. As security holes are found, they should be added. Note that suspicions should be listed (but marked as suspicions) until they are proven or disproven.

Error Handling

This describes what happens if errors occur. For example, suppose an index into an array is out of bounds; does the program dump core? Suppose a file is not in the correct format? Are there checks to ensure any reading or writing succeeds?

Library Functions Used

List the names, versions, and dates of any library functions used.

Manual Page Version

Give the author, date, version of the program, and system for which this document was prepared.

We shall call this document the *security manual page* to distinguish it from the usual manual page. (A sample page, for the UNIX* library routine *getlogin*, follows the references.)

Of course, in the Known Bugs section, one should document any discovered security problems.

This documentation should not be confined to the program only. Very often system programs need to perform a task such as looking up a name in a table to obtain associated data. These functions are performed so often that they have been collected into a set of library routines. Since these routines affect the function of each program in which they are used, it would save time and work to document these routines as described above. This would provide one reference for each library routine, rather than having the same routine be checked once for each program in which it is used (and risking a security hole being overlooked once). Similarly, new programs should use library routines whenever possible, and rather than duplicating code amongst several programs, the code should be changed into a library routine which the programs then call.

As an example of why documentation that describes the implementation of a program or library routine is necessary, consider the *getlogin*() bug, which exists on many UNIX systems. According to the manual[10], "[g]etlogin returns a pointer to the login name as found in */etc/utmp*." Although accurate, this description is very imprecise. *Getlogin* actually returns the login name of the user whose terminal is associated with the input, output, or error streams; this may or may not be the same as the login name of the person who executed the program. The security manual page should make this final statement, even though the manual page states *getlogin*'s function as indicated.**

Because of its complexity and function, the kernel must be checked differently than system programs and libraries. The principle is the same – analyze the code and document those parts which interact with other programs and equipment – but many security manual pages, not just one, will be written for it. Specifically, at least one page per system call and device driver will be necessary, stating error conditions and precisely how they are handled, as well as how the system calls and device drivers are accessed. Main components of the kernel – the initialization routines, the scheduler, and so forth – must also be documented, as must any routines that rely on files or specific memory locations or any other external factors.

Hence, the first step to checking the security of programs and the operating system is:

> *Document each program, system call and device driver, and library routine thoroughly, not just as to purpose but also as to its side effects and error handling.*

* UNIX is a Trademark of Bell Laboratories.

** See the sample security manual page that follows the references.

Hypothesizing the Flaws

Once a manual page or set of manual pages have been written, the process of locating security flaws begins. Unfortunately, the only known approach to doing this is largely *ad hoc.*

There are analogies in other fields. For example, the only way communications analysts can assess vulnerabilities of communications systems is to study the system thoroughly, and then draw on their knowledge of that system, their experience, and their knowledge of attacks that worked with other systems, to hypothesize security problems. They then test for these suspected flaws. The situation is precisely the same for computer security.

As with analyses of the vulnerability of communications systems, we can draw on past experience. There have been a number of studies of operating system security in general and of specific penetrations of various operating systems (some of these have been referred to earlier.) These studies provide knowledge of attacks that worked with many different systems. Combined with the knowledge gleaned from mathematical analyses of other systems and the weaknesses uncovered using those tools, all this experience provides a very solid background for hypothesizing security flaws.

The security manual described in the previous section will provide both the means of studying the system thoroughly and a reference guide useful in formulating hypotheses. As each program or routine algorithm is considered by itself, flaws may become apparent. (In fact, this happened with the *getlogin* manual page attached to this report. The second of the Known Bugs section was found by noticing the assumption made in step 4 of the algorithm, comparing it to step 3, and wondering what would happen if the assumption was invalid.) Correlating programs which use the same system files may reveal that the interaction of some such programs presents attackers with opportunities to subvert the system, or that these programs make inconsistent assumptions (or invalid assumptions) about the data in the file, or the way the file is used. A similar comment holds for programs and system calls; special attention should be paid to those system calls used to access and manipulate system files. The section on error handling should be quite fruitful for hypothesizing flaws. Many error conditions are not adequately handled, not handled correctly, or simply ignored. Very often this produces unusual situations that may present security holes which a clever attacker can exploit[11,12].

Hence, the second step to checking the security of programs and the operating system is:

> *Drawing on the documentation, past experience, and general knowledge of operating system vulnerabilities, hypothesize security flaws in the computer system, and test either to confirm or to deny that those flaws exist.*

Summary

When checking an existing computer system for security, both the operating system kernel and the system libraries and privileged programs must be examined. (If none of these has security flaws, applications programs will not be able to breach security.) They should be examined in the above order; note that this will ensure that the operating system calls, which are the basis for system library routines and system programs, will be examined before the code using them is examined.

Within each of these aspects, the steps of the "Flaw Hypothesis Method" as described in sections 2, 3, and 4 should be used to locate security flaws, paying special attention to the problem areas described in section 2. For each aspect, a security manual of the sort described in section 3 should be written and used as the basis for examining the interaction of the various components of the kernel, the libraries, and the system programs as discussed in section 4.

While this method will not ensure perfect security of a computer system, it will significantly increase the difficulty of an attacker penetrating the system.

Acknowledgements: My deepest thanks to Barry Leiner and Peter Neumann, who both made very valuable suggestions towards improving this paper; to Mike Long, Bill Wall, and George Hays, for their incisive comments; and to Larry Hofman and Bob Brown.

References

[1] Denning, Dorothy E., *An Intrusion-Detection Model*, Technical Report CSL-149, SRI International, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park, CA 94025 (Nov. 1985)

[2] –, *Trusted Computer System Evaluation Criteria*, CSC-STD-001-83, Department of Defense Computer Security Center, Fort George G. Meade, MD 20755 (Aug. 1983)

[3] Walker, Bruce, *et al.*, *Specification and Verification of the UCLA UNIX Security Kernel,* CACM **23**(2), pp. 118-131 (Feb. 1980)

[4] Neumann, Peter G., *et al.*, *A Provably Secure Operating System: The System, Its Applications, and Proofs*, Computer Science Laboratory Report CSL-116, SRI International, Computer Science Laboratory, Menlo Park, CA (May 1980)

[5] Linde, Richard R., *Operating System Penetration,* in the 1975 National Computer Conference Proceedings (AFIPS Conference Proceedings **44**), pp. 361-368 (May 1975)

[6] Neumann, Peter G., *Computer System Security Evaluation,* in the 1978 National Computer Conference Proceedings (AFIPS Conference Proceedings **47**), pp. 1087-1095 (Jun. 1978)

[7] Attanasio, C. R., Markstein, P. W., and Phillips, R., *Penetrating an Operating System: a Study of VM/370 Integrity*, IBM Systems Journal **15**(1), International Business Machines Corp., pp. 102 - 116 (1979)

[8] Grampp, F. T., and Morris, R. H., *"UNIX Operating System Security"*, AT&T Bell Laboratories Technical Journal **63**(8), pp. 1649-1672 (Oct. 1984)

[9] Ritchie, Dennis M., "On the Security of UNIX", in *UNIX System Manager's Manual, 4.2 Berkeley Software Distribution, Virtual VAX*-11 Version,* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA (Mar. 1984), as reprinted by the USENIX Association

* VAX is a Trademark of Digital Equipment Corporation.

[10] –, *UNIX Programmer's Manual Reference Guide, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version,* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA (Mar. 1984), as reprinted by the USENIX Association

[11] Bishop, Matt, *How to Write a Setuid Program* (extended abstract), Proceedings of the Spring 1986 Cray User Group (May 1986)

[12] Darwin, Ian, and Collyer, Geoff, *Can't Happen or /* NOTREACHED */ or Real Programs Dump Core*, 1985 Winter USENIX Proceedings (January 1985)

Appendix – Sample Security Manual Page

NAME

getlogin – get login name

INVOCATION

char *getlogin();

ACTIONS

Getlogin returns the user believed to be using the controlling terminal. It does this as follows:

1. Find the first of the file descriptors 0, 1, 2 associated with a terminal by running an *ioctl*(2) on each and seeing which one succeeds; if all fail, return 0.
2. Find the device/inode pair corresponding to that terminal by using *fstat*(2), and scan the files in the directory */dev/* until one is found with that device/inode pair. If none is found, return 0.
3. Search the file */etc/ttys* for that file name, and count the number of lines *N* skipped before it is found. If not found, return 0.
4. Read the *N*th record in */etc/utmp*; this corresponds to the user currently using that terminal. It is in the format of *utmp*(5).
5. Return the contents of the *ut_name* field of that record. Note it is kept in a static area, and is overwritten the next time *getlogin* is called.

APPARENT ASSUMPTIONS

The first of the file descriptors 0, 1, and 2 that is associated with a terminal is associated with the terminal the user logged in on.

The number of the (text) line in */etc/ttys* describing a terminal corresponds to the offset into the file */etc/utmp* for that terminal.

FILES USED

/etc/ttys List of terminal names, one per line; *open*(2), *read*(2), *close*(2)

/etc/utmp List of logged-in users; assumes each record corresponds to a line in */etc/ttys* and that the records have the same order; *open*(2), *lseek*(2), *read*(2), *close*(2)

/dev/ Directory containing files corresponding to terminals; used to determine the name of the controlling terminal; *open*(2), *read*(2), *close*(2)

SYSTEM CALLS

close(2), *fstat*(2), *ioctl*(2), *lseek*(2), *open*(2), *read*(2), *sbrk*(2), *stat*(2)

EXECUTION MODES

This is a system library function.

KNOWN BUGS

If the first file descriptor found to be associated with a terminal is not associated with the controlling terminal, the name of the user at the associated terminal will be returned, and not the name of the user at the controlling terminal.

If a line is added to or deleted from */etc/ttys*, the algorithm used to associate users with their terminal names fails miserably. This problem can be corrected by looking in the *ut_term* field of the record and comparing it with the name obtained from */etc/ttys.*

ERROR HANDLING

On error, it is supposed to return 0.

No error check to be sure the *lseek*(2) to the record in */etc/utmp* succeeds.

No error check to be sure the record in */etc/utmp* corresponds to the name of the terminal.

Silently assumes names which are shorter than the space allocated in the record for user names are blank padded.

LIBRARY FUNCTIONS USED

NAME	*VERSION*	*DATE*
getlogin.c	4.2	11/14/82
isatty.c	4.1 (Berkeley)	12/21/80
ttyslot.c	4.1 (Berkeley)	12/21/80
ttyname.c	4.3 (Berkeley)	5/7/82
closedir.c	4.5 (Berkeley)	7/1/83
opendir.c	4.5 (Berkeley)	7/1/83
readdir.c	4.5 (Berkeley)	7/1/83

MANUAL PAGE VERSION

AUTHOR	Matt Bishop
DATE	December 1, 1985
SYSTEM	4.2 BSD
VERSION	getlogin.c 4.2 (11/14/82)

A NETWORK TECHNIQUE TO ACHIEVE PROGRAM AND DATA SECURITY WITH NOMINAL COMMUNICATIONS OVERHEAD

J. R. Driscoll*, H. N. Srinidhi*, and T. S. Chesser**

*Dept. of C. S., Univ. of Central Florida, Orlando, FL 32816
**Martin Marietta Data System, Orlando, FL 32809

ABSTRACT

In an environment where copyrighted programs are shared by numerous independent users via a network configuration (e.g., a local area network), security research efforts have focused primarily on program security or on data security, but not both. In this paper we are concerned with achieving both types of security and characterizing the communication cost of doing this. We survey existing security techniques for the environment of concern and present a radically new technique. We analyze and simulate the communication cost of each technique to demonstrate their relative merits. It is shown that the technique requires nominal communications overhead while achieving both program and data security.

1. Introduction

This paper deals with three issues - securing user-data, securing program-logic, and reducing communications traffic. The securing of user-data has been widely studied and many effective techniques exist (see, for example, [DENN82]). By program-logic, we mean the set of all executable paths of a program. The securing of program-logic in a network environment is an issue not commonly studied or even mentioned in the literature. Achieving security of program-logic, in conjunction with securing user-data, will have an effect on communications traffic in a network environment.

Security in computer networks is hampered by the requirement that the application program and the portion of the database it manipulates must be physically resident on the same processor node at program execution. This may require transmission of programs or data over wire, microwave, or satellite links with resulting vulnerability to interception, decryption, and analysis. Since database management systems are the most widely used applications in main frame sites with information centers [KNIG85], we feel that a way which minimizes the shipment of program-logic and user-data (with special emphasis on database applications) would increase overall security within network environments.

Proprietary software and sensitive data are two assets, each of which have value [PARK81]. Our work to secure these two assets in a network environment has been influenced by the following additional concerns:

(1) Software updates: Information systems inherently require periodic software upgrading. Dispersion of a change to all users can be expensive and may never result in a complete upgrade among all the users.

(2) Communications traffic: The volume of communications traffic and the communications rate affects the performance of the network.

(3) Number of users serviced: The number of users serviced by a single copy of a program can determine main storage requirements. A processor servicing a large number of users could very easily require a large I/O buffer space.

(4) Volume of user-data: The volume of user-data determines the amount of secondary storage required. The issues are cost and responsibility for that storage.

Section 2 uses an example to motivate and describe two prevalent techniques for the network environment of concern and then describes a new, more secure, technique. Our descriptions allude to the above concerns. Section 3 outlines an approach for implementing the new technique, and Section 4 presents performance results for a simulation of this approach. In Section 5, we provide an analysis to help verify our simulations. Section 6 concludes our paper by discussing the significance of our work and presents alter-

CH2345-7/86/0000/1120$01.00 © 1986 IEEE

native approaches for implementing our new technique.

2. Interface Techniques

In this section, we present a commercial example to motivate and describe the characteristics of two prevalent techniques for the program-logic and user-data security problem when these two assets do not have the same owner or do not have the same node residency during execution. It will be seen that each of these techniques causes a compromise of either program-logic or user-data, and neither one provides an acceptable solution to the four security related concerns listed in Section 1. We then introduce a radically new technique which is applicable when generalized database management systems are considered for the program-logic environment. It will be seen in Sections 4 and 5 that this new technique provides a viable alternative to existing techniques for the defined environment.

2.1 Commercial Application Example

Let a person's house be an environment. (A business could just as well be used here for an environment.) Consider the Internal Revenue Service. The IRS requires, essentially, annual reports from people. People should have access to standardized application programs which enable them to maintain IRS acceptable records for completing IRS forms (management information). It is easy to imagine that, in the near future, a company like H & R Block would make convenient IRS applications programs available for people to maintain IRS acceptable bookkeeping records so that IRS forms can be automatically completed. The conditions here are the following:

(a) People insist on keeping their data in-house, so they will insist on having their own copy of H & R Block's application program.

(b) H & R Block will not provide a copy of any of their application programs to people for fear that the programs would be passed on without reimbursement. H & R Block will insist that any person who desires to use programs submit their data to H & R Block.

(c) The application programs developed by H & R Block will need to be modified periodically to reflect annual changes, such as new tax tables, alternations in computation procedures, and new formats for printing. H & R Block will insist that, at any particular time, every person must use the same application programs. This requirement will be impossible to enforce unless the programs "run" at computer installations controlled or owned by H & R Block.

We will refer to this example to help explain the security techniques described in the following subsections.

2.2 Data Secure Technique

This technique is characterized essentially by the fact that the user purchases the database management system software and data storage hardware. Since the user executes the program in his environment and the data is not transmitted, absolute data security is insured. However, protection of proprietary software is of concern since the software may be plagiarized. Software updates involve additional expense to the software vendor. The cost of the processing hardware and software may be of concern to the user. The weakest security mechanism here is the software licensing agreement [PARK81].

Referring to the commercial example in Section 2.1, H & R Block would have to allow users to purchase their application programs which violates condition (b). Consequently, the Data Secure technique is not a realistic technique for this example.

2.3 Program Secure Technique

This technique is characterized essentially by the fact that the database management system runs on one machine (the program node), while all the user-data originates on another machine (the data node) and must be transmitted to the program node for processing. Assuming the data traffic contains sensitive information, effective data protection mechanisms [ABRA85, DENN82, HOFF77, MCMA85, WOOD86] are essential. But, since data is transmitted in some form, it will always be vulnerable; absolute data security is not provided. However, software maintenance is simplified and proprietary software is protected. The volume of data and the communications transfer rate impact system performance. The communications traffic may be a problem. For some techniques of this type, the user-data is transmitted and then resides at the program node for processing (e.g., Martin Marietta Data Systems of Orlando, Florida); for others, it is repeatedly transmitted for processing and resides at the data node. The number of users serviced is of concern for the program node. Volume of user-data may be of concern to either the program node or the data node.

Referring to the commercial example in Section 2.1, people would have to transmit their data to H & R Block which violates condition (a). Consequently, the Program Secure technique is not a realistic technique for this example.

2.4 Program and Data Secure Technique

Absolute protection of proprietary software and sensitive data can be accomplished. This is true when the program-logic environment is defined by a generalized data base management system. The configuration shown in Fig. 1 provides a solution by keeping the program-logic separate from the user-data [CHES85, DRIS84].

Execution of a DBMS occurs on the program node. The user-data resides on the data node. This technique employs a signal/response method to accomplish DBMS execution. A signal packet is generated by the program node and transmitted to the data node for interpretation. The data node has processing and storage abilities. Interpretation of the signal packet requires some software interface to exist on the data node. The intent of this configuration is to keep this interface simple and compatible with various data nodes so that any off-the-shelf personal computer can act as a data node. The data node performs the requested signal operation and transmits a simple response back to the program node if necessary. Thus, the number of users serviced will not be a concern. Since the user-data is never transmitted across the communications link between the two nodes, absolute data security is insured. Thus volume of user-data is of concern only to the data node. The DBMS is resident on the program node and its program-logic is never transmitted; only data manipulation instructions are transmitted. Thus, software maintenance is simplified and proprietary software is protected. The volume of signals and responses and the communications traffic rate impact system performance. Communications traffic may be a problem and needs to be studied.

Referring to the commercial example in Section 2.1 and the configuration shown in Figure 1, the processing facility of the program node would be owned by H & R Block. People would buy personal computers of their choice and each person's equipment would represent a data node. A person would select commands offered by the H & R Block application programs. The application programs would issue commands to cause data movement or modification within the person's system. The financial records of the person do not move outside the person's system. Each person would establish and maintain their private financial records during a year. At the end of a year, each person would produce their own income tax return on their printer. H & R Block application programs are used but not seen by the people. H & R Block can easily alter or update their programs to comply with IRS whims. Consequently, all the conditions (a), (b) and (c) of Section 2.1 are satisfied by this technique.

The primary purpose of this paper is to present evidence which demonstrates that this new technique is a viable alternative to existing techniques. This is done in Sections 4 and 5. In the next section, we outline a method of implementing this new technique.

3. The Ship-Bit Method

In this section we describe a low-level hardware method of implementing the Program and Data Secure technique. We consider only generalized database management systems for the program-logic environment. This method involves shipping machine language data manipulation instructions with the generalized database management system to the data node for execution. This is accomplished by extending the machine's instruction opcode field to include a special bit identifying an instruction for shipment. This method requires additional storage for the shipping bits, interpretation logic for instruction type identification, and modified compilers which identify the instructions that need to be shipped.

If an instruction directly references data which exists on the data node, the instruction must be marked as an instruction to be shipped to the data node. Clearly, I/O operations using peripheral devices residing on the data node (i.e. printer or terminal) must also be marked

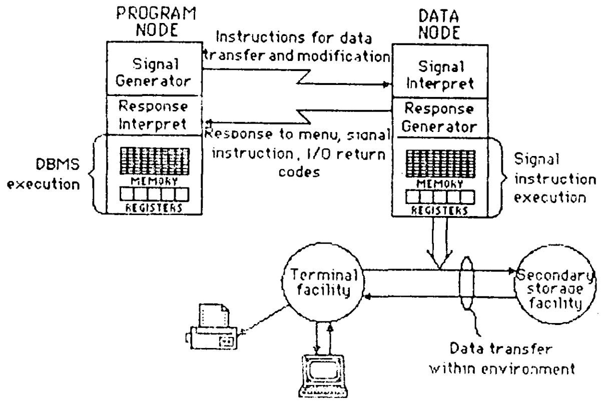

PROGRAM NODE
DATA NODE
Instructions for data transfer and modification
Signal Generator
Signal Interpret
Response Interpret
Response Generator
Response to menu, signal instruction, I/O return codes
DBMS execution
MEMORY
REGISTERS
Signal instruction execution
Terminal facility
Secondary storage facility
Data transfer within environment

Fig. 1. The Program and Data Secure Configuration.

for shipment to the data node. Since the DBMS executes on the program node, instructions which control the execution flow of the DBMS are never shipped to the data node for execution. An example of this type of instruction is a branch instruction.

Automation of the process of identifying the data manipulation related DBMS instructions poses an interesting problem. A likely solution is to design a compiler capable of distinguishing between data node and program node variables. If there is a mix of program node and data node variables in an instruction, then the compiler must decide as to where the instruction is best executed at that instant. An alternative solution is to have a preprocessing software which would flag those instructions that need shipping and replace them by the appropriate calls to shipping routines. We are not concerned with the design of such a software in this paper.

4. Simulation Results

Our concern here is to show by experimentation the feasibility of the method described in Section 3. It is true that the messages passing via the link between the program and data nodes need be encrypted and authenticated [DENN82]. The overhead associated with encryption would be proportional to the length of the message and do not alter our results mentioned here. All software overheads are ignored as the slow communication link speed will dominate the execution times

A simple DBMS was written using IBM 370 assembly language. The DBMS was capable of defining up to four files, inserting and deleting records, printing files, deleting files, and displaying directory contents. The DBMS was not designed for storage of real data as we were more interested in writing a DBMS capable of performing a reasonable test of basic data manipulation operations. Consequently, the DBMS was not complex; it offered only basic file-handling capabilities which served our purpose.

The data manipulation instructions of the DBMS were identified and the DBMS was executed on an ISPS [BARB81] simulation of the three techniques corresponding to Data Security, Program Security, Program and Data Security. The ISPS simulator ran on VAX 11/730 operating on VMS. The DBMS operations performed under the simulator consisted of three stages of testing:

(I) Define DBMS Schema
-Define 1 File
-Define 2 Files
-Define 3 Files
-Define 4 Files

(II) Mass Load DBMS Data
-Insert Records in Random Order
-Insert Sorted Records

(III) Typical DBMS Operations
-Display Directory Contents
-Print Files
-Delete Files
-Delete Records
-Insert Records
-Search Records on Key Field

A more efficient DBMS design in which the directory data is read once per session is also considered. This reduces the communications traffic in the Program and Data Secure technique because an instruction to read directory data is shipped only once. In the original DBMS design, an instruction to read directory data was shipped to the data node for each query.

The statistics resulting from each stage of testing are now summarized. The communications traffic overhead incurred by each technique is presented in terms of both volume and transmission delay time.

Stage I involved the definition of the database files. This activity involved mostly disk directory operations. The timing delay graph for each technique in Fig. 2 shows the transmission delay due to communications traffic volume plus I/O delay occurring in each technique. This graph does not indicate the effects of a more efficient DBMS design since detailed peripheral activity was not collected by the simulator.

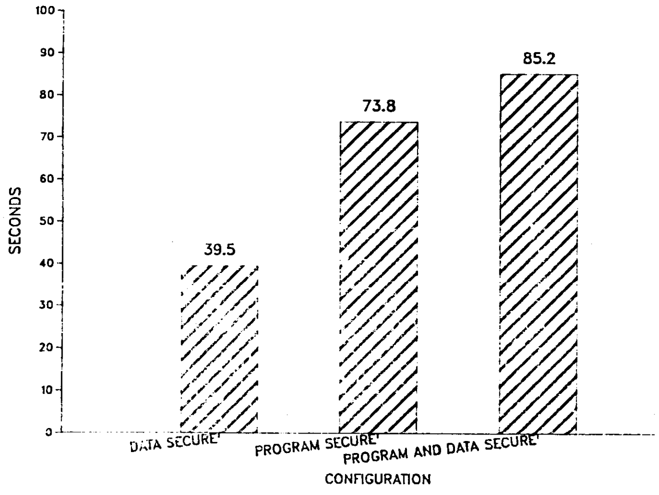


Fig. 2. Communications Plus I/O Traffic Delay for Stage 1.

The increase in communications traffic for each file defined is linear as shown in Fig. 3. The graph clearly indicates the increase in communications traffic of the Program and Data Secure technique over the Program Secure technique. The projected traffic for a more efficient DBMS design substantially reduces this increase for the new technique. The Program Secure technique is unchanged because there is no communications traffic in this situation that results from disk I/O operations. The residency of the disk causes the DBMS design change to have no effect on communications traffic in that technique.

Stage II testing involved a mass load of the database. This type of activity references the disk frequently. Therefore, the residency of the disk data will dictate the volume and types of communications traffic. The timing delays due to communications traffic plus I/O traffic in each technique is presented in Fig. 4 for the Stage II testing. The more efficient DBMS design would have reduced these values since the communications traffic volume would be reduced. The timing delay for the Data Secure technique is the time required to perform the peripheral I/O operations. The delays for the other two techniques involve this I/O delay plus communications traffic delay.

Stage III involved a collection of typical DBMS operations. For Stage III testing, Fig. 5 summarizes the communications traffic delay for each technique. This timing delay is directly related to the communications traffic volume plus I/O activity which occurs for this testing stage in each technique. The timing delay in the data secure technique results only from I/O activity. The same amount of I/O activity occurs in all the three techniques. Because the detailed peripheral I/O activity was not collected by the simulator for a more efficient DBMS design, an accurate display of timing delays for that DBMS design could not be presented.

The overall communications traffic volume is compared in Fig. 6 for the Program Secure technique and the new technique. The communications traffic volume resulting from a more efficiently designed DBMS is presented. The Program and Data Secure technique reflects a significant savings in traffic volume with the more efficient DBMS. This graph illustrates that for a more efficient DBMS, the new technique has an overhead of only 3.7% over the communications traffic volume for the Program Secure technique. To achieve absolute program and data security, this overhead expense is minimal.

5. Estimates of Query Execution Times

We present here a simplified model to estimate the query execution times for the Data Secure, the Program Secure and the Program and Data Secure techniques. Parametric values used for the simulation will be plugged into this model and the results obtained will be compared with those of Section 4 to illustrate the validity of our model.

We list below the various parameters that are included in the model:

C Number of bytes of control instructions in the program to answer a user query.

I_d Number of bytes of data manipulation instructions executed in the program to answer a user query.

R Number of bytes of data that need to produced and transmitted between the data and program nodes in the process of answering a user query.

B Baud rate of the transmission line (in bits/sec.).

b Number of bits transmitted per byte (byte + start and stop bits).

T_d Total time for all data accesses between the disk, the main memory and the processor, to answer a user query.

T_i Total time for transferring the program from the disk to the main memory to answer a user query.

T_e Total execution time for the program to answer a user query excluding any disk accesses.

l Average length (in bytes) of a data manipulation instruction.

T_m Time to access a single byte from main memory.

Note that $(C + I_d)$ indicates the total number of bytes of the program-logic to answer a user query.

5.1 Query Execution Time in the Data Secure Technique

This technique will be the fastest. However, this technique offers no program-logic security whatsoever. Assuming that the results are transmitted to the user terminal via Direct Memory Access

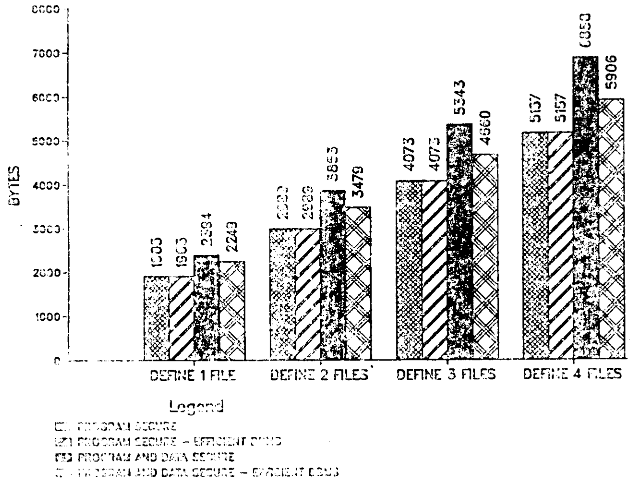


Fig. 3. Communications Traffic Volume for Stage I

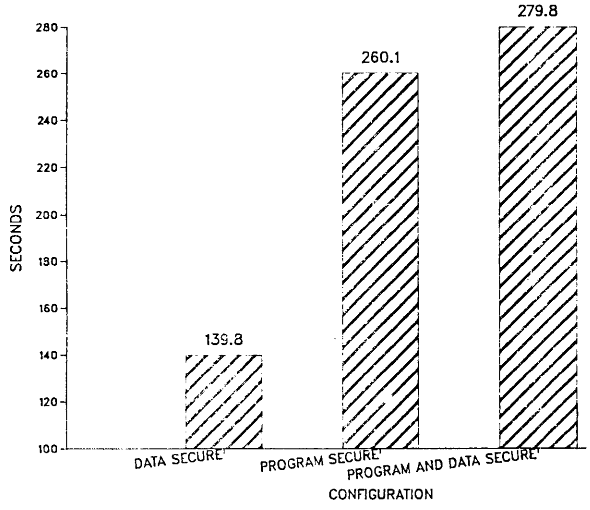


Fig. 4. Communications Plus I/O Traffic Delay for Stage II.

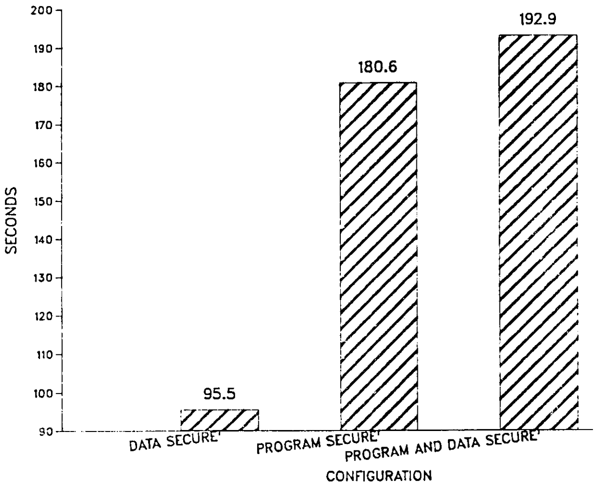


Fig. 5. Communications Plus I/O Traffic Delay for Stage III.

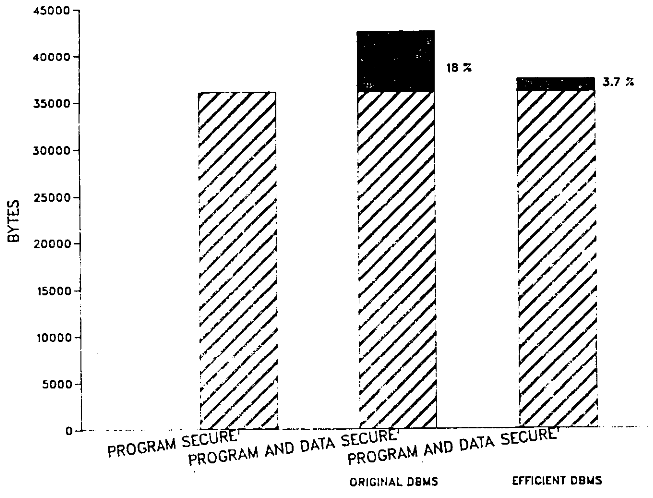


Fig. 6. Overall Communications Traffic Volume.

(DMA), the total query execution time is

$$T_{DS}=T_i+T_d+T_a+T_m(C+I_d)+T_mR \qquad (1)$$

where T_mR is the time to store the result bytes in main memory. There is no communication cost in this technique.

5.2 Query Execution Time in the Program Secure Technique

This technique offers only program-logic security and experiences a penalty in shipping the menu and the result bytes R from the program node to the data node. As in Section 5.1, we assume that the results are accessed in a DMA fashion for transmission to the data node. The total query execution time is

$$T_{PS}=T_{DS}+(b/B)R+T_mR \qquad (2)$$

of which

the communications cost=(b/B)R. (3)

5.3 Query Execution Time in the Program and Data Secure Technique

This technique provides the security for both program-logic and user-data, but suffers the penalty in shipping the data manipulation instructions to the data node. We assume here that we have to transmit a control byte with each transmitted instruction. The control byte instructs the data node as to the contents of the signal packet and the type of response (or no response) expected. As before, we assume that the result bytes are transmitted to the user terminal in a DMA fashion. The total query execution time is

$$T_{PDS}=T_{DS}+(I_db/B)(1+1/l)+T_m(2I_d+R)+(b/B)R' \qquad (4)$$

of which

$$\textit{the communication cost} = (I_db/B)(1+1/l)+(b/B)R' \qquad (5)$$

The factor $2I_dT_m$ accounts for the receiving, storing and fetching of the data manipulation instructions at the data memory. The factor $I_d(1/l)$ accounts for the number of control bytes that get transmitted with the data manipulation instructions. The value of R' in this case would be lower than the value of R in Section 5.2 because only menu but no query results need be shipped to the data node.

5.4 An Example

We illustrate below an application of our model to validate the simulation results of stage III of the testing described in Section 4. A comprehensive testing of typical DBMS operations, seems to be appropriate as our example. The following parametric values were used for or obtained from the simulation:

T_m 1 Microsecond.

B 1200 bits/sec.

b 8 bits(ignoring start and stop bits).

l 3(majority of data manipulation instructions being the RR, the SI and the RX instruction types of IBM370).

C 1248 bytes.

I_d 3354 bytes.

R 12383 bytes for the Program Secure technique; 9824 bytes for the Program and Data Secure technique.

From equations (3) and (5), the communications costs are 82.55 seconds and 95.23 seconds for the Program Secure technique and the Program and Data Secure technique, respectively. The corresponding values from Fig. 5 are 85.1 seconds and 97.4 seconds, respectively, after eliminating I/O delays. The model agrees fairly closely with the simulation results. The small disparity is accounted in that the above parametric values do not take into account transmission of directory data or condition code responses between the program node and the data node.

6. Conclusions

In this paper we considered an environment where copyrighted programs are shared by numerous independent users via a network configuration. We were concerned with achieving both program and data security and characterizing the communication cost of doing this. We surveyed existing security techniques for the environment of concern and presented a radically new technique. We analyzed and simulated the communication cost of each technique to demonstrate their relative merits. We showed that our new technique requires nominal communications overhead while achieving both program and data security.

The new technique discussed in this paper achieves security by controlled shipment of flagged instructions at the assembly level. Two other approaches are possible:

(1) The Software Trap Approach which consists of software flagging of code sections that need to be shipped via a software trap instruction. This approach requires a special handler for shipping instructions invoked by execution of a software trap instruction, and modified compilers which identify sections of code that are to be shipped. The compiler will insert the trap instruction and the number of bytes to be shipped prior to the

section of code that will be shipped. Upon execution of the trap instruction, the appropriate number of bytes will be shipped by the trap handler.

(2) The Software Macro Call Approach which consists of software macro calls to the data node which make use of the data node's operating system. This approach requires modified compilers which identify what can be invoked as a system call to the node's operating system or a call to a receiver subroutine at the data node.

Of the above two approaches, the Software Macro Call approach seems promising in improving the communication costs as the number of instructions that need to be shipped are considerably reduced. The performance of our new technique using this approach is expected to considerably excel the other techniques. We are studying the Software Macro Call approach by implementing it on a demonstration system consisting of two Intel system/310 microcomputers obtained under a grant from the Intel Corporation.

REFERENCES

[ABRA85] M. D. Abrams, "Observations on Local Area Network Security," Proceedings of the IEEE Aerospace Computer Security Conference, McLean, Virginia, pp. 77-82, Mar. 1985.

[BARB81] M. R. Barbacci, "Instruction Set Processor-Specification (ISPS): The Notation and its Application", IEEE Trans. Comp., Vol. C-30, pp. 24-40, Jan. 1981.

[CHES85] T. S. Chesser, "A Secure Network Configuration with Minimal Communications Traffic," Master's Project, Department of Computer Science, University of Central Florida, Dec. 1985.

[DENN82] D. E. Denning, Cryptography and Data Security, Addison-Wesley, Reading, Mass., 1982.

[DRIS84] J. R. Driscoll and H. N. Srinidhi, "Minimization of Logic and Data Traffic in Distributed Networks," Intel Grant Proposal, University of Central Florida, awarded Dec. 1984.

[HOFF77] L. Hoffman, Modern Methods for Computer Security and Privacy. Prentice-Hall, Inc.,1977.

[KNIG85] B. Knight, "Info Center Concept Enters Middle-age," Software News, pp. 23-25, Oct. 1985.

[MCMA85] E. M. McMahon, "Restricted Access Processor - An Application of Computer Security Technology," Proceedings of the IEEE Aerospace Computer Security Conference, McLean, Virginia, pp. 71-73, Mar. 1985.

[PARK81] D. R. Parker, Computer-Security Management, Prentice Hall, Inc., 1981.

[WOOD86] P. Woodie, "Distributed Processing Systems Security: Communications, Computer, or Both," Proceedings of the IEEE Int. Conf. on Data Engineering, Los Angeles, California, pp. 630-636, Feb. 1986.

From RIG to Accent to Mach: The Evolution of A Network Operating System

Richard F. Rashid

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Abstract

This paper describes experiences gained during the design, implementation and use of the CMU Accent Network Operating System, its predecessor, the University of Rochester RIG system and its successor CMU's Mach multiprocessor operating system. It outlines the major design decisions on which the Accent kernel was based, how those decisions evolved from the RIG experiences and how they had to be modified to properly handle general purpose multiprocessors in Mach. Also discussed are some of the major issues in the implementation of message-based systems, the usage patterns observed with Accent over a three year period of extensive use at CMU and a timing analysis of various Accent functions.

1. Background

Mach is a multiprocessor operating system kernel currently under development at Carnegie-Mellon University. In addition to binary compatibility with Berkeley's current UNIX 4.3 bsd release, Mach provides a number of new facilities not available in 4.3, including:

- Support for tightly coupled and loosely coupled general purpose multiprocessors.
- An internal symbolic kernel debugger.
- Support for transparent remote file access between autonomous systems.
- Support for large, sparse virtual address spaces, copy-on-write virtual copy operations, and memory mapped files.
- Provisions for user-provided memory objects and pagers.
- Multiple threads of control within a single address space.
- A capability-based interprocess communication facility integrated with virtual memory management to allow transfer of large amounts of data (up to the size of a process address space) via copy-on-write techniques.
- Transparent network interprocess communication with preservation of capability protection across network boundaries.

As of May 1986, Mach runs on most uniprocessor VAX architecture machines: VAX 11/750, 11/780, 11/785, 8200, 8600, 8650, MicroVAX I, and MicroVAX II. Mach also runs on two multiprocessor VAX machines, the four (11/780 or 11/785) processor VAX 11/784 with 8 MB of shared memory the VAX 8300 (with up to 4 processors). Mach has already been ported to the IBM RT/PC and work has begun on ports to the uniprocessor SUN 3 and multiprocessor Encore MultiMax. The current version of the system, Mach-1, includes all of the features listed above and is in production use by CMU researchers on a number of projects including a multiprocessor speech recognition system called Agora [5] and a project to build parallel production systems.

Mach is the logical successor to CMU's Accent [16, 17] kernel -- an operating system designed to support a large network of uniprocessor scientific personal computers. The design and implementation of Accent was in turn based on experiences gained during the development of the University of Rochester's RIG system [3, 14], a message-based network access machine. Both RIG and Accent have seen considerable use over the years. RIG provided a variety of functions including terminal support and remote file access within the Rochester environment until early this year when the last RIG machine was decommissioned. Accent continues in use at CMU as the basic operating system for a network of 150 PERQ workstations and has seen commercial use in printing and publishing workstations as well as engineering design systems. As a third generation network operating system Mach, benefits from the lessons learned in over ten years of design, implementation and use of RIG and Accent. This paper summarizes the lessons of those systems and their impact on the design and implementation of Mach.

2. The Evolution of Accent from RIG

Implementation of RIG began in 1975 on an early version of the Data General Eclipse mini-computer. The first usable version of the system came on-line in the fall of 1976. Eventually the Rochester network included several RIG Eclipse nodes as network servers and a number of Xerox Altos acting as RIG client hosts. RIG provided clients network file services, ARPANET access, printing services and a variety of other functions. Active development continued well into the 1980's but obsolescence of its Data General Eclipse and Xerox Alto hardware base eventually dictated its demise in the Spring of 1986.

2.1. The RIG Design

The basic system structuring tool in RIG was an interprocess communication (IPC) facility which allowed RIG processes to communicate by sending packets of information between themselves. RIG's IPC facility was defined in terms of two basic abstractions: *messages* and *ports*.

CH2345-7/86/0000/1128$01.00 © 1986 IEEE

A RIG port was defined to be a kernel-provided queue for messages and was referenced by a global identifier consisting of a dotted pair of integers *<process number.port number>*. A RIG port was protected in the sense that it could only be manipulated directly by the RIG kernel, but it was unprotected in the sense that any process could send a message to a port. A RIG port was tied directly to the RIG abstraction of a *process* -- a protected address space with a single thread of program control.

A RIG message was composed of a header followed by data. Messages were of limited size and could contain at most two scalar data items or two array objects. The type tagging of data in messages was limited to a small set of simple scalar and array data types. Port identifiers could be sent in messages only as simple integers which would then be interpreted by the destination process.

Due largely to the hardware on which it was implemented, RIG did not allow either a paged virtual memory or an address space larger than 2↑16 bytes. RIG did, however, use simple memory mapping techniques to move data [3]. The largest amount of data which could be transferred at a time was 2K bytes.

2.2. Problems with RIG

The RIG message passing architecture was originally intended more as a means for achieving modular decomposition (much like Brinch-Hansen's RC4000) rather than as the basis for a distributed system. It was discovered early on, though, that RIG's message passing facility could be adapted as the communication base for a network operating system. Unfortunately, as RIG became heavily used for networking work at Rochester a number of problems with the original design became apparent:

- **Protection**

  The fact that ports were represented as global identifiers which could be constructed and used by any process implied that a process could not limit the set of processes which could send it a message. To function correctly, each process had to be prepared to accept any possible message sent to it from any potential source. A single errant process could conceivably flood a process or even the entire system with incoherent messages.

- **Failure notification**

  Another difficulty with global identifiers was that they could be passed in messages as simple integers. It was therefore impossible to determine whether a given process was potentially dependent on another process. In principle any process could store in its data space a reference to any other process. The failure of a machine or a process could therefore not be signaled back to dependent processes automatically. Instead, a special process was invented which ran on each machine and was notified of process death events. Processes had to explicitly register their dependencies on other processes with this special "grim reaper" process in order to receive event-driven notifications.

- **Transparency of service**

  Because ports were tied explicitly to processes, a port defined service could not be moved from one process to another without notifying all parties. Transparent network communication was also compromised by this naming scheme. A port identifier was required to explicitly contain the network host identifier as part of its process number field. As the system expanded from one machine to one network to multiple interconnected networks this caused the port identifier to expand in size -- usually resulting in considerable reimplementation work.

- **Maximum message size**

  The limited size of messages in RIG resulted in a style of interprocess interaction in which large data objects (such as files) had to be broken up into chunks of 2K bytes or less. This constraint impacted on the efficiency of the system (by increasing the amount of message communication) and on the complexity of client/server interactions (e.g., by forcing servers to maintain state information about open files).

2.3. The evolution of RIG

CMU's Spice [8] distributed personal workstation project provided an oportunity to effectively "redo" a RIG-like system taking into account that system's limitations. The result was the Accent operating system kernel for the PERQ Systems Corporation PERQ computer.

The Accent solution to the problems present in the RIG design was based on two basic ideas:

1. **Define ports to be capabilities as well as communication objects.**

   By providing processes with capabilities to ports rather than a global identifier for them, it was possible to solve at one time the problems of protection, failure notification and transparency:

   - Protection in Accent is provided by allowing processes access only to those ports for which they have been given capabilities.

   - Processes can be notified automatically when a port disappears on which those processes are dependent because the kernel now has complete knowledge of which processes have access to each port in the system. There is no hidden communication between processes.

   - Transparency is complete because the ultimate destination of a message sent to a port is unknown by the sender. Thus transparent intermediary processes can be constructed which forward messages between groups of processes without their knowledge (either for the purpose of debugging and monitoring or for the purpose of transparent network communication).

2. **Use virtual memory to overcome limitations in the handling of large objects.**

   The use of a large address space (a luxury not possible in the design of RIG) and copy-on-write memory mapping techniques permits processes to transmit objects as large as they can directly access themselves. This allows processes such as file servers to provide access to large objects (e.g., files) through a single message exchange -- drastically reducing the number of messages sent in the system [9].

The first line of Accent code was written in April 1981. Today Accent is used at CMU in a network of 150 PERQ workstations. In addition to network operating system functions such as distributed process and file management, window management and mail systems, several applications have been built using Accent. These include research systems for distributed signal processing [10], distributed speech understanding [5] and distributed transaction processing [18]. Four separate programming environments have been built -- CommonLisp, Pascal, C and Ada -- including language support for an object-oriented remote procedure call facility [12].

3. The Accent Design

Accent is organized around the notion of a protected, message-based interprocess communication facility integrated with copy-on-write virtual memory management. Access to all services and resources, including the process management and memory management services of the operating system kernel itself, are provided through Accent's communication facility. This allows completely uniform access to such resources throughout the network. It also implies that access to kernel provided services is indistinguishable from access to process provided resources (with the exception of the interprocess communication facility itself).

3.1. Interprocess communication

The Accent interprocess communication facility is defined in terms of abstractions which, as in RIG, are called *ports* and *messages.*

The *port* is the basic transport abstraction provided by Accent. A port is a protected kernel object into which messages may be placed by processes and from which messages may be removed. A port is logically a finite length queue of messages sent by a process. Ports may have any number of senders but only one receiver. Access to a port is granted by receiving a message containing a port capability (to either send or receive).

Ports are used by processes to represent services or data structures. For example, the Accent window manager uses a port to represent a window on a bitmap display. Operations on a window are requested by a client process by sending a message to the port representing that window. The window manager process then receives that message and handles the request. Ports used in this way can be thought of as though they were capabilities to objects in a object oriented system(Jones78). The act of sending a message (and perhaps receiving a reply) corresponds to a cross-domain procedure call in a capability based system such as Hydra [2] or StarOS [11].

A *message* consists of a fixed length header and a variable size collection of typed data objects. Messages may contain both port capabilities and/or imbedded pointers as long as both are properly typed. A single message may transfer up to 2↑32 bytes of by-value data.

Messages may be sent and received either synchronously or asynchronously. A software interrupt mechanism allows a process to handle incoming messages outside the flow of normal program execution.

Figure 3-1 shows a typical message interaction. A process A sends a message to a port P2. Process A has send rights to P2 and receive rights to a port P1. At some later time, process B which has receive rights to port P2 receives that message which may in turn contain send rights to port P1 (for the purposes of sending a reply message back to process A). Process B then (optionally) replies by sending a message to P1.

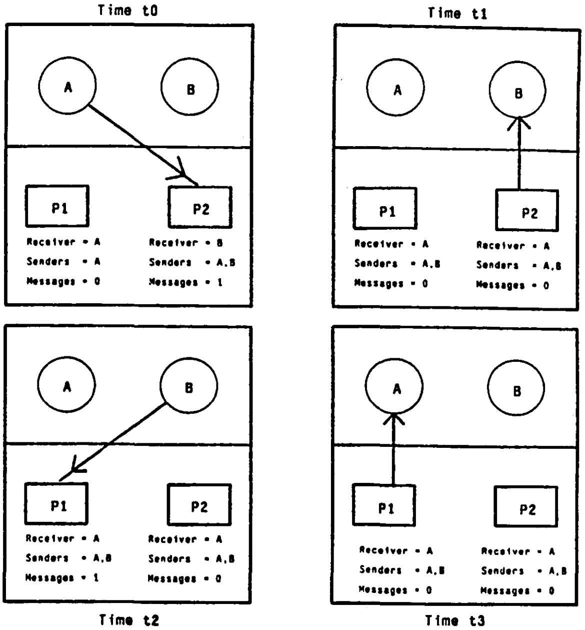


Figure 3-1: Typical message exchange

Should port P2 have been full, process A would have had the option at the point of sending the message to: (1) be suspended until the port was no longer full, (2) have the message send operation return a port full error code, or (3) have the kernel accept the message for future transmission to port P2 with the proviso that no further message can be sent by that process to P2 until the kernel sends a message to A telling it the current message has been posted.

3.2. Virtual memory support

Accent provides a 2↑32 byte paged address space for each process in the system and a 2↑32 byte paged address space for the operating system kernel. Disk pages and physical memory can be addressed by the kernel as a portion of its 2↑32 byte address space. Accent maintains a virtual memory table for each user process and for the operating system kernel. The kernel's address space is paged and all user process maps are kept in paged kernel memory. Only the kernel virtual memory table, a small kernel stack, the PERQ screen, I/O memory and those PASCAL modules required for handling the simplest form of page fault need be locked in physical memory, although in practice parts of the kernel debugger and symbol tables for locked modules are also locked to allow analysis of system errors. The total amount of kernel code and symbol table information locked is 64K bytes [9].

Whenever large amounts of data (the threshold is a system compile-time constant normally set at 1K bytes) are transmitted in a message, Accent uses memory mapping techniques rather than data copying to move information from one process to another within the same machine. The semantics of message passing in Accent imply that all data sent in a message are logically copied from one address space to another. This can be optimized by the kernel by mapping the sent data copy-on-write in both the sending and receiving processes.

Figure 3-2 shows a process A sending a large (for example 24 megabyte) message to a port P1. At the point the message is posted to P1, the part of A's address space containing the message is marked copy-on-write -- meaning any page referenced for writing will be copied and the copy placed instead into A's virtual memory table. The copy-on-write data then resides in the address space of the kernel until process B receives the message. At that point the data is removed from the address space of the kernel. By default, the operating system kernel determines where in the address space of B the newly received message data is placed. This allows the kernel to minimize memory mapping overhead. Any attempt by either A or B to change a 512 byte page of this copy-on-write data results in a copy of that page being made and placed into that process' address space.

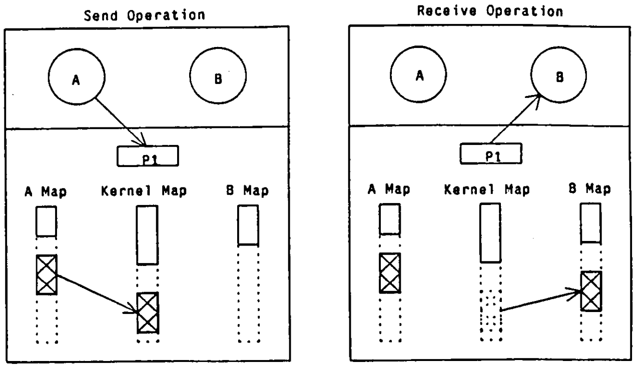


Figure 3-2: Mapping operations during message transfer

3.3. Network communication

The abstraction of communication through ports permits the distinction between access to local and remote resources to be completely invisible to a client process. In addition, Accent exploits the integration of memory management and IPC to provide a number of options in the handling of virtual memory, including the ability to allow memory to be sent copy-on-reference across a network. Each entry of an Accent virtual memory table maps a contiguous region of process virtual memory to a contiguous portion of an Accent *memory object*. A memory object is the basic unit of secondary storage in Accent. Memory objects can be contiguous physical memory (as used for the PERQ screen or I/O buffers) or a randomly addressed disk file. A memory object can also be backed not by disk or main memory, but by a process through a port. Initial references to a page of data mapped to a port are trapped by the kernel and a request for the necessary data is forwarded in a message on that port. This feature allows processes to provide the system with virtual memory that they themselves maintain (either locally or over a network connection to another machine). In this way network communication servers can provide copy-on-reference network transmission of pages in a large message.

4. Key Implementation Issues in Accent

Many of the implementation decisions made in Accent were based on experiences with RIG. Nevertheless, the addition of virtual memory and capability management to the RIG design made it unclear how the RIG experiences would extrapolate to the Accent environment.

4.1. IPC Implementation

The actual implementation of the message mechanism relied on several assumptions about the use of messages:

- the average number of messages outstanding at any given time per process would be small,
- the number of port capabilities needed by a process could vary from two to several hundred, and
- the use of simple messages (meaning messages which contained port capabilities only in their header and which contained less than a few kilobytes) would so dominate complex messages that simple messages would be an important special case.

Each of these assumptions had held true for RIG [4, 14]. It was hoped that although Accent provided a substantially different application environment than RIG, the RIG experiences would provide a reasonable prediction of Accent performance.

Given these expectations, the implementation was optimized for anticipated common cases, including:

- The assumption that there would seldom be more than one message waiting for a process at a time led to an implementation in which messages are queued in per-process rather than per-port queues.
- To allow large numbers of ports per process and fast lookup, port capabilities are represented as indexes into a global port record array stored in kernel virtual memory. Port access is protected through the use of a bitmap of process access rights kept per port (the number of processes is much less than the number of ports).
- The assumption that simple messages would be an important special case led to the addition of a field to the message header so that user processes can indicate whether or not a message is simple and thus allow special handling by the kernel.

These usage assumptions did in fact prove true for Accent. Table 4-1 demonstrates the properties of Accent message passing as measured during an active day of use.

1.01	Average probes to requested message
33.42	Average port rights held per process
14.38	Average ports owned per process
0.094	Ratio of complex to simple messages

Table 4-1: Message use statistics

4.2. Virtual Memory Implementation

The lack of sophisticated virtual memory management in RIG (and in fact in nearly all message-based systems of that era) meant that Accent could not benefit from previous experience with virtual memory use resulting from message operations. Instead, the design of Accent's virtual memory implementation grew out of simple assumptions based purely on intuition. These initial assumptions influenced the design of the Accent virtual memory implementation:

- process maps had to be compact, easy to manipulate and support sparse use of a process address space,
- the number of contiguously mapped regions of the address space would be reasonably small, and
- large amounts of memory would frequently be passed copy-on-write in messages.

The Accent process virtual memory map is maintained as a two-level indirect table terminating in linked lists of entries (see Figure 4-1). Each entry on the linked list maps a contiguous portion of process virtual memory into contiguous regions of Accent memory objects. The map is organized so that large portions can be validated, invalidated or copied without having to modify the linked lists of map entries. This is accomplished by having valid, copy-on-write and write-protect bits at each level of the table. During lookup, these bits are "ored" together. Thus all of memory can be efficiently made copy-on-write by just setting the copy-on-write bits of valid entries in level one of the process map table. Figure 4-1 illustrates the translation of a virtual address to an offset within a memory object.

Physical memory in Accent is used as a cache of secondary storage. There are no special disk buffers. Access to all information (e.g., files) is through message passing (and subsequent page faulting if necessary).

This scheme is flexible enough to be used internally by the kernel to remap portions of its own address space. An entire process virtual memory map, for example, is copied in a fork operation without physically copying the map by using Accent's copy-on-write facility. To reduce map manipulation overheads, changes caused by copy-on-write updates are recorded first in a virtual to physical address translation table (kept in physical memory) and are not incorporated into a process map until the relevant page must be written out to secondary storage.

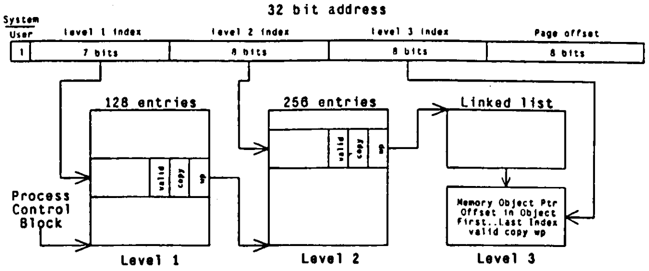


Figure 4-1: Mapping a virtual address in Accent

Copy-on-write access to memory objects is provided through the use of *shadow* memory objects which reflect page differences between a copied object and the object it shadows (which could in turn be a shadow). Disk space for newly created pages or pages written copy-on-write is allocated on an as-needed basis from a special paging area. No disk space is ever allocated to back up a process address space unless the paging algorithms need to flush a dirty page. See figure 4-2.

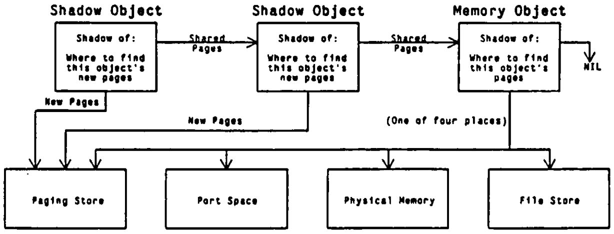


Figure 4-2: An example of memory object shadowing

Most shadow memory objects are small (under 32 pages). Most large shadows contain only a few pages of data different from the objects they shadow. These facts led to an allocation scheme in which small shadows are allocated contiguously from the paging store and larger shadows use a page map and are allocated as needed.

Overall, the basic assumptions about the use of process address space in Accent appear to hold true. The typical user process table:

- is between 1024 and 2048 bytes in size,
- contains 34-70 mapping entries, and
- maps a region of virtual memory approximately eight megabytes in extent (in PERQ PASCAL each separately compiled module occupies a distinct 128K byte region of memory) and about one to two megabytes in size.

Although all memory is passed copy-on-write from one process to another, the number of copy-on-write faults is typically small. A typical PASCAL compile/link/load cycle, for example, requires only slightly more than one copy-on-write fault per second. Clearly most of the data passed by copy in Accent is read and not written. The result is that the logical advantages of copy-on-write are obtained with costs similar to that of mapped shared memory [6].

4.3. Programming issues

One of the problems with message based systems has traditionally been the fact that existing programming languages do not support their message semantics. In RIG, a special remote procedure call function was provided called "Call" [13] which took as its arguments a message identifier, a process-port identifier, and operation arguments along with their type information. One of the early decisions in the implementation of Accent was to define all interprocess message interfaces in terms of a high-level specification language. The properties of ports allow them to be viewed as object references. The interprocess specification language is defined in terms of operations on objects. Subsystem specifications in this language are compiled by a program called Matchmaker into remote procedure call stubs for the various programming languages used in the system -- currently C, PASCAL, ADA and Common LISP. The result is that all interprocess interfaces look to the programmer as though they were procedural interfaces in each of these languages. In PASCAL, for example, the interface procedure for writing a string to a window of the screen would look like:

```
WriteString(window,string-to-be-written)
```

All Matchmaker specified calls take as their first argument the port object on which the operation is to be performed. The remote procedure call stub then packages the request in a message, sends it to the port, and waits for a reply message (if necessary). Initial access to server ports is accomplished either through inheritance (by having the parent process send port rights to its children) or by accessing a name server process (a port for which is typically passed to a process by inheritance). A complete description and specification of Matchmaker can be found in [12].

Matchmaker's specification language allows both synchronous and asynchronous calls as well as the specification of timeouts and exception handling behavior. It supports both by-value and by-value-result parameters. It allows types to be defined as well as the specification of their bit packing characteristics in the message. For the server process, Matchmaker produces routines which allow incoming messages to be decoded and server subroutines automatically invoked with the proper arguments.

The support provided by Matchmaker is similar to some of the features which have been introduced in modern languages for managing multiple tasks such as the ADA rendezvous mechanism [1]. Matchmaker, however, supports a number of different programming languages and provides a much greater range of options for synchronous and asynchronous behavior in a distributed environment.

Despite the obvious simplicity of simple "remote procedure call" style interfaces, a suprisingly high percentage of network operating system interfaces take advantage of the asynchronous form of Matchmaker interfaces. Of 225 system interfaces:

- 170 (approximately 77 percent) are synchronous,
- 45 (approximately 19 percent) are asynchronous and
- 10 (approximately 4 percent) represent exceptions.

Runtime statistics show that over 50 percent of messages actually sent during normal system execution are sent as part of asynchronous Matchmaker specified operations -- normally due to the behaviour of I/O subsystems (such as handlers for the PERQ keyboard and display) or basic system servers (such as network protocol servers).

Matchmaker server interfaces account for approximately 10 percent of the total network operating system code -- roughly 75.5k bytes out of 757k bytes. For the Accent kernel itself, the Matchmaker interface is 10280 bytes out of approximately 115k bytes. Runtime costs are considerably less. During a PASCAL compilation, for example, less than 2 percent of CPU time is due to Matchmaker interface overheads.

4.4. Key Statistics

4.4.1. Hardware and basic system performance of Accent

Table 4-2 compares the relative performance of PERQ and VAX-11/780 CPUs. Timings were performed in PASCAL on the PERQ and in C on a VAX running UNIX 4.1bsd.

PASCAL programs written for the PERQ range in overall speed from 1/5 to 1/3 the speed of comparable programs on the VAX 11/780, depending on whether 16-bit or 32-bit operations predominate. In fairness to the PERQ hardware, the underlying microengine is much faster than the PASCAL timings in table 4-2 would indicate. Microcoded operations often run as fast as or faster than equivalent VAX 11/780 assembly language. Note, for example, the relative speeds of the microcoded context switch and kernel trap operations. Moreover, instruction sets better tuned to the PERQ hardware, such as the Accent CommonLisp instruction set, run at speeds closer to 50 percent of the VAX. Nevertheless, for the purpose of gauging the performance of the Accent kernel code, which is written in PASCAL and makes heavy use of 32-bit arithmetic, pointer chasing and packed field accessing, the CPU speed of a PERQ is about 1/5 that of a VAX 11/780.

Perq	Vax	Ratio	Operation
2300ns	720ns	.31	Tick (32-bit stack local)
12us	4us	.25	Simple loop (16-bit integer)
20us	3us	.17	Simple loop (32-bit integer)
35us	20us	.57	Null procedure call/return
75us	25us	.33	Procedure call with 2 arguments
80us	400us	5.00	Context switch
132us	264us	2.00	Null kernel trap
30s	9s	.30	Baskett Puzzle Program (16-bit)
50s	10s	.20	Baskett Puzzle Program (32-bit)

Table 4-2: Comparison of Perq and Vax-11/780 operation times

4.4.2. IPC Costs

Table 4-3 shows the costs of various forms of message passing in Accent. As was previously described, Accent distinguishes between *simple* and *complex* messages to improve performance of common message operations. Simple messages are defined to be those with less than 960 bytes of in-line data that contain no pointers or port references (other than those in the message header). Other messages are considered complex. The times for complex messages listed in the table were measured for messages containing one pointer to 1024 bytes of data. The observed ratio of simple to complex messages in Accent is approximately 12-to-1.

Time	IPC Operation
1.15	Simple message send
1.35	Simple message receive
10.	Complex message send (1024 bytes)
10.	Complex message receive (1024 bytes)

Table 4-3: IPC operation times in milliseconds

The average number of messages per second observed during periods of heavy standard version use (e.g., compilation) is less than 30. There were 67378 simple messages and 4279 complex messages sent during one measurement of three hours of editing, network file access, and text formatting, an average of less than eight per second [9].

4.4.3. Accessing file data

One of the reasons for the relatively low message rate of message exchange in Accent is the heavy reliance on virtual memory mapping techniques for transferring large amounts of data in messages. A process making a request for a large file typically receives the entire file in a single message sent back from a file server process. As a result, all file access in Accent is mediated through the memory management system. There are no separate file buffers maintained by the system or special operations required for file access versus access to other forms of process mapped memory. By contrast, in RIG the same operation would have required as many message exchanges between client and server as there were pages in the file.

Table 4-4 shows the costs associated with reading a 56K byte file under UNIX 4.1bsd on a VAX 11/780 with a 30 millisecond average access time Fujitsu disk and under the standard version of Accent with a 30 millisecond average access time MAXSTORE drive.

The measured cost of a file access in Accent as shown in table 4-4 is due, in part, to the cost of a disk write to update the file access time. This disk write is unbuffered in Accent and thus is included in the file request time. The Unix disk write associated with an open is buffered and is excluded from the open/close time.

Accent file access speed is limited by the basic fault time of about four milliseconds (see table 4-5), the average number of consecutive file pages on a disk track and the cost of making new VP entries. Its page size is only 512 bytes, in contrast to 1024 bytes for 4.1bsd and 4096 or 8192 for 4.2bsd.

System	Time	Operation
Accent	66	Request file from server
UNIX 4.1	5-10	Open/close
Accent	5-10	Read a page (512 bytes)
UNIX 4.1	16-18	Read a page (1024 bytes)
UNIX 4.2	16-18	Read a page (4096 bytes)

Table 4-4: File access times in milliseconds

Once mapped, file access in Accent ranges from somewhat faster than 4.1bsd to slightly slower, depending on the locality of file pages. 4.2bsd file access [15] is considerably faster than either 4.1bsd or Accent. This increase in speed appears to be due almost entirely to the larger (typically 4096 byte) file page size. The actual number of disk I/O operations per second under 4.2 is almost identical to 4.1, about 50-60 per second, and appears to be bounded by the rotational speed of the disk (60 revolutions per second).

4.4.4. Fault handling and copy-on-write

Table 4-5 summarizes the results from test programs that caused 100,000 instances of a variety of memory fault types. It shows the average total times required to handle single faults.

Total	Type of fault
0.623	Null fault
3.355	Read fault, zero fill
3.704	Write fault, zero fill
3.760	Read fault, memory fill, small file
4.504	Read fault, memory fill, large file
3.833	Write fault, CopyOnWrite copy

Table 4-5: Fault handling times in milliseconds

Overall, the costs of copy-on-write memory management are nearly identical to that of by-reference memory mapping. Less than 0.01 percent of the total time associated with an entire rebuilding of the operating system and user programs from source is used to handle copy-on-write faults [9].

5. Mach: Adapting Accent to Multiprocessors

Accent went beyond demonstrating the feasibility of the message passing approach to building a distributed system. Experience with Accent showed that a message based network operating system, properly designed, can compete with more traditional operating system organizations. The advantages of this approach are system extensibility, protection and network transparency.

By the fall of 1984, however, it became apparent that, without a new hardware base, Accent would eventually follow RIG into oblivion. Hastening this process of electronic decay was Accent's inability to completely absorb the ever burgening body of UNIX developed software both at CMU and elsewhere -- despite the existence of a "UNIX compatibility" package.

Mach was conceived as an Accent-like operating system which would provide complete UNIX compatibility. It was also designed to better accommodate the kind of general purpose shared-memory multiprocessors which appear to be on their way to becoming the successors to traditional general purpose uniprocessor workstations and timesharing systems.

5.1. The design of Mach

The design of Mach differs from that of Accent in several crucial ways:

- The Accent notion of a process, which like RIG is an address space and single program counter, was split into two new concepts:
    1. a *task*, which is the basic unit of resource allocation including a paged address space, protected access to system resources (such as processors, ports and memory), and
    2. a *thread*, which is the basic unit of CPU utilization.
- A facility for handling a form of structured sharing of read/write memory between tasks in the same family tree was added to allow finer granularity synchronization than could be achieved with a kernel provided mechanism.
- The Mach IPC facility was further simplified. This came about as the logical result of using thread mechanisms to handle some forms of asynchrony and error handling (much as was done in the V Kernel [7]).
- The notion of memory object was generalized to allow general purpose user-state external pager tasks to be built.

These design modifications are a consequence of handling shared-memory multiprocessor architectures. Accent provided no tool for fine grain synchronization or lightweight processes. Both are important for effective use of multiprocessor cycles in a variety of applications.

Despite these changes, the basic features which allowed Accent to provide uniform access to both local and network resources are still in place. This allows networks of multiprocessors or of multiprocessors and uniprocessors to be built using the same basic system abstractions. As in Accent, operations on all Mach objects other than messages are performed by sending messages to ports which are used to represent them. For example, the act of creating a task or thread returns access rights to the port which represents the new object and which can be used to manipulate it. A thread can suspend another thread by sending a suspend message to that thread's *thread port*, even across a network boundary.

Tasks are related to each other in a tree structure by task creation operations. Virtual memory may be marked as inheritable to a task's children. Memory regions may be inherited read-write, copy-on-write or not at all. A standard UNIX fork operation, for example, takes the form of a task with one thread creating a child task with a similar single thread of control and all its memory shared copy-on-write.

The notions of multiple threads of control within a task and limited sharing between task allows Mach to provide three levels of synchronization and communication: fine grain, intra-application interprocess communication and inter-application interprocess communication.

Fine grain communication is performed on memory shared either within a task or between related tasks. Mach provides a library to support synchronization on shared memory to avoid the cost of kernel trap operations on short-term locks. Network read/write shared memory is not provided by the kernel, but is potentially implementable by a user-state process acting as an external object pager (see discussion of object pagers below).

Intra-application inter-thread communication is performed using the standard Send and Receive ports primitives but can be implemented more efficiently in the presence of shared libraries and memory. By the nature of the abstractions, threads can ignore the difference between intra-application communication and inter-application communication.

Inter-application communication requires the intervention of the Mach kernel to provide protection. As in Accent, large amounts of data in messages may be mapped copy-on-write from one address space to another rather than copied. Data forwarded in messages over the network can be transmitted on reference rather than all at once at the discretion of the network server.

5.2. Implementation

5.2.1. Virtual memory modifications

While system analysis indicated that the basic Accent virtual memory scheme worked well, it also demonstrated that the data structure used to represent an Accent process map -- a two-level indirect table terminated in linked lists of mapping descriptors -- was unnecessarily complicated. Because nearly all operations on maps are sequential and maps seldom get very large, Mach implements task address maps as simple ordered lists of mapping descriptors. Each descriptor maps a range of virtual addresses to a range of bytes in a memory object. The only non-sequential operation -- lookup events due primarily to memory faults -- is sped by the use of hints based on previous lookup requests.

Another innovation of Mach over Accent is in the use of *sharing maps* to represent read/write shared regions between tasks. A Mach mapping descriptor may point either directly to a memory object (which can then only be shared copy-on-write) or indirectly to memory objects through a sharing map. A sharing map is simply an address map which maps a range of virtual addresses shared by at least two task address maps. All operations on tasks maps in a shared range of addresses are performed through indirection on sharing maps.

Overall, the Mach data structures are simpler, more compact and more expressive than those of Accent. A Mach address map can be thought of as a simple run-length encoding of a process address space. A typical UNIX-style process can be expressed in less than 100 bytes.

5.2.2. Mach IPC

The introduction of the notion of tasks and threads into Mach necessitated some changes to Accent's basic IPC facility. Port access rights in Mach are owned by a task. All threads within a task may therefore send or receive messages on that task's ports.

ՙhe availability of threads to manage asynchronous activities simplified handling of software interrupts. Moreover, several message options, such as message priorities and the ability to preview the contents of a message before it had to be received, had been found to be largely unused for their intended purpose in Accent and have been removed.

5.2.3. Managing hardware diversity

Mach was intended from the outset to handle a wide diversity of both uniprocessor and multiprocessor hardware. For example, Mach provides a task memory sharing and a thread memory sharing model for multiprocessor memory synchronization. This allows Mach to support both multiprocessors which support full memory sharing with cache consistency as well as machines with only partial sharing or explicit memory caching. In practice, the system already is configured to handle a wide range of uniprocessor and multiprocessor VAX configurations. The same binary kernel image is used on both uniprocessor and multiprocessor systems.

Mach also handles another form of diversity. Messages, because they contain tagged data, are transformed from one machine data format to another by network servers. Properly typed Matchmaker interfaces allow programs written on an RT PC to communicate with VAX applications despite different byte ordering, data packing and data format conventions. There are, however, limits on this form of machine independence. For example, no attempt is made to preserve precision of floating point numbers converted from one form to another.

5.2.4. Confronting UNIX

One mechanism for ensuring Mach's survival in the face of a flood of UNIX based software is to make certain that it is compatible with an existing UNIX environment. This was achieved by building Mach to allow UNIX 4.3bsd system calls to be handled in much the same way they would be handled in a completely native system. The Mach kernel effectively supplants the basic system interface functions of the UNIX 4.3bsd kernel: trap handling, scheduling, multiprocessor synchronization, virtual memory management and interprocess communication. 4.3bsd functions are provided by kernel-state processes which are scheduled by the Mach kernel and share communication queues with it. Work is now underway to remove non-Mach UNIX functionality from kernel-state and provide these services through user-state processes.

6. Conclusions

The evolution of network operating systems from RIG through Mach was, in a sense, driven by the evolution of distributed computer systems from small networks of minicomputers in the middle 1970s to large networks of personal workstations and mainframes in the early 1980s to networks of uniprocessor and multiprocessor systems today. Not suprisingly, the basic software primitives of Mach -- task, thread, port, message and memory object -- parallel the hardware abstractions which characterize modern distributed systems -- nodes, processors, network channels, packets and primary and secondary memory. Experiences, both good and bad, with RIG and Accent have played an important role in determining the exact definition of the Mach mechanisms and their implementation.

7. Acknowledgements

In addition to anything the author may have done, the heroes of the RIG kernel development were Gene Ball and Ilya Gertner. Jerry Feldman was in large part responsible for the initial RIG design and the system's name. The Accent development team included George Robertson and Gene Ball as well as the author. Keith Lantz and Sam Harbison made notable contributions to the design. Mary Shaw contributed the name. Others contributed greatly to Accent's evolution: particularly Doug Philips, Jeff Eppinger, Robert Sansom, Robert Fitzgerald, David Golub, Mike Jones and Mary Thompson. Matchmaker could not have come into existence without the aid of Mary Thompson, Mike Jones, Rob MacLachlin and Keith Wright. Mach was the brainchild of many including Avie Tevanian, Mike Young and Bob Baron. Dario Giuse came up with the name.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

References

[1] Department of Defense.
Preliminary Ada Reference Manual
1979.

[2] Almes, G. and G. Robertson.
An Extensible File System for Hydra.
In *Proc. 3rd International Conference on Software Engineering*. IEEE, May, 1978.

[3] Ball, J.E., J.A. Feldman, J.R. Low, R.F. Rashid, and P.D. Rovner.
RIG, Rochester's Intelligent Gateway: System overview.
IEEE Transactions on Software Engineering 2(4):321-328, December, 1976.

[4] Ball, J.E., E. Burke, I. Gertner, K.A. Lantz and R.F. Rashid.
Perspectives on Message-Based Distributed Computing.
In *Proc. 1979 Networking Symposium*, pages 46-51. IEEE, December, 1979.

[5] Bisiani, R., Alleva, F., Forin, A. and R. Lerner.
Agora: A Distributed System Architecture for Speech Recognition.
In *International Conference on Acoustics, Speech and Signal Processing*. IEEE, April, 1986.

[6] Bobrow, D.G., Burchfiel, J.D., Murphy, D.L. and Tomlinson, R.S.
TENEX, a paged time sharing system for the PDP-10.
Communications of the ACM 15(3):135-143, March, 1972.

[7] Cheriton, D.R. and W. Zwaenepoel.
The Distributed V Kernel and its Performance for Diskless Workstations.
In *Proc. 9th Symposium on Operating Systems Principles*, pages 128-139. ACM, October, 1983.

[8] Spice Project.
Proposal for a joint effort in personal scientific computing.
Technical Report , Computer Science Department, Carnegie-Mellon University, August, 1979.

[9] Fitzgerald, R. and R. F. Rashid.
The integration of Virtual Memory Management and Interprocess Communication in Accent.
ACM Transactions on Computer Systems 4(2):, May, 1986.

[10] Hornig, D.A.
Automatic Partitioning and Scheduling on a Network of Personal Computers.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, November, 1984.

[11] Jones, A.K., R.J. Chansler, I.E. Durham, K. Schwans and S. Vegdahl.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proc. 7th Symposium on Operating Systems Principles*, pages 117-129. ACM, December, 1979.

[12] Jones, M.B., R.F. Rashid and M. Thompson.
MatchMaker: An Interprocess Specification Language.
In *ACM Conference on Principles of Programming Languages*. ACM, January, 1985.

[13] Lantz, K.A.
Uniform Interfaces for Distributed Systems.
PhD thesis, University of Rochester, May, 1980.

[14] Lantz, K.A., K.D. Gradischnig, J.A. Feldman and R.F. Rashid.
Rochester's Intelligent Gateway.
Computer 15(10):54-68, October, 1982.

[15] McKusick, M.K., W.N. Joy, S.L. Leach and R.S. Fabry.
A Fast File System for UNIX.
ACM Transactions on Computer Systems 2(3):181-197 , August, 1984.

[16] Rashid, R.F. and G. Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 64-75. ACM, December, 1981.

[17] R.F. Rashid.
The Accent Kernel Interface Manual.
Technical Report , Department of Computer Science, Carnegie-Mellon University, January, 1983.

[18] Spector, A.Z. et al.
Support for Distributed Transactions in the TABS Prototype.
In *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, pages 186-206. October, 1984.

LOAD BALANCING IN NEST: A NETWORK OF WORKSTATIONS

Ahmed K. Ezzat

Computing Systems Technology Research Department
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes a fully distributed load balancing model for scheduling processes in a distributed system. The model consists of two modules, information policy module and control policy module. These modules are replicated at every host and they cooperate together to provide a network-wide objective function. The model takes into consideration the imperfect knowledge about the system state, and assumes no a priori knowledge of the new processes description. The scheme is fully distributed, and the algorithm is adaptive and stable. The remote execution phase of the model has been implemented in the NEST distributed system. We give the essential details of the implementation and initial results on its performance. The scheme was tested under a realistic environment using an adapted version of an internal benchmark. The results confirm the feasibility and effectiveness of a simple policy to reduce the response time by dynamically taking advantage of load differences in the network.

1. INTRODUCTION

One of the interesting problems in the design of any distributed computing system is how to improve the overall system performance (i.e., throughput, response time, etc.) by dynamically distributing the load over the entire system. This should be done in a transparent way so that physical boundaries between resources on different hosts are hidden from system users. Optimal load balancing is expensive computationally, and would require a priori knowledge of the run-time characteristics of the workload in the system. Alternatively, it is desirable to have a suboptimal algorithm which requires less information about the workload, can deal with the imperfect knowledge about the system state, and is inexpensive to use.

In this paper, we present an implementation of a decentralized control dynamic load balancing algorithm in NEST [1, 2, 12], a multiprocessor system consisting of a network of AT&T 3B2 computers interconnected with AT&T 3BNET (which is an Ethernet compatible 10 megabit/second local area network). The purpose of this work is to validate the assumptions made in our scheduling model; furthermore, a prototype implementation would give us more insight into the feasibility, effectiveness, and performance of load balancing algorithms under sophisticated and realistic environment. We report our experience on both of these counts.

In the following section we review related research on load balancing. Section 3 gives an overview of our scheduling model. Section 4 discusses the load balancing algorithm in detail. Section 5 presents a brief background about the NEST system. Section 6 describes the benchmark used in evaluating the algorithm. Section 7 discusses the algorithm implementation considerations and gives an initial performance evaluation and its implications. We compare our work to other related systems in Section 8, and give some concluding remarks in Section 9.

2. BACKGROUND

Numerous studies have addressed the problem of load balancing in distributed systems. We found it convenient to review previous work in load balancing by dividing the problem into its three fundamental components and discussing the different approaches for each one individually. These components are namely load balancing mechanism, policy, and cost function formulation. In general these

CH2345-7/86/0000/1138$01.00 © 1986 IEEE

approaches for the different components are not mutually exclusive.

Load balancing can either have a centralized or a decentralized mechanism. In centralized mechanism schemes, the decision making process of assigning processes to different hosts in the network is done at a central controller where the system state is maintained. Stone [28] presented a centralized resource allocation algorithm based on the Ford-Fulkerson maxflow-mincut algorithm in order to maximize flow in commodity networks. Chow and Kohler [8] used a central dispatcher to allocate processes to hosts in order to minimize the mean response time in the network. In decentralized mechanism schemes, processes are scheduled at the arrival host. This approach is faster in making decisions, however it requires more communication overhead to update all hosts with the system state. Also, the policy in such a scheme has to be able to handle inconsistent hosts' view of the system state as a result of communication delays [13,16]. Stankovic [27] presented a heuristic for the effective cooperation of multiple decentralized components of a process scheduling function. The heuristic itself is based on the application of Bayesian decision theory as a systematic approach to complex decision making under conditions of imperfect knowledge.

Load balancing policy can be either static or dynamic. In static load balancing policies, assignment of processes to hosts are done based on a predetermined average behavior of the network and not on the current system state. Tantawi and Towsley [29] presented two decentralized static algorithms. The load balancing scheme is formulated as a nonlinear programming problem. Alternatively, in dynamic policies assignment of processes are done based on the current estimate of the system state. These policies would assign the process upon creation or external arrival permanently to what appears to be the best host at that time. A non-preemptive dynamic policy does not move a process even though its run-time characteristics may prove to be poor later. For example, Bryant and Finkel [5] developed an algorithm for point-to-point interconnection networks. The algorithm uses the current load estimate in forming a set of host pairs that differ greatly in load. Once a pair is formed, the more heavily loaded host decides which processes, if any, to send to the other host. This is based on the greatest expected improvement in the response ratio of the migrant process. On the other hand, a preemptive dynamic policy would reassign the process whenever the system state changes and the run-time characteristics of the process indicates that it would be better to execute the process at some other host.

Finally, a cost function is needed to represent the objective function of the load balancing policy. Depending on the formulation, the policy would solve for the minimum or maximum cost function in the system. Algorithms for process scheduling can be grouped into four main general approaches: mathematical programming [9], graph theoretic [4,7,28,31], queuing theory [19,22,25], and heuristics [5,10,18,20].

In this paper, we propose a decentralized dynamic preemptive scheduling model. The model assumes no a priori knowledge about the execution time of processes. Different hosts in the network may have different views of the system state; it is considered a natural extension to the scheduling function in a single host operating system. The nonpreemptive phase of the model has been implemented under the NEST system. Initial results are presented and its implications on the model are discussed.

3. OVERVIEW OF THE MODEL

Our process scheduling model consists of two modules, the *information policy module* and the *control policy module*. These modules are replicated at every host and they cooperate together to provide a network-wide objective function. The information policy module sends the local host's workload state to the network, and receives the corresponding messages from the network. This information is used by the control policy module in assigning processes to hosts in the network. The control policy module is part of the local host's copy of the distributed operating system and is considered a natural extension to the traditional scheduler in the single host case. Since process scheduling is a critical operating system function, any algorithm must be inexpensive. This precludes the use of any sophisticated algorithms including many potential solutions based on mathematical programming. We envision as a part of the distributed operating system that every host will have a scheduling entity called the "Network Scheduler," (see Fig. 1), to perform three functions:

- **Remote execution** — The remote execution function assigns each newly arrived (or created) process using global strategy to the "run queue" of one of the hosts in the network.
- **Local scheduling** — This function schedules the processor among the processes in the local run queue using a local scheduling discipline (e.g., such as Round Robin with interval Δt).
- **Process migration** — The process migration function, at periodic intervals ($N\Delta t$ where N is very large), uses the estimate of the local load and the estimate of the average load in the network to determine whether to migrate one or more processes to the run queue of another host with less load.

Both remote execution and process migration functions use the information gathered by the information policy module about the network state in making assignment decisions. Issues with the process migration function include the criteria on which to select a process for reassignment, how to prevent processor thrashing (i.e., spending most of the time reassigning processes rather than doing actual work), and finally how often to invoke this function. The process migration function is not addressed further in this paper.

Both control and information policy modules use the communication subsystem of the network to perform their functions. The control policy module sends a *request message* to the target host that describes the process to be initiated and the information policy module sends a *status message* that contains information on that host's workload state.

4. REMOTE EXECUTION

In this section we present an algorithm that implements the *remote execution* function of the Network Scheduler as presented in the previous section. Processes arrive at different hosts in an unpredictable manner, and no a priori knowledge about the process description is assumed. The objective of this algorithm is to balance the *processing load* over the entire network in order to improve the response time with a minimum number of processes movement. Given that this function of the Network Scheduler is performed, the same approach (with suitable constraints) can be adapted to handle processes in execution (i.e., the *process migration* function).

State Representation

Because of the rapid fluctuations in the instantaneous value of the processor load and the communication delays, transmitting the instantaneous value of the host load may result in a nonproductive processes movement. To avoid such situation, we suggest that the instantaneous local host state parameters are taken every T_s units of time, and that it is averaged over a period of time nT_s, where n is the number of samples and defines the window size over which we calculate the state distribution parameters. Every T_u time units, the information policy module calculates the average local host state parameters during the most recent window of time. The new calculated local state is sent to the rest of the network if the calculation differs significantly from the last value transmitted. Both n, T_s, and T_u values are affected by factors such as the speed of the communication subsystem and the nature of the applications used by the network.

This scheme contributes to the system stability as sudden changes in the instantaneous value of the host state would not be completely reflected in its representation until it is proven to be a steady trend rather than a temporary one.

Parameters of the Algorithm

Let M be the number of compute servers available to a specific client host. Let Ψ_j be the processing cost of assigning a new process to host j. Let β be the bias used in the algorithm. The bias is defined as the percentage saving of the local cost below which a process is not moved to another host; it is a function of the load unbalance in the system as we shall see later. Let f_1, f_2 be the weighting factors used in the normalized response time

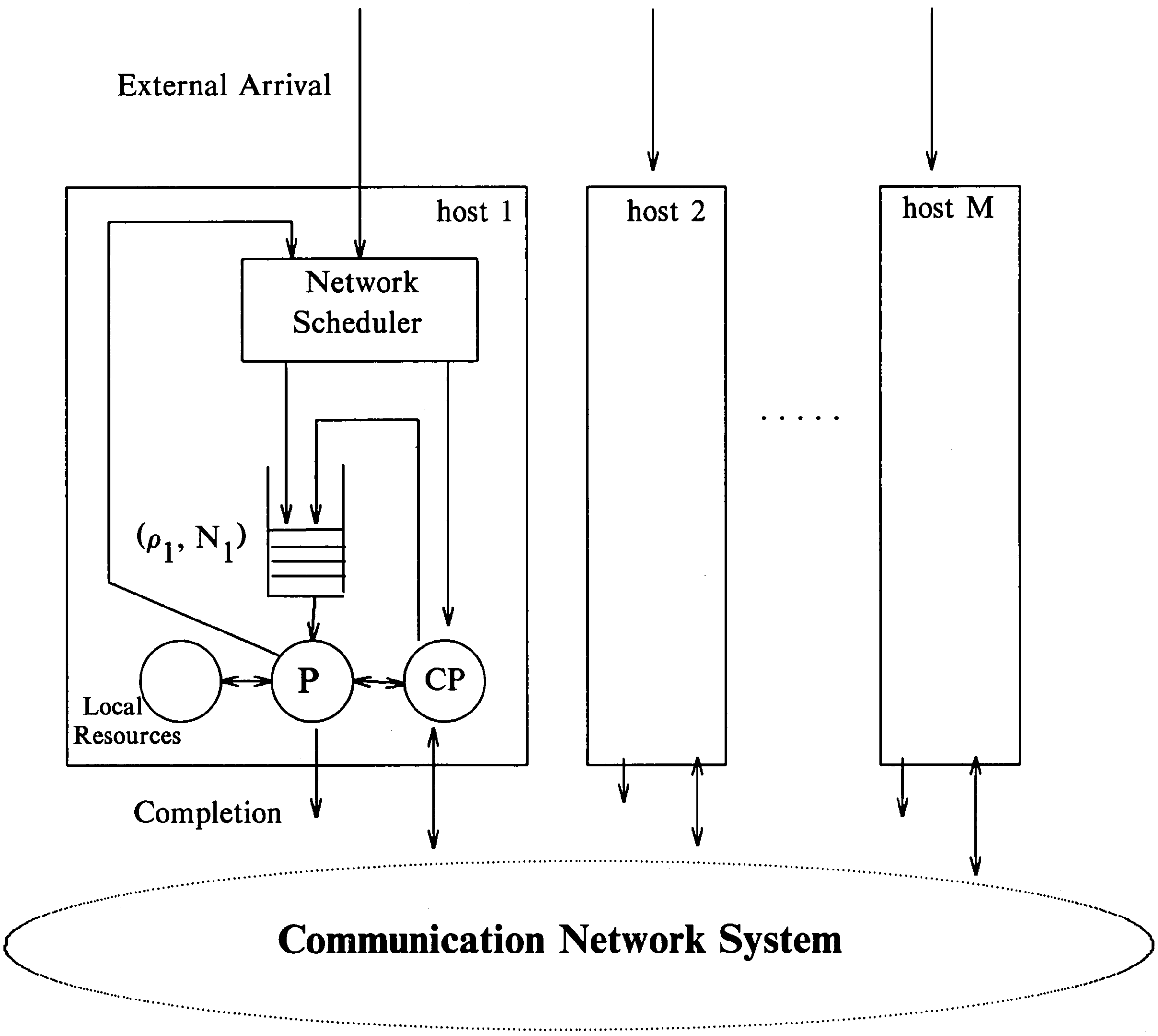


Figure 1. A Distributed Computer Model

formula. The information policy module sends the following state parameters to the rest of the network:

$\bar{N}_j \equiv$ the last updated estimate for the average number of processes at host j.

$\bar{\rho}_j \equiv$ the last updated estimate for the average CPU utilization at host j.

The NRT Algorithm

This algorithm is designed to minimize the *normalized response time* (NRT) for all processes in the network. NRT is defined as the weighted sum of the average number of processes ($\bar{N}$) and the average number of processes during the busy period ($\bar{N}/\bar{\rho}$) over the most recent window in time. The combination of these two parameters is a better description of the load probability density function distribution than just any of the two parameters individually. This is because they represent the average and the peak of the load distribution. Notice that we do not distinguish between user processes, system processes (i.e., swapper, etc.), and network server processes (i.e., network server, file server, or dispatcher server). The cost function Ψ_j to assign a process to host j is defined as the ratio of time needed to complete this process at host j to the time required by the process from host j; ($\Psi_j \geqslant 1$) due to the fact that the process may be sharing the processor with other processes.

Assume that an external process arrives at host i. The following steps are carried out by the Network Scheduler at host i.

Step 1. Calculate the cost function for assigning the process locally, Ψ_i.

$$\overline{NRT_i} = \begin{cases} 0, & \text{if } \overline{\rho}_i = 0. \\ f_1\overline{N}_i + f_2(\frac{\overline{N}_i}{\overline{\rho}_i}), & \textit{otherwise.} \end{cases}$$

$$\Psi_i = \overline{NRT_i}+1$$

Step 2. Calculate the cost function Ψ_j for every host j in the network.

$$\Psi_j = \overline{NRT_j}+1$$
$$\text{for } j = 1, 2, .., i-1, i+1, .. , M$$

Calculate the average cost function in the network, Ψ_a.

$$\Psi_a = \frac{1}{M} \; (\; (\sum_{j=1}^{M} \overline{NRT_j}) + 1 \;)$$

If the local cost is less than or equal to the average cost in the network ($\Psi_i \leqslant \Psi_a$), the process is assigned to the local run queue and the algorithm terminates. This comparison is intended to minimize process movement and contribute to the scheduler stability.

Step 3. Calculate the bias against process movement, δ.

$$\delta = \beta\Psi_i$$

Find the set of hosts in the network having the minimum cost function Ψ_m. If Ψ_m is less than the local cost Ψ_i by at least the bias against process movement ($\Psi_m \leqslant \Psi_i - \delta$), the process is assigned to host m, otherwise it is assigned to the local host. If there is more than one host with minimum cost, one is selected at random.

We conclude this section with a note about incorporating the processor capacity in our algorithm. Let C_j denote the CPU capacity of host j (measured in any commonly units). If $C_j = 2C_i$, and the two hosts have the same load, then the expected response time of host j is half that of host i. Then the normalized cost function formula at host j, $\overline{\Psi}_j$, is defined as $\overline{\Psi}_j = \Psi_j/C_j$. In all results shown in this paper, all processors have the same capacity; therefore $\overline{\Psi} = \Psi$.

5. OVERVIEW OF THE NEST SYSTEM

In NEST, we consider a computing environment that consists of a network of highly autonomous yet cooperative personal computer workstations and shared servers. These computers are interconnected via AT&T 3BNET hardware, which is an Ethernet compatible 10 megabit/second local area network. An important aspect of NEST is to provide processor sharing by creating a pool of compute servers in the network that may be used by the workstations to supplement their computing needs. Some processors are permanently designated to be compute servers. In addition, through an advertisement mechanism [12], any workstation may make itself temporarily available for a specific duration of time to be used as a compute server. Each processor runs the NEST system [1,2,12]. The basic software building block for developing parallel and distributed applications is the "rexec" primitive [1] which provides the execution location independent remote execution capability. At the user level, the remote execution can be initiated as follows,

```
rexec [-s host-name]
      "command string"
```

where host-name specifies the target compute server at which the "command string" to be executed. The "command string" may include a compound operation such as pipe, standard input/output redirection, background execution, etc. It is also possible to initiate the remote execution of a "command string" at the program level by *execing* the *rexec* from a C program [17]. This *rexec* capability allows processes to be offloaded to a compute server and preserves the execution environment of these processes as if they were still executing locally at the originating host. The execution location independent capability is achieved by preserving the process view of the file system (i.e., treating the CPU and the file system as a separate and independent resources), parent-child relationships, process group relationship, and process signaling across machine boundaries in a transparent way.

In NEST, each workstation has its own file tree which is configured from the available network file resources. It is likely that system commands such as *nroff*, *cc*, etc. are replicated at every machine. The SWITCH variable capability gives the user a *system controlled capability* to access absolute pathnames either from the local tree or the compute server tree. This capability is used to alleviate copying system files across the network. Another example of using the SWITH capability is temporary files which would be more efficiently created at the machine on which the process is executing. This is because by default temporary files are deleted at the end of the operation.

To have a feeling for the penalty of remotely executing a process in NEST as well as to demonstrate the use of the SWITCH capability, we measured the response time for both local and remote compilation. Table 1 depicts the penalty of compiling a 26 Kbytes source file remotely using different values for the SWITCH variable. The penalty includes both overheads of remotely invoking the compilation operation and accessing/creating remote files during the compilation. When the SWITCH was set to NULL, all system files (cc, comp, cpp, as, ld, etc.), temporary files (symbol table file, etc.), include files in the source file, and source and output files were accessed or created on the originating machine. The penalty in this case for remotely compiling the source file was 55.6% relative to the local compilation. When the SWITCH was set to "/bin:/lib:/tmp", only input and output files were accessed or created on the originating machine. The penalty in this case dropped to 6% relative to the local compilation. For more details about the remote execution and the SWITCH capability refer to [1].

6. WORKLOAD BENCHMARK

In this section, we present an adapted version of an internal benchmark [15] which is widely used to give a picture of the global UNIX[†] system performance rather than measurements of individual components (e.g., CPU, file system). The benchmark uses a scaling technique which allows measuring the degree to which a time sharing system manages concurrent, competing demands for system resources. The workload is represented by a script of interactive commands. A model was constructed by measuring three days of activity on a UNIX system in a development environment. The script workload was constructed so that resources are consumed in the same proportions as the real workload model. The representative script consists of atomic sequence of commands including compile, load, execute, text format, file edit, spelling check, searching for and copying files, etc. Multiple scripts can be initiated in parallel to increase the level of concurrency. In this case, the ordering of the set of commands in the different scripts are permuted in order to avoid synchronization problem, i.e., all simulated users execute the same command at the same time. This approach allows each instance of the workload to be different, while imposing the same overall amount of work on the system.

In measuring performance for our implemented load balancing algorithm, different loads (i.e., light, moderate, and heavy) are represented by different number of scripts executing concurrently. Performance measures include the average host response time (a script is the unit of load rather than individual commands), average system response time, percentage of commands remotely executed, and the CPU utilization over the life time of the workload.

7. IMPLEMENTATION OF NRT ON NEST

Host's State Measurement

The mechanism used for measuring the local processor load parameters is turned on automatically whenever a processor advertises

[†] UNIX is a trademark of AT&T Bell Laboratories

Execution Mode		Penalty
Local		0.0%
Remote	SWITCH= NULL	55.6%
	SWITCH= /tmp	33.6%
	SWITCH= /bin:/lib	29.0%
	SWITCH= /bin:/lib:/tmp	6%

TABLE 1. Compilation Penalty Measurements

itself as a compute server and is turned off otherwise.

Because parameters needed in our algorithm $(\bar{N}, \bar{p})$ are already computed and available in an accounting data structure in the kernel, we found it convenient to sample this information every second (i.e., select T_s= 1*sec*) in the clock interrupt service routine. In order to tune the other parameters of the information policy module, we have taken measurements of the overhead incurred by the NEST system to move a process and resume its execution at a compute server. For a source program of size 2*Kbytes*, the difference between local and remote compilation is 2.5 seconds. This overhead includes moving the process environment, process view of the file system; it also include copying user source program, system programs (i.e., cc, cpp, ld, etc.), and output module across the network. This would suggest [11] that an update rate of 3 seconds (T_u= 3*sec*), and window size of 6 seconds (n= 6 samples) would help in alleviating the effect of load fluctuations between the time of selecting a compute server and resuming execution at the compute server.

Every T_u= 3*sec*, the clock interrupt service routine wakesup a *state server* kernel process to calculate the local state parameters. If it is different significantly from the last broadcasted value, a state update message is sent to the list of clients previously advertised by this compute server.

On the reception of a state update message by the network server [12] at a client host, the new server's state is updated in the appropriate entry at the server state kernel table. This information is used by the load balancing policy to assign an *rexeced* process to a host in the network.

Bias Selection

The bias β value as expected is very much dependent on the underlying system and the load differences in the network. In a network where the load differences among the different hosts is small, processes movement should be discouraged as the overhead of moving the process may not counterbalance the gain from executing the process remotely. This means a higher bias for fairly balanced networks than that needed for unbalanced loaded networks. Optimal bias values for typical balanced and unbalanced loads were deducted from experiments discussed below and then plugged back into the algorithm as a list. The algorithm indexes the appropriate bias value from the list depending upon the current load differences in the network. This modification does not impose significant overhead as these information are simple to extract.

Policy Implementation

The *rexec* command discussed earlier has been modified as follows,

```
rexec [-s host-name]
      [-g group-name]
      "command string"
```

Using only the "s" option, the system will behave exactly as discussed earlier. If no options are specified, the system applies the NRT algorithm to assign the "command string" request either locally or to any of the available compute servers. If the option "g" is specified, the system applies the NRT algorithm to assign the "command string" request to a host from the group of hosts defined by the alias group-name. Information about the compute servers' load parameters are accessed from the kernel server state table.

This table is maintained by the *state server* process as discused before.

System Resources Considerations

In a case where system resources (e.g., process table entry) are near fully utilized, newly forked processes are forced to rexec remotely if compute servers are available.

Initial Performance Evaluation

In this section, we present results of the remote execution function of our process scheduling model. The load balancing algorithm was tested in a network of three AT&T 3B2 computers interconnected via 3BNET hardware. All computers are functioning as a client and a compute server simultaneously. Different loads in the network (light, moderate, and heavy) are represented by different number of scripts executing concurrently. The selected set of loads have a wide range of load differences among the hosts in the network. The three sets of load parameters are shown in Table 2.

First, we note that in NEST all objects, except processes, are static, i.e., they do not migrate across host boundaries. Examples of such objects are files, devices, etc. These objects however can be shared by some or all hosts in the network. Access to such objects may be originated at any host, but it is performed in the host where the object physically resides using a pool of file servers. In our experiment, we generated enough number of file servers so that they do not contribute degradation to the response time in the system.

The results shown here used the following set of parameters: sampling time $T_s = 1sec$, number of samples $n = 6$, update rate $T_u = 3sec$, and weighting factors for the NRT formula $f_1 = f_2 = 0.5$. In all tests, in order to maintain the same load throughout the tests, the scripts were executed in a loop, i.e., upon completion each script restarts.

We are presenting here three set of tests, e.g., light, moderate, and heavy loads, each including two phases. In the first phase, the load balancing algorithm is enabled and the bias β is varied over a wide range of values. Performance measurements includes the average host response time, average system response time, percentage of processes movement, and the CPU utilization. In the second phase, we run the same set of tests with the load balancing algorithm disabled, i.e., commands in the scripts were executed locally and recorded the same performance measurements. Notice that in the second phase the SWITCH value is irrelevant. The purpose of the second phase is to give an upper bound to our load balancing algorithm. The tests proceeded as follows:

For light balanced load with the SWITCH set to NULL, whose results is shown in Table 3, the algorithm needed high bias 75% to prohibit nonproductive processes movement. The gain achieved at bias 75% is also negligible (less than 1%) as was expected. In general, at low bias, processes movement was nonproductive and degraded the average system response time relative to the nonload balancing case.

For a moderate unbalanced load with the SWITCH set to NULL, whose results is shown in Table 4, the test demonstrates the potential advantage of remote process assignment. In spite of the fact that the load (i.e., scripts) has commands which cost much less than the process movement cost (i.e., local execution of *ls* command costs around 0.8*sec*), the algorithm achieved on the average system-wide gain of 17% at bias value of 50% over the nonload balancing case. Also, we point out that the CPU utilization is almost evenly utilized. The difference in response time between Phase-1 results with 75% bias and Phase-2 results in Table 4 is due to the overhead of the control and information policy modules. When the SWITCH was set to "/bin:/lib:/tmp", the average system response time at bias of 50% was 8 minutes and 25 seconds, i.e., 35% improvement over the nonload balancing case.

For a heavy balanced load with the SWITCH set to NULL, whose results is shown in Table 5, these test results are very similar to the light balanced load case. The algorithm needed high bias of 75% to prohibit nonproductive processes movement. The gain achieved at bias 75% is negligible (less than 1%). At low bias of 25% processes movement was nonproductive and reduced 8.3% degradation in the average system response time.

We conclude this section with a note about selecting the bias value dynamically. The

Host	Light	Moderate	Heavy
1	2	4	4
2	1	1	3
3	1	2	5

TABLE 2. Workload Sets (script/host)

Bias	Phase-1 (SWITCH = NULL)									Phase-2		
	25%			50%			75%					
Host	1	2	3	1	2	3	1	2	3	1	2	3
Host rpt	9:06	5:02	7:51	8:58	4:51	4.55	7:53	4:52	4:54	8:05	4:47	4:47
% of movement	18.7	0	12.5	6.25	0	0	3.12	0	0	0	0	0
CPU util	82	90	85	96	88	86	95	89	89	93	88	88
system rpt	7:47			6:56			6:23			6:26		

TABLE 3. Light Balanced Workload Set (response time in minutes:seconds)

Bias	Phase-1 (SWITCH = NULL)									Phase-2		
	25%			50%			75%					
Host	1	2	3	1	2	3	1	2	3	1	2	3
Host rpt	15:41	11:29	11:38	12:31	8:52	9.08	17:40	4:52	8:14	17:28	4:47	8:05
% of movement	53.1	25	25	31.2	0	0	0	0	0	0	0	0
CPU util	87	98	77	97	95	94	98	89	95	94	88	93
system rpt	13:54			10:46			13:08			12:58		

TABLE 4. Moderate Unbalanced Workload Set (response time in minutes:seconds)

Bias	Phase-1 (SWITCH = NULL)									Phase-2		
	25%			50%			75%					
Host	1	2	3	1	2	3	1	2	3	1	2	3
Host rpt	21:01	19:14	19:34	18:56	13:31	23.36	17:29	13:10	22:16	17:28	13:08	22:25
% of movement	34.37	20.8	45	6.3	0	15	0	0	2	0	0	0
CPU util	84	98	94	98	97	93	91	98	92	94	93	94
system rpt	19:58			19:31			18:22			18:26		

TABLE 5. Heavy Balanced Workload Set (response time in minutes:seconds)

NRT algorithm was modified so that if the current ratio of the maximum host estimated load to the minimum one is greater than or equal to 4, the bias is selected to be 50%; otherwise, it is selected to be 75%. Different tests have been conducted using the automatic bias selection and resulted in a consistent response time improvement; however, we feel that more study is needed in this area.

8. RELATED WORK

Our work is related to processor sharing and load balancing in distributed systems. Processor sharing has been an issue of research in many distributed systems like Arachne [14], Accent [24], Cambridge Distributed System [21], DEMOS/MP [23]. However, it is difficult to draw parallels with our work because even though some of these systems support transparent remote process execution, none have reported system wide automatic scheduling. Other systems such as the *Worm* programs [26] have developed the notion of multi-segment worm programs. The warm mechanism is built into each program, necessitating modification in user programs. While the stress in the worm system is on failure recovery and making use of idle workstations, we stress on preserving the user applications without any changes, and load balancing across multi-user machines.

Locus provides preemptable remote execution [6]. The Locus system is geared toward providing a *single virtual machine*, meanwhile in a workstation environment we stress on autonomy. As a consequence, many design and implementations issues are very different.

A network-transparent remote execution facility has also been designed for the V-system [30] that allows a user of a workstations to offload programs onto idle workstations. The emphasis in this work is similar to the Worm programs system in making use of idle workstations rather than load balancing across a multi-user machines. Also, there are no reported results indicating the benefit of automatic scheduling to make a meaningful comparison.

Recently, a decentralized dynamic load balancing policy on the MOS system has been reported [3]. The scheduling policy used the average number of processes during the busy period as the measure of the machine's load. Our algorithm uses more sophisticated measure for representing the machine's load. Using the SWITCH capability in NEST, the average system response time improvement in our scheme without the process migration phase is much higher than those reported in MOS. However, more information is needed about the workload used in testing the load balancing algorithm to make more accurate comparison.

9. CONCLUDING REMARKS

In this paper we described the organization of a load balancing scheme implemented in the NEST distributed system. The scheme consists of two modules, the information policy module and the control policy module. The remote execution phase of the model has been implemented and many of its primitives will be also used for the process migration phase. The remote execution algorithm assigns processes to hosts in order to improve the overall system response time. The algorithm is adaptive in the sense that host's view of the network state is updated to reflect changes in the network workload, and the bias is selected dynamically depending on the current load unbalance in the network.

Some details of the load balancing scheme were described and initial results on its performance were given. The scheme was tested under a realistic environment. The main implication of these results is that, it is possible to reduce the response time by dynamically taking advantage of load differences in the system.

In the conduction of this work we have identified several issues which require further study. One issue is that it may prove to be useful to make assignment decisions for new processes after allowing them to execute locally at least an amount of time equal to the cost of moving the process. This would require studying the tradeoffs between the gain in performance by filtering inexpensive processes from being remotely executed and the extra cost of moving the process after it starts execution.

10. ACKNOWLEDGEMENTS

We gratefully acknowledge the effort and the comments of the referees which inspired a significant revision and improvement to the paper.

References

[1] R. Agrawal and A. K. Ezzat, "Processor Sharing in NEST: A Network of Computer Workstations," *1st*

International Conf. on Computer Workstations, Nov. 1985, pp. 198-208.

[2] R. Agrawal and A. K. Ezzat, *"Processor Sharing in a Network of Workstations,"* AT&T Workstation Symposium, Oct. 1985.

[3] A. B. Barak and A. Shiloh, "A Distributed Load-balancing Policy for a Multicomputer," *Software-Practice and Experience 15(9)*, (September 1985), pp. 901-913.

[4] S. H. Bokhari, "Dual Processor Scheduling with Dynamic Reassignment," *IEEE Trans. Software Eng. SE-5*, 4 (July 1979), pp. 341-349.

[5] R. M. Bryant and R. A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proc. 2nd Int. Conf. Distrib. Comput. Syst.*, Apr. 1981, pp. 314-323.

[6] D. A. Butterfield and G. J. Popek, "Network Tasking in The LOCUS Distributed UNIX System," *Proc. Summer USENIX Conference*, June 1984, 62-71.

[7] T. C. K. Chou and J. A. Abraham, "Load Balancing in Distributed Systems," *IEEE Trans. Software Eng. SE-8*, 4 (July 1982), pp. 401-412.

[8] Y. C. Chow and W. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous multiple Processor System," *IEEE Trans. Comput. C-28*, 5 (May 1979), pp. 354-361.

[9] W. W. Chu, "Optimal File Allocation in a Multiple Computing System," *IEEE Trans. Comput. C-18*, (Oct. 1969), pp. 885-889.

[10] G. Chuanshan, J. W. S. Liu and M. Railey, "Load Balancing Algorithms in Homogeneous Distributed Systems," *IEEE*, 1984, pp. 302-306.

[11] A. K. Ezzat, R. D. Bergeron and J. L. Pokoski, *"Adaptive Decentralized Control Job Scheduling in Distributed Computing Systems,"* Distributed Computing Journal. Submitted to.

[12] A. K. Ezzat and R. Agrawal, "Making Oneself Known in a Distributed World," *Proc. 1985 Int'l Conf. on Parallel Processing*, Aug. 1985, pp. 139-142.

[13] A. K. Ezzat, R. D. Bergeron and J. L. Pokoski, "Task Allocation Heuristics For Distributed Computing Systems," *Proc. 6th Int'l Conf. on Distributed Computing Systems*, May. 1986, pp. 337-346.

[14] R. Finkel, "The Arachne Kernel," Computer Sciences Tech. Rep. #380, Univ. Wisconsin, Madison, April 1980.

[15] S. L. Gaede, "A Scaling Technique For Comparing Interactive System Capacities," *Proc. CMG XIII Int'l Conf.*, San Diego, California, Dec. 1982, pp. 62-67.

[16] E. D. Jensen, "Distributed Control," in *Lecture Notes in Computer Science*. vol. 105, Springer-Verlag, New York, 1981, pp. 175-190.

[17] B. W. Kernighan and D. M. Ritchie, "The C Programming Language," Prentice-Hall, 1978.

[18] A. Kratzer and D. Hammerstrom, "A Study of Load Levelling," *IEEE COMPCON Fall 80*, Sept. 1980, pp. 647-654.

[19] M. N. Lionel and K. Hwang, "Optimal Load Balancing in a Multiple Processor System with Many Job Classes," *IEEE Trans. Software Eng. SE-11*, 5 (May 1985), pp. 491-496.

[20] M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proc. ACM Computer Network Performance Symposium*, Apr. 1982, pp. 47-55.

[21] R. M. Needham and A. J. Herbert, "The Cambridge Distributed Computing System" Addison-Wesley, 1982.

[22] L. M. Ni and K. Hwang, "Optimal Load Balancing Strategies for a Multiple Processor System," *Proc. 1981 Int'l Conf. on Parallel Processing*, Columbus, Ohio, Aug. 1981, pp. 352-357.

[23] M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proc. ACM-SIGOPS 9th Symp. on Operating Systems Principles*, Oct. 1983, 110-119.

[24] R. F. Rashid and G. G. Robertson, "Accent: A communication Oriented Network Operating System Kernel," *Proc. ACM-SIGOPS 8th Symp. on Operating Systems Principles*, Dec. 1981, 64-75.

[25] W. D. Roome and H. C. Torng, "Modeling and Design of Computer Networks with Distributed Computation Facilities," *IEEE Computer Networks: Trends and Applications*, May 1974, pp. 30-38.

[26] J. F. Shoch and J. A. Hupp, "The Worm Programs - Early Experience With a Distributed Computation," *Commun. ACM 25*, 3 (March 1982), 172-180.

[27] J. A. Stankovic, "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling," *IEEE Trans. Comput. C-34*, 2 (Feb. 1985), pp. 117-130.

[28] H. S. Stone, "Multiprocessor Scheduling with the aid of Network Flow Algorithms," *IEEE Trans. Software Eng. SE-3*, 1 (Jan. 1977), pp. 85-93.

[29] A. N. Tantawi and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," *ACM 32*, 2 (April 1985), pp. 445-465.

[30] M. M. Theimer, K. A. Lantz and D. R. Cheriton, "Preemptable remote Execution Facilities For The V-System" *Proc. ACM-SIGOPS 10th Symp. on Operating Systems Principles*, Dec. 1985, pp. 2-12.

[31] S. B. Wu and M. T. Liu, "Assignment of Tasks and Resources for Distributed Processing," *IEEE COMPCON Fall 80*, Sept. 1980, pp. 655-662.

Checkpointing and Rollback-Recovery for Distributed Systems[*]

Richard Koo[**]
Sam Toueg[†]

Department of Computer Science
Cornell University
Ithaca, New York 14853

ABSTRACT

We consider the problem of bringing a distributed system to a consistent state after transient failures. We address the two components of this problem by describing a distributed algorithm to create consistent checkpoints, as well as a rollback-recovery algorithm to recover the system to a consistent state. In contrast to previous algorithms, they tolerate failures that occur during their executions. Furthermore, when a process takes a checkpoint, a minimal number of additional processes are forced to take checkpoints. Similarly, when a process rolls back and restarts after a failure, a minimal number of additional processes are forced to roll back with it. Our algorithms require each process to store at most two checkpoints in stable storage. This storage requirement is shown to be minimal under general assumptions.

1. Introduction

Checkpointing and rollback-recovery are well-known techniques that allow processes to make progress in spite of failures[12]. The failures under consideration are transient problems such as hardware errors and transaction aborts, i.e., those that are unlikely to recur when a process restarts. With this scheme, a process takes a checkpoint from time to time by saving its state on stable storage[9] When a failure occurs, the process rolls back to its most recent checkpoint, assumes the state saved in that checkpoint, and resumes execution.

We first identify consistency problems that arise in applying this technique to a distributed system. We then propose a checkpoint algorithm and a rollback-recovery algorithm to restart the system from a consistent state when failures occur. Our algorithms prevent the well-known "domino effect" as well as livelock problems associated with rollback-recovery. In contrast to previous algorithms, they are fault-tolerant and involve a minimal number of processes. With our approach, each process stores at most two checkpoints in stable storage. This storage requirement is shown to be minimal under general assumptions.

The paper is organized as follows: We discuss the notion of consistency in a distributed system in section 2, and describe our system model in section 3. In section 4 we identify the problems to be solved. Sections 5 and 6 contain the checkpoint and rollback-recovery algorithms respectively. The algorithms are extended for concurrent executions in section 7. In section 8 we consider optimizations. Sections 9 and 10 contain a discussion and our conclusion.

2. Consistent Global States in Distributed Systems

The notion of a consistent global state is central to reasoning about distributed systems. It was considered by Randell[11], Russell[13], and Presotto[10], and formalized by Chandy and Lamport[2]. In this section, we summarise their ideas.

In a distributed computation, an *event* can be a spontaneous state transition by a process, or the sending or receipt of a message. Event a *directly happens before*[8] event b if and only if

(1) a and b are events in the same process, and a occurs before b; or

(2) a is the sending of a message m by a process and b is the receiving of m by another process.

[*] The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defence position, policy, or decision.

[**] This author was supported by the Defence Advanced Research Projects Agency (DoD) under ARPA order 53,8. Contract MDA903-85-C-0124, and by the National Science Foundation under grants DCR-8412582 and MCS 83-03135.

[†] This author was supported by the National Science Foundation under grant MCS 83-03135.

CH2345-7/86/0000/1150$01.00 © 1986 IEEE

The transitive closure of the *directly happens before* relation is the *happens before* relation. If event a happens before event b, b *happens after* a. (We abbreviate *happens before*, "before" and *happens after*, "after".)

A *local state* of a process p is defined by p's initial state and the sequence of events that occurred at p. A *global state* of a system is a set of local states, one from each process. The *state of the channels* corresponding to a global state s is the set of messages sent but not yet received in s. We can depict the occurrences of events over time with a time diagram, in which horizontal lines are time axes of processes, points are events, and arrows represent messages from the sending process to the receiving process. In this representation, a global state is a cut dividing the time diagram into two halves. The state of the channels comprises those arrows (messages) that cross the cut. Figure 1 is a time diagram for a system of four processes.

Informally, a cut (global state) in the time diagram is *consistent* if no arrow starts on the right hand side and ends on the left hand side of it. This notion of consistency fits the observation that a message cannot be received before it is sent in any temporal frame of reference. For example, the cuts c and c' in Figure 1 are consistent and inconsistent cuts, respectively. The state of the channels corresponding to cut c consists of one message from p to q, and another message from s to r. Readers are referred to the work of Chandy and Lamport[2] for a formal discussion of consistent global states.

3. System Model

The distributed system considered in this paper has the following characteristics:

(1) Processes do not share memory or clocks. They communicate via messages sent through reliable first-in-first-out (FIFO) channels with variable nonzero transmission time.

(2) Processes fail by stopping, and whenever a process fails, all other processes are informed of the failure in finite time. We assume that processes' failures never partition the communication network.

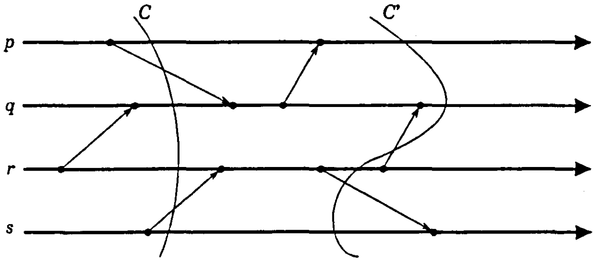


FIG. 1. Consistent and inconsistent cuts in a distributed system

We want to develop our algorithms under a weak set of assumptions. In particular, we do not assume that the underlying system is a database transaction system[4,6]. This special case admits simpler solutions: the mechanisms that ensure atomicity of transactions can hide inconsistencies introduced by the failure of a transaction. Furthermore, we do not assume that processes are deterministic: this simplifying assumption is made in previous results[6,16].

4. Identification of Problems

A checkpoint is a saved local state of a process. A set of checkpoints, one per process in the system, is consistent if the saved states form a consistent global state. For example, consider the system history in Figure 2. Process p takes a checkpoint at time X and then sends a message to q. After receiving this message, q takes a checkpoint at time Y. Subsequently, p fails and restarts from the checkpoint taken at X. The global state at p's restart is inconsistent because p's local state shows that no message has been sent to q, while q's local state shows that a message from p has been received. If p and q are processes supervising a customer's account at different banks, and the message transfers funds from p to q, the customer will have the funds at *both* banks when p restarts. This inconsistency persists even if q is forced to roll back and restart from its checkpoint taken at Y.

The problem of ensuring that the system recovers to a consistent state after transient failures has two components: checkpoint creation and rollback-recovery; we examine each one in turn.

4.1. Checkpoint Creation

There are two approaches to creating checkpoints. With the first approach, processes take checkpoints independently and save all checkpoints on stable storage. Upon a failure, processes must find a consistent set of checkpoints among the saved ones. The system is then rolled back to and restarted from this set of checkpoints[1,5,14,19].

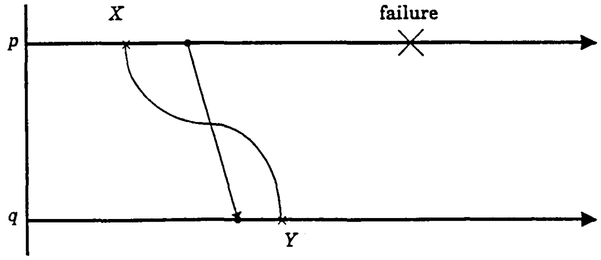


FIG 2. Inconsistent checkpoints.

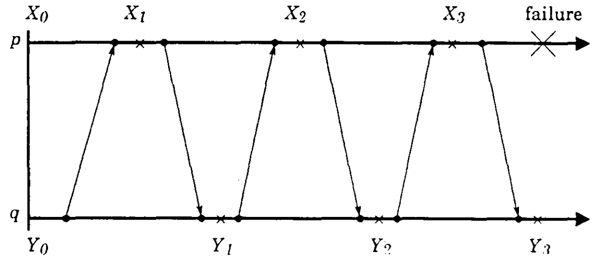


FIG. 3. "Domino effect" following a failure.

With the second approach, processes coordinate their checkpointing actions such that each process saves only its most recent checkpoint, and the set of checkpoints in the system is guaranteed to be consistent. When a failure occurs, the system restarts from these checkpoints[17].

The main disadvantage of the first approach is the "domino effect" as illustrated in Figure 3[10,11]. In this example, processes p and q have independently taken a sequence of checkpoints. The interleaving of messages and checkpoints leaves no consistent set of checkpoints for p and q, except the initial one at $\{X_0, Y_0\}$. Consequently, after p fails, both p and q must roll back to the beginning of the computation. For time-critical applications that require a guaranteed rate of progress, such as real time process control, this behavior results in unacceptable delays. An additional disadvantage of independent checkpoints is the large amount of stable storage required to save all checkpoints.

To avoid these disadvantages, we pursue the second approach. In contrast to Tamir[17], our method ensures that when a process takes a checkpoint, a minimal number of additional processes are forced to take checkpoints.

4.2. Rollback-Recovery

Rollback-recovery from a consistent set of checkpoints appears deceptively simple. The following scheme seems to work: Whenever a process rolls back to its checkpoint, it notifies all other processes to also roll back to their respective checkpoints. It then installs its checkpointed state and resumes execution. Unfortunately, this simple recovery method has a major flaw. In the absence of synchronization, processes cannot all recover (from their respective checkpoints) simultaneously. Recovering processes asynchronously can introduce livelocks as shown below.

Figure 4 illustrates the histories of two processes, p and q, up to p's failure. Process p fails before receiving the message n_1, rolls back to its checkpoint, and notifies q Then p recovers, sends m_2, and receives n_1. After p's recovery, p has no record of sending m_1, while q has a record of its receipt. Therefore the global state is inconsistent. To restore consistency, q must also roll back (to "forget" the receipt of m_1) and notify p. However, after q rolls back, q has no record of sending n_1 while p has a record of its receipt. Hence, the global state is inconsistent again, and upon notification of q's rollback, p must roll back a second time. After q recovers, q sends n_2 and receives m_2. Suppose p rolls back before receiving n_2 as shown in Figure 5. With the second rollback of p, the sending of m_2 is "forgotten". To restore consistency, q must roll back a second time. After p recovers it receives n_2, and upon notification of q's rollback, it must roll back a third time. It is now clear that p and q can be forced to roll back forever, even though no additional failures occur.

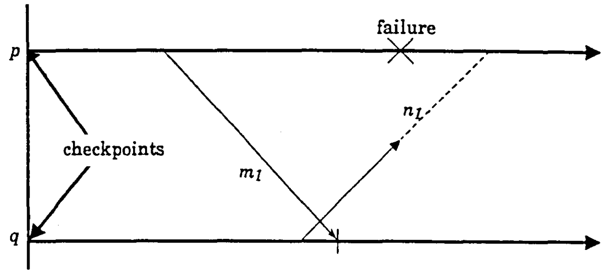


FIG. 4. Histories of p and q up to p's failure.

Our rollback-recovery algorithm solves this livelock problem. It tolerates failures that occur during its execution, and forces a minimal number of processes to roll back after a failure. However, in Tamir[17], a single failure forces the system to roll back as a whole. Furthermore, the system crashes (and does not recover) if a failure occurs while it is rolling back.

5. Checkpoint Creation

5.1. Naive Algorithms

From Figure 2 it is obvious that if every process takes a checkpoint after every sending of a message, and these two actions are done atomically, the set of the most recent checkpoints is always consistent. But creating a checkpoint after every send is expensive. We may naively reduce the cost of the above method with a strategy such

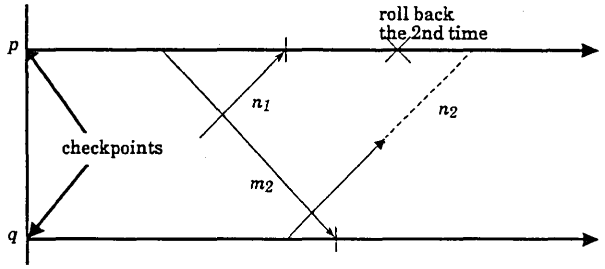


FIG. 5. History of p and q up to p's 2nd rollback.

as "every process takes a checkpoint after every k sends, $k > 1$" or "every process takes a checkpoint on the hour". However, the former can be shown to suffer domino effects by a construction similar to the one in Figure 3, while the latter is meaningless for a system that lacks perfectly synchronized clocks.

5.2. Classes of Checkpoints

Our algorithm saves two kinds of checkpoints on stable storage: permanent and tentative. A permanent checkpoint cannot be undone. It guarantees that the computation needed to reach the checkpointed state will not be repeated. A tentative checkpoint, however, can be undone or changed to be a permanent checkpoint. When the context is clear, we call permanent checkpoints "checkpoints".

Consider a system with a consistent set of permanent checkpoints. A checkpoint algorithm is *resilient* to failures if the set of permanent checkpoints is still consistent after the algorithm terminates, even if some processes fail during its execution. To exclude the impractical "naive" algorithm (in last section) from our consideration, henceforth, we consider only those systems where processes either cannot afford to take a checkpoint after every send, or cannot combine the sending of a message and the taking of a checkpoint into one atomic operation. The following theorem shows that checkpoint algorithms for these systems must store at least two checkpoints in stable storage to be resilient to failures. (The proofs of all lemmas and theorems in this paper can be found in Koo and Toueg[7].)

Theorem 1: No resilient checkpoint algorithms that take only permanent checkpoints exist. □

Theorem 1 shows that in those systems we consider, any resilient checkpoint algorithm must store at least two checkpoints on stable storage.

5.3. Our Checkpoint Algorithm

We assume the algorithm is invoked by a single process that wants to take a permanent checkpoint. We also assume that no failures occur in the system. In section 5.3.4 we extend the algorithm to handle failures, and in section 7 we describe concurrent invocations of this algorithm.

5.3.1. Motivation The algorithm is patterned on two-phase-commit protocols. In the first phase, the initiator q takes a tentative checkpoint and requests all processes to take tentative checkpoints. If q learns that all processes have taken tentative checkpoints, q decides all tentative checkpoints should be made permanent; otherwise, q decides tentative checkpoints should be discarded. In the second phase, q's decision is propagated and carried out by all processes. Since all or none of the processes take permanent checkpoints, the most recent set of checkpoints is always consistent.

However, our goal is to force a minimal number of processes to take checkpoints. The above algorithm is modified as follows: a process p takes a tentative checkpoint after it receives a request from q *only if* q's tentative checkpoint records the receipt of a message from p, and p's latest permanent checkpoint does not record the sending of that message. Process p determines whether this condition is true using the label appended to q's request. This labeling scheme is described below.

Messages that are not sent by the checkpoint or rollback-recovery algorithms are *system* messages. Every system message m contains a label $m.l$. Each process appends outgoing system messages with monotonically increasing labels. We define $\perp$ and $\top$ to be the smallest and largest labels, respectively. For any processes q and p, let m be the last message that q received from p after q took its last permanent or tentative checkpoint. Define:

$$\mathit{last\_rmsg}_q(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \perp & \text{otherwise} \end{cases} .$$

Also, let m be the first message that q sent to process p after q took its last permanent or tentative checkpoint. Define:

$$\mathit{first\_smsg}_q(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \perp & \text{otherwise} \end{cases} .$$

When q requests p to take a tentative checkpoint, it appends $\mathit{last\_rmsg}_q(p)$ to its request; p takes the checkpoint only if $\mathit{last\_rmsg}_q(p) \geq \mathit{first\_smsg}_p(q) > \perp$.

5.3.2. Informal Description Process p is a *ckpt_cohort* of q if q has taken a tentative checkpoint, and $\mathit{last\_rmsg}_q(p) > \perp$ before the tentative checkpoint is taken. The set of ckpt_cohorts of q is denoted $\mathit{ckpt\_cohort}_q$. Every process p keeps a variable $\mathit{willing\_to\_ckpt}_p$ to denote its willingness to take checkpoints. Whenever p cannot take a checkpoint (for any reason), $\mathit{willing\_to\_ckpt}_p$ is "no". The initiator q starts the checkpoint algorithm by making a tentative checkpoint and sending a request "take a tentative checkpoint and $\mathit{last\_rmsg}_q(p)$" to all $p \in \mathit{ckpt\_cohort}_q$. A process p *inherits* this request if $\mathit{willing\_to\_ckpt}_p$ is "yes" and $\mathit{last\_rmsg}_q(p) \geq \mathit{first\_smsg}_p(q) > \perp$. If p inherits a request, it takes a tentative checkpoint and sends "take a tentative checkpoint and $\mathit{last\_rmsg}_p(r)$" requests to all $r \in \mathit{ckpt\_cohort}_p$. If p receives but does not inherit a request from q, p replies $\mathit{willing\_to\_ckpt}_p$ to q.

After p sends out its requests, it waits for replies that can be either "yes" or "no", indicating a ckpt_cohort's acceptance or rejection of p's request. If any reply is "no", *willing_to_ckpt*$_p$ becomes "no"; otherwise *willing_to_ckpt*$_p$ is unchanged. Process p then sends *willing_to_ckpt*$_p$ to the process whose request p has inherited. From the time p takes a tentative checkpoint to the time it receives the decision from the initiator, p does not send any system messages.

If all the replies from its ckpt_cohorts arrive and are all "yes", the initiator decides to make all tentative checkpoints permanent. Otherwise the decision is to undo all tentative checkpoints. This decision is propagated in the same fashion as the request "take a tentative checkpoint" is delivered. A process discards its previous checkpoints after it takes a new permanent checkpoint.

The algorithm is presented in Figure 6. For simplicity, we create a fictitious process called *daemon* to assume the initiation and decision tasks of the initiator. In practice, daemon is a part of the initiator process.

5.3.3. Proofs of Correctness

We consider a single invocation of the algorithm, and we assume no process fails in the system.

Lemma 1: Every process inherits at most one request to take a tentative checkpoint. □

Lemma 2: Every process terminates its execution of Algorithm C1. □

The next lemma shows that C1 takes a consistent set of checkpoints.

Lemma 3: If the set of checkpoints in the system is consistent before the execution of Algorithm C1, the set of checkpoints in the system is consistent after the termination of C1. □

We now show the number of processes that take new permanent checkpoints during the execution of Algorithm C1 is minimal. Let $P=\{p_0, p_1, \cdots, p_k\}$ be the set of processes that take new permanent checkpoints in C1, where p_0 is the initiator of C1. Let $C(P)=\{c(p_0), c(p_1), \cdots, c(p_k)\}$ be the new permanent checkpoints taken by processes in P. Define an alternate set of checkpoints: $C'(P)=\{c'(p_0), c'(p_1), \cdots, c'(p_k)\}$ where $c'(p_0) = c(p_0)$ and for $1 \leq i \leq k$, $c'(p_i)$ is either $c(p_i)$ or the checkpoint p_i had before executing C1.

Theorem 2: $C'(P)$ is consistent if and only if $C'(P)=C(P)$.

```
Daemon process:

send(initiator, "take a tentative checkpoint and ⊤");
await(initiator, willing_to_ckpt_initiator);**
if willing_to_ckpt_initiator = "yes" then
  send(initiator, "make tentative checkpoint permanent")
else
  send(initiator, "undo tentative checkpoint")
fi.

All processes p:

INITIAL STATE:
  first_smsg_p(daemon) = ⊤;
  willing_to_ckpt_p = { "yes" if p is willing to checkpoint
                      { "no"  otherwise                      ;

UPON RECEIPT OF "take a tentative checkpoint and
  last_rmsg_q(p)" from q DO
  if willing_to_ckpt_p and last_rmsg_q(p) ≥ first_smsg_p(q) > ⊥
    then take a tentative checkpoint;
    for all r ∈ ckpt_cohort_p send(r, "take a tentative
      checkpoint and last_rmsg_p(r)");
    for all r ∈ ckpt_cohort_p await(r, willing_to_ckpt_r);
    if ∃ r ∈ ckpt_cohort_p, willing_to_ckpt_r = "no"
      then willing_to_ckpt_p ← "no" fi;
  fi;
  send(q, willing_to_ckpt_p);
OD.

UPON FIRST RECEIPT OF m = "undo tentative checkpoint" or
  m = "make tentative checkpoint permanent" DO
  if m = "make tentative checkpoint permanent" then
    make tentative checkpoint permanent;
  else
    undo tentative checkpoint;
  fi;
  for all r ∈ ckpt_cohort_p, send(r, m);
OD.
```

FIG. 6. Algorithm C1: the Checkpoint Algorithm

Theorem 2 shows that if p_0 takes a checkpoint, then all processes in P must take a checkpoint to ensure consistency.

*await does not prevent a process from receiving messages

5.3.4. Coping with Failures

We now extend Algorithm C1 to handle processes' failures. We first consider the effects of failures on nonfaulty processes. When failures occur, a nonfaulty process may not receive some of the following messages:

(1) "yes" or "no" from ckpt_cohorts,

(2) "make tentative checkpoint permanent" or "undo tentative checkpoint" from the initiator.

Suppose that process p fails before replying "yes" or "no" to process q's request. By the assumption of section 3, q will know of p's failure. After q knows that p has failed, it sets $willing\_to\_ckpt_q$ to "no" and stops waiting for p's reply. Therefore, to take care of a missing "yes" or "no", it suffices to change the statement in C1 from

```
if $\exists\ r \in ckpt\_cohort_p,\ willing\_to\_ckpt_r$ = "no"
  then $willing\_to\_ckpt_p \leftarrow$ "no" fi
```

to

```
if $\exists\ r \in ckpt\_cohort_p,\ willing\_to\_ckpt_r$ = "no" or $r$ has failed
  then $willing\_to\_ckpt_p \leftarrow$ "no" fi.
```

Suppose that process p does not receive the decision regarding its tentative checkpoint. If p undoes its tentative checkpoint or makes it permanent, it risks contradicting the initiator. The two-phase structure of C1 requires p to block until it discovers the initiator's decision[15]. We will discuss ways to prevent blocking in section 8.

We now consider the recovery of faulty processes. When a process restarts after a failure, its latest checkpoint on stable storage may be tentative or permanent. If this checkpoint is tentative, the restarting process must decide whether to discard it or to make it permanent. The decision is made as follows:

Suppose that the restarting process is the initiator. The initiator knows that every process that has taken a tentative checkpoint is still blocked waiting for its decision. Hence, it is safe for the initiator to decide to undo all tentative checkpoints and send this decision to its ckpt_cohorts. If the restarting process is not the initiator, it must discover the initiator's decision regarding tentative checkpoints. It may contact either the initiator or those processes of which it is a ckpt_cohort; it follows the decision accordingly to terminate C1.

The restarting process is now left with one permanent checkpoint on stable storage. It can recover from this checkpoint by invoking the rollback-recovery algorithm of section 6.

Let C2 be the Algorithm C1 as modified above. C2 terminates if all processes that fail during the execution of C2 recover. At termination, the set of checkpoints in the system is consistent, and the number of processes that took new permanent checkpoints is minimal. The proofs for these properties are similar to those of C1 and they are omitted.

6. Rollback-Recovery

We assume that the algorithm is invoked by a single process that wants to roll back and recover (henceforth denoted *restart*). We also assume that the checkpoint algorithm and the rollback-recovery algorithm are not invoked concurrently. Concurrent invocations of these algorithms are described in section 7.

6.1. Motivation

The rollback-recovery algorithm is patterned on two-phase-commit protocols. In the first phase, the initiator q requests all processes to restart from their checkpoints. Process q decides to restart all the processes if and only if they are all willing to restart. In the second phase, q's decision is propagated and carried out by all processes. We will prove that the two-phase structure of this algorithm prevents livelock as discussed in section 4.2. Since all processes follow the initiator's decision, the global state is consistent when the rollback-recovery algorithm terminates.

However, our goal is to force a minimal number of processes to roll back. If a process p rolls back to a state saved before an event e occurred, we say that e is *undone* by p. The above algorithm is modified as follows: the rollback of a process q forces another process p to roll back *only if* q's rollback undoes the sending of a message to p. Process p determines if it must restart using the label appended to q's "prepare to roll back" request. This label is described below.

For any processes q and p, let m be the last message that q sent to p before q took its latest permanent checkpoint. Define

$$last\_smsg_q(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \top & \text{otherwise} \end{cases}.$$

When q requests p to restart, it appends $last\_smsg_q(p)$ to its request. Process p restarts from its permanent checkpoint only if $last\_rmsg_p(q) > last\_smsg_q(p)$.

6.2. Informal Description

Process p is a $roll-cohort$ of q if q can send messages to it. The set of roll-cohorts of q is $roll-cohort_q$. Every process p keeps a variable $willing\_to\_roll_p$ to denote its willingness to roll back. Whenever p cannot roll back (for any reason), $willing\_to\_roll_p$ is "no". The initiator q starts the rollback-recovery algorithm by sending a request "prepare to roll back and $last\_smsg_q(p)$" to all $p \in roll-cohort_q$. A process p *inherits* this request if $willing\_to\_roll_p$ is "yes", $last\_rmsg_p(q) > last\_smsg_q(p)$, and p has not already inherited another request to roll back. After p inherits the request, it sends "prepare to roll back and $last\_smsg_p(r)$" to all $r \in roll-cohort_p$; otherwise, it replies $willing\_to\_roll_p$ to q.

After p sends out its requests, it waits for replies from each process in $roll-cohort_p$. The reply can be an explicit "yes" or "no" message, or an implicit "no" when p discovers that r has failed. If any reply is "no", $willing\_to\_roll_p$ becomes "no", otherwise $willing\_to\_roll_p$ is unchanged. Process p then sends $willing\_to\_roll_p$ to the process whose request p inherits. From the time p inherits the rollback request to the time it receives the decision from the initiator, p does not send any system messages.

If all the replies from its roll-cohorts arrive and are all "yes", the initiator decides the rollbacks will proceed, otherwise it decides no process will roll back. This decision is propagated to all processes in the same fashion as the request "prepare to roll back" is delivered. If failures prevent the decision from reaching a process p, p must block until it discovers the initiator's decision. We discuss nonblocking algorithms in section 8.

The rollback-recovery algorithm is presented in Figure 7. Like the presentation of Algorithm C1, we introduce a fictitious process called *daemon* to perform functions that are unique to the initiator of the algorithm.

6.3. Proofs of Correctness

We consider a single invocation of the rollback-recovery algorithm. The variable $ready\_to\_roll_p$ ensures that a process p inherits at most one request to roll back. As a result, the variable also ensures that a process rolls back at most once. To prove the termination of Algorithm R, it suffices to show that Algorithm R is free of deadlocks.

Lemma 4: Algorithm R is deadlock free. □

We show next that the global state of the system is consistent after the termination of R

```
Daemon process:

send(initiator, "prepare to roll back and $\perp$");
await(initiator, $willing\_to\_roll_{initiator}$);
if $willing\_to\_roll_{initiator}$ = "yes" then
  send(initiator, "roll back")
else
  send(initiator, "do not roll back")
fi.

All processes p:

INITIAL STATE:
  $ready\_to\_roll_p$ = true;
  $last\_rmsg_p(daemon) = \top$;
  $willing\_to\_roll_p = \begin{cases} \text{"yes"} & \text{if } p \text{ is willing to roll back} \\ \text{"no"} & \text{otherwise} \end{cases}$ ;

UPON RECEIPT OF "prepare to roll back and
  $last\_smsg_q(p)$" from q DO
  if $willing\_to\_roll_p$ and $last\_rmsg_p(q) > last\_smsg_q(p)$
    and $ready\_to\_roll_p$ then
    $ready\_to\_roll_p \leftarrow$ false;
    for all $r \in roll-cohort_p$
      send(r, "prepare to roll back and $last\_smsg_p(r)$");
    for all $r \in roll-cohort_p$ await(r, $willing\_to\_roll_r$);
    if $\exists\ r \in roll\_cohort_p$, $willing\_to\_roll_r$ = "no"
      or r has failed then $willing\_to\_roll_p \leftarrow$ "no" fi;
  fi;
  send(q, $willing\_to\_roll_p$);
OD.

UPON RECEIPT OF m ="roll back" or
  m ="do not roll back" and $ready\_to\_roll_p$ = false DO
  if m = "roll back" then
    restart from p's permanent checkpoint;
  else
    resume execution;
  fi;
  for all $r \in roll-cohort_p$, send(r, m);
OD.
```

FIG. 7. Algorithm R: the Rollback Algorithm

Lemma 5: If the system is consistent before the execution of Algorithm R, the system is consistent after the termination of Algorithm R. □

Lemma 5 ensures that a single execution of Algorithm R brings the system to a consistent state after a failure; since processes roll back at most once in any execution of R, the rollback algorithm prevents livelocks. Thus, Algorithm R prevents livelocks.

Many existing rollback algorithms exhibit the following undesirable property. If the initiator rolls back, it forces an additional set of processes P to roll back with it, even though the system will be consistent if some of the processes in P omit to roll back. For instance, all processes are required to roll back every time any process wants to roll back[17]. However, in some cases, the initiator could roll back alone and the system would still be consistent. With our algorithm, the number of processes that are forced to roll back with the initiator is minimal.

Theorem 3: Let E be an execution of R in which the initiator, p_0, and an additional set of processes P roll back. Consider an execution E', identical to E except that a non-empty subset of processes in P omit to roll back upon receipt of the "roll back" decision. The execution E' leaves the system in an inconsistent state. □

7. Interference

In this section, we consider concurrent invocations of the checkpoint and rollback-recovery algorithms. An execution of these algorithms by process p is *interfered* with if any of the following events occur:

(1) Process p receives a rollback request from another process q while executing the checkpoint algorithm.

(2) Process p receives a checkpoint request from q while executing the rollback-recovery algorithm.

(3) Process p, while executing the checkpoint algorithm for initiator i, receives a checkpoint request from q, but q's request originates from a different initiator than i.

(4) Process p, while executing the rollback-recovery algorithm for initiator i, receives a rollback request from q, but q's request originates from a different initiator than i.

One single rule handles the four cases of interference: once p starts the execution of a checkpoint [rollback] algorithm, p is unwilling to take a tentative checkpoint [roll back] for another initiator, or to roll back [take a tentative checkpoint]. As a result, in all four cases, p replies "no" to q. We can show this rule is sufficient to guarantee that all previous lemmas and theorems hold despite concurrent invocations of the algorithms. This rule can, however, be modified to permit more concurrency in the system. The modification is that in case (1), instead of sending "no" to q, p can begin executing the rollback-recovery algorithm after it finishes the checkpoint algorithm. We cannot allow a similar modification in case (2) lest deadlocks may occur.

8. Optimization

When the initiator of the checkpoint or of the rollback-recovery algorithm fails before propagating its decision to its cohorts, it is desirable for processes not to block for its recovery. To prevent processes from blocking, we can modify our algorithms by replacing the underlying two-phase commit protocol with a nonblocking three-phase commit protocol[15]. However, nonblocking protocols are inherently more expensive than blocking ones[3].

We next address the following problem: after a ckpt_cohort q of a process p fails, p cannot take a permanent checkpoint until q restarts (p cannot know if the latest checkpoint of q records the sendings of all messages it received from q). To avoid waiting for q's restart, p can remove q from $ckpt\_cohort_p$ by restarting from its checkpoint (using the rollback-recovery algorithm). After its restart, process p can take new checkpoints.

9. Message Loss

Rollback-recovery can cause message loss as illustrated in Figure 8. When p is rolled back to X following a failure at F, the global state is consistent, but the message m from q is lost. It is lost because the state of the channels corresponding to the global state $\{X, Y\}$ contains m.

One method to circumvent message loss is to have that processes use transmission protocols that transform lossy channels to virtual error-free channels, e.g., sliding window protocols[18]. Another method is to ensure that the state of the channels corresponding to the most recent set

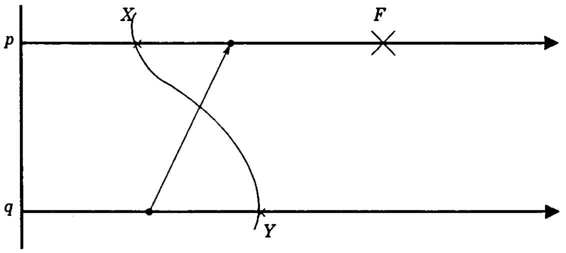


FIG. 8. Message loss following p's rollback to X.

of checkpoints contains no messages. We can modify the checkpoint and rollback-recovery algorithms to implement this latter method, but such modification increases the number of processes that are forced to take checkpoints and roll back.

10. Conclusion

We have presented a checkpoint algorithm and a rollback-recovery algorithm to solve the problem of bringing a distributed system to a consistent state after transient failures. In contrast to previous algorithms, they tolerate failures that occur during their executions. Furthermore, when a process takes a checkpoint, a minimal number of additional processes are forced to take checkpoints. Similarly, a minimal number of additional processes are forced to restart when a process restarts after a failure. We also show that the stable storage requirement of our algorithms is minimal.

Acknowledgements

We would like to thank Amr El Abbadi, Ken Birman, Rance Cleaveland, and Jennifer Widom for commenting on earlier drafts of this paper.

Bibliography

[1] T. Anderson, P. A. Lee and S. K. Shrivastava, System fault tolerance, in *Computing System Reliability*, T. Anderson, B. Randell (eds.) Cambridge University Press, Cambridge, 1979, pp. 153-210.

[2] K. M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63-75, February 1985.

[3] C. Dwork and D. Skeen, The inherent cost of nonblocking commitment, *Proc. ACM Symposium on Principles of Database Systems*, March 1983.

[4] M. Fischer, N. Griffeth, and N. Lynch, Global states of a distributed system, *IEEE Transaction on Software Engineering*, May 1982, pp. 198-202

[5] V. Hadzilacos, An algorithm for minimizing rollback cost, *Proc. ACM Symposium on Principles of Database Systems*, March 1982.

[6] T. Joseph and K. Birman, Low cost management of replicated data in fault-tolerant distributed systems, *ACM Transactions on Computer Systems*, February 1986, pp. 54-70.

[7] R. Koo and S. Toueg, Checkpointing and Rollback-Recovery for Distributed Systems, To appear in a special issue of *IEEE-TSE*.

[8] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM*, vol. 21, no. 7, July 1978, pp. 558-565.

[9] B. Lampson and H. Sturgis, Crash recovery in a distributed storage system, *Xerox PARC Tech. Rep.*, Xerox Palo Alto Research Center, April 1979.

[10] D. L. Presotto, Publishing: A reliable broadcast communication mechanism, *Tech. Rep. UCB/CSD 83-165, Computer Science Division, University of California, Berkeley*, December 1983.

[11] B. Randell, System structure for software fault tolerance, *IEEE Transactions On Software Engineering*, vol. SE-1, no.2, June 1975, pp. 226-232.

[12] B. Randell, P.A. Lee, and P.C. Treleaven, Reliability issues in computing system design, *ACM Computing Surveys*, vol. 10, no. 2, June 1978, pp. 123-166.

[13] D. L. Russell, Process backup in producer-consumer systems, *Proc. ACM Symposium on Operating Systems Principles*, November, 1977.

[14] D. L. Russell, State restoration in systems of communicating processes, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, March 1980, pp. 183-194.

[15] D. M. Skeen, Crash recovery in a distributed database system, *Ph.D. dissertation, Computer Science Division, University of California, Berkeley*, 1982.

[16] R. Strom and S. Yemini, Optimistic recovery in distributed systems, *Transactions on Computer Systems*, August 1985, pp. 204-226.

[17] Y. Tamir and C. H. Sequin, Error recovery in multicomputers using global checkpoints, *Proc. of 13th International Conference on Parallel Processing*, August 1984.

[18] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, New Jersey, 1981, pp. 148-164.

[19] W. G. Wood, A decentralized recovery control protocol, *Proc. of the 11th Annual International Symposium on Fault-Tolerant Computing*, June 1981.

The Gutenberg Operating System Kernel [1]

Panayiotis Chrysanthis, Krithi Ramamritham, David Stemple, and Stephen Vinter [2]

Department of Computer and Information Science
University of Massachusetts Amherst MA. 01003

ABSTRACT

The Gutenberg system is a port-based, object-oriented operating system kernel designed to facilitate the design and structuring of distributed systems. This is achieved by providing primitives for controlling process interconnections and thereby controlling access to shared resources. Only shared resources are viewed as protected objects. Processes communicate with each other and access protected objects through the use of ports. Each port is associated with an abstract data type operation and can be created by a process only if the process possesses the privilege to execute that operation. Capabilities to create ports for requesting operations are contained in the *capability directory* which is a kernel object. At any time, each process is associated with a single subdirectory of the capability directory designated as its *active directory*.

This paper describes the design philosophy and the structure of the Gutenberg kernel. First, it discusses the principles and motivations behind the Gutenberg design. Then, it presents the structure, contents and operations of kernel objects and discusses their use in structuring access to protected user-defined objects. Also discussed are the salient aspects of the distributed Gutenberg kernel. A brief comparison with other related systems is given.

1 INTRODUCTION

Modern distributed operating systems must be designed to facilitate structuring distributed computations in an understandable and reliable manner that is suitable for validation. Experience in programming languages and in controlling complexity of large software systems has shown that the strongly typed object paradigm and its associated information hiding can be used to produce manageable and understandable systems. In large distributed systems, lack of a corresponding clean, well-structured interprocess communication paradigm leads to more complexity than is required by the nature of the distribution alone. This points to the need for an efficient mechanism which structures all interprocess communications along the abstract data type lines.

This paper deals with the design of such a mechanism, the distributed Gutenberg kernel [11,12,7,8]. The Gutenberg Operating System Kernel, currently being developed at the University of Massachusetts, takes a unique approach to interprocess communication, provides multiple programming language support, and attempts to minimize overheads. The following three salient features of Gutenberg provide for the controlled establishment of communication connections which serves the goal of understandability and verifiability, and should contribute to its better performance compared to previous communication schemes:

- the adoption of *port-based communication*, whereby interprocess communication is solely by means of ports, queue-like objects which are managed by the kernel.
- the adoption of *non-uniform object-orientation*, whereby only interprocess communication, but not module interconnections within a process, are structured and controlled by means of capabilities; and
- the use of a *capability directory*, which expresses all potential process interconnections in the system and is a distributed persistent object, maintained and manipulated only by the kernel.

Principles underlying the Gutenberg Kernel. The Gutenberg system evolved from the following series of design decisions concerning the nature of communication and protection in the system.

Limit the responsibilities of the kernel. Operating systems that support the definition and protection of arbitrary objects traditionally have had performance problems because of the maintenance of protection domains and the overhead of dynamic access checks, e.g. Hydra [16] and iMAX [4]. We hoped to restrict the overheads of the kernel by taking a *non-uniform object orientation,* in which only resources shared by different processes need to be structured as objects at the operating system level. We believe that conventional mechanisms for protecting unshared data (e.g., local variables) are adequate and desirable, from both a downward compatibility and an efficiency viewpoint. Conventional memory protection mechanisms (e.g. page tables) ensure that the local data of processes are not accessible to other processes. Gutenberg depends on programming languages to provide static type-checking for self-protection. Thus, although local data *are not* necessarily object-oriented (depending on the programming language), shared data *are* required to be object-oriented. An underlying assumption in Gutenberg is that the granularity of objects is medium to large, a size adequate to amortize the cost of the interprocess communication required to access the object.

Programming language independence. There should be no requirement that a specific programming language or that only

[1] This material is based upon work supported in part by the National Science Foundation under grants DCR-8403097, and DCR-8500332.

[2] Stephen T. Vinter's current address is: BBN Laboratories, Cambridge, Mass.

CH2345-7/86/0000/1159$01.00©1986 IEEE

object-oriented languages run under the Gutenberg system kernel. In fact, non-object-oriented languages can achieve an object oriented view of other processes through the use of kernel primitives [11]. The system is designed to support applications implemented in any programming language.

Resources are directly manipulated only by their managers. Managers are processes that synchronize the operations on an object (i.e., a shared resource). A process managing an object is the only subject able to directly manipulate the object. The process performs the operations, in part, by invoking kernel primitives for manipulating kernel objects.

Object sharing is via interprocess communication. Processes do not share address spaces. They can interact only through interprocess communication using explicit message passing. An operation can be performed on the object only as a result of a request from another process via interprocess communication. Processes are provided with communication primitives allowing both synchronous and asynchronous communication.

Interprocess communication connections are established using functional addressing [12]. Processes use communication channels called *ports* to request operations on objects. A port can be used to request an operation only if it was created for that purpose. Ports are created by indicating the function of the port (viz., to request an operation), not by identifying a particular process.

The kernel controls access to shared objects by controlling interprocess communication. An operation may be requested only by transmitting the operation request over a port to the object's manager. By limiting the use of ports and constraining their use to request a specific operation on a specific object, access to the object is controlled.

Details of the implementation of the communication mechanism are hidden from the processes using it. Ports are themselves objects with a small set of operations defined on them. They are managed by the kernel. The representation of ports and details of message transmission and reception are hidden from communicating processes.

Privileges persist in a single kernel-managed structure. Gutenberg recognizes that privileges need to persist in the system independent of the execution of processes. Rather than allowing privileges to be placed on secondary storage in user objects (hidden from the view of the kernel), privileges that are not dependent on the existence of a process are stored in a kernel managed structure called the *capability directory.* This directory is shared by the processes in the system. Being physically distributed, it can be designed to ensure availability and reliable access despite communication and node failures. However, since it is logically unified, i.e., appears as a single globally addressable entity, the physical distribution will be transparent to its accessors. Transient capabilities of a process, i.e., capabilities that exist only as long as the process is active, are kept in the process' *c-list.*

The above design principles made it possible to use Gutenberg as the basis for a distributed operating system for the following reasons. The use of resource managers is a common approach to structuring software in distributed systems. Second, the logical separation of process address spaces in Gutenberg corresponds to the physical realization of processes in a distributed system. Third, the close association of communication and protection contributes to decentralized access authorization.

The rest of the paper is structured as follows: Section 2 introduces the structure and the contents of the kernel objects. User-defined objects, type creation, and manager instantiation are the subject of section 3. Issues related to the dynamic control of interprocess communication are also discussed in section 3. The distributed kernel is examined in section 4. Section 5 contains a brief comparison of Gutenberg with related systems. We conclude with a section summarizing our approach and future work.

2 GUTENBERG KERNEL OBJECTS

The Gutenberg kernel itself is structured as a set of cooperative abstract data type managers. Furthermore, the kernel is viewed by the processes as an abstract data type manager of kernel objects with the kernel primitives as the corresponding abstract data type operations. There are four types of kernel objects: *processes*, *ports*, *capability directory* and *transient capabilities.*

2.1 Processes

A process is an independently schedulable unit of computation with the ability to communicate with other processes. Each process is represented by a unique *process control block*, abbreviated *PCB*, which resides within the kernel address space.

Processes can communicate only through explicit message exchanges over communication channels called *ports.* As a result, processes do not *share address spaces*, eliminating the need for synchronization in memory access and generalizing the process interactions in a distributed system.

2.2 Ports

A port is a kernel object that processes manipulate by invoking kernel primitives. It is a communication channel between a pair of processes in one-to-one topology connecting just one pair of processes at a time. Typically, one process has the privilege to place messages on the port, which behaves as a queue of messages awaiting delivery. The other has the privilege to remove messages. This communication can be either synchronous or asynchronous.

The basic interprocess communication of the Gutenberg system is based on the client/server model, in which the creator of a port, called the *client*, communicates with the port server, the manager of some shared object, for the purpose of requesting an operation on the object. A port is established with *functional addressing*: A client creates a port by naming the service it would like to request using the port rather than by identifying the server process. As a result, the server process does not have to be in existence prior to the creation of the port. The advantage of this strategy is that it allows the dynamic creation of server processes. In fact, in Gutenberg, process creation and destruction are byproducts of port operations. There are no primitives for process creation and destruction: *processes are hidden from the programmers, the lowest level of abstraction being the level of operations on the ports.*

Therefore, the only way a process can request an operation on a shared object is to create a port and execute a kernel primitive on that port. The possible kernel primitives on ports that a client is entitled to, include SEND, RECEIVE, and SEND-RECEIVE (to receive the result of an operation based on the parameters sent).

In order to restrict the use of ports to the functionality for which they have been created, a port is typed as either a Send, Receive, or Send-Receive port. This typing specifies the directionality of the port and the kernel primitives used by the client to transmit messages through it. Consequently, it specifies the kernel primitives that the server of the port may use. Figure 1 shows the port primitives used by clients and servers for each port type. Send and Receive ports are unidirectional. Send-Receive ports are bidirectional, allowing the port's client to send a message and receive a response from the port's server.

Port typing also determines the format of messages that may be placed in the port and the *object operation* associated with this, which identifies the operation that will be requested via the port. Each port is represented by a unique *channel control block*, abbreviated *CCB*, which resides within the kernel address space. CCB contains the port type along with information about the

Port Type	Client Primitives	Server Primitives
S	SEND REVOKE	RECEIVE REFUSE EXAMINE
R	RECEIVE EXAMINE	SEND REFUSE
SR	SEND-RECEIVE REVOKE EXAMINE	GETDETAILS SEND REFUSE EXAMINE

Figure 1: Port primitives used by clients and servers for each port type

status of the client and server processes and the owner of the port. Ownership represents the privilege to destroy the port. The creator of a port becomes the initial owner of the port. As part of the sharing mechanism supported by the Gutenberg system, a process may transfer part of its privileges, including port privileges, to another, over ports.

Here is a short summary of the functionality of port primitives.

CREATE-PORT (only a client primitive) creates a port of a specific type. The type is specified via a parameter to the call.

DESTROY-PORT (only a client primitive) destroys a port. The caller must be the owner of the port. The port-id is specified via a parameter to the call.

SEND puts a message on a port. The system has two kinds of SEND primitives: acknowledge-SEND and no-acknowledge-SEND. If the SEND is an acknowledge-SEND, the sending process is informed when its correspondent over the port receives the message. The sender can choose to block until the receipt of the acknowledgement.

RECEIVE requests the next message from the port. The caller elects via a parameter to the call, to either block, if there is no message on the port, or execute concurrently with the servicing of the request.

SEND-RECEIVE (only a client primitive) puts information, termed *request details*, on a port for the server to use in satisfying the request. When the server responds to the request by executing a SEND, the server's reply is returned to the client as in RECEIVE. The caller may block until the server replies, or execute concurrently with the servicing of the request.

ACCEPT-REQUEST (only a server primitive) is used to obtain access to newly created ports and to query a set of existing ports to see if new messages have arrived. The caller may block until the kernel replies, or execute concurrently with the servicing of the request.

GETDETAILS gets request details from a port. The caller (the port's server) may block if there is no pending SEND-RECEIVE, and thus no request details, on the port, or it may execute concurrently with the satisfaction of its request.

EXAMINE examines messages on the port without removing them.

REFUSE rejects a client's request for service as unsatisfiable and notifies the requester by setting a status.

REVOKE revokes privileges sent as part of request details by a SEND-RECEIVE or in a SEND message up to receipt of the message [3].

The choice of these primitives during system design was based on the desire to keep their number and complexity to a minimum while providing users a set of primitives for building systems of communicating processes in arbitrary topologies with reasonable ease. Thus, we have added to the basic SEND and RECEIVE primitives the bidirectional SEND-RECEIVE and its receiving reciprocal GETDETAILS in order to allow such functions as reading a record with a given key (the key being sent as request details) or a remote procedure call (the procedure's parameters being sent as request details) to be implemented by a single primitive. However, it should be noted that the asynchronous mode makes the SEND-RECEIVE operation more robust and flexible than a remote procedure call semantics.

2.3 Capability Directory

Gutenberg controls the creation and use of ports through the use of capabilities. All capabilities for accessing potentially sharable objects are maintained in a logically unified structure called the capability directory. Thus, the capability directory expresses all potential process interconnections in the system. This is similar to the UNIX file directory [10] which provides uniform treatment of files, devices and interprocess communication. The capability directory is a *stable* structure in that its existence does not depend on the existence of any process. It is also *shared* since more than one process may concurrently access the same segment of the directory. It should be noted that no portion of the directory is owned by any process at any time.

2.3.1 Capability Directory Nodes

Capabilities within the capability directory are further organized into groups called the *capability directory nodes*, abbreviated *cd-nodes*. They are identified by both a system-wide unique name created by and visible only to the kernel, and by user-specified names. In general, cd-nodes contain other information along with capabilities. Cd-nodes are linked to other cd-nodes through capabilities. The same cd-node may be linked to several cd-nodes under possibly different user-specified names. All capabilities pointing to a cd-node have equal status. That is, cd-nodes are unique and are *not* contained within other cd-nodes. A cd-node exists independently of any other cd-node and disappears along with the last capability link to it, if it is not explicitly destroyed. In this way the capability directory is structured as a graph in which nodes (each node corresponds to a cd-node) are connected by edges corresponding to capabilities. Figure 2 shows a sample capability directory.

The capability directory may contain two kinds of cd-nodes: *subdirectories* and *manager definitions* (see figure 3). Note that an *asterisk* next to attribute names used in the figures designates that the attribute cannot be modified by the users but is maintained by the kernel.

A subdirectory is a list of capabilities. It is merely an organizational unit of the capability directory, similar to a file directory in a file system. At any time, every process in the system is associated with a single subdirectory in the capability directory designated as its *active directory*. The active directory of a process is the set of capabilities from the capability directory that a process may use or exercise.

[3] A scheme for revoking transferred capabilities anytime after the transfer is discussed in [8].

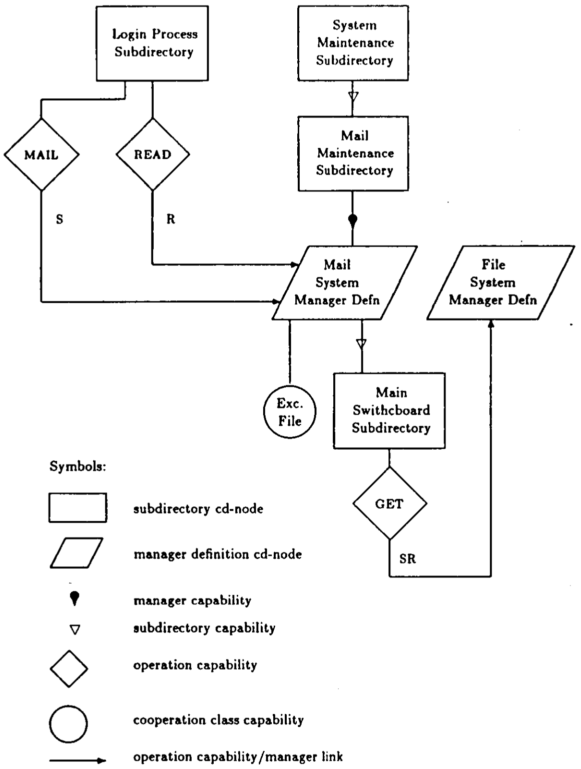

Login Process Subdirectory
System Maintenance Subdirectory
MAIL
READ
Mail Maintenance Subdirectory
S
R
Mail System Manager Defn
File System Manager Defn
Exc. File
Main Swithcboard Subdirectory
GET
SR
Symbols:
subdirectory cd-node
manager definition cd-node
manager capability
subdirectory capability
operation capability
cooperation class capability
operation capability/manager link

Figure 2: Example of Capability Directory Segment
This is a sample capability directory of a basic mail system. The Mail Maintenance Directory which contains the Mail System Manager Definition is created by the creator of the system and stored in the System Maintenance Directory. The two operations defined on the Mail System, namely the Mail and Read, are contained in the login process subdirectory. Each user gets capabilities for these operations during the login procedure. In the default subdirectory of the Mail System, operations on the File Manager exist that permit access to files.

The active directory of a process is one component of a process protection domain. The other component is its current set of transient capabilities; this is discussed later. A process may dynamically switch from one protection domain to another by changing to a new active directory or changing the contents of its current active directory, if it has the privilege to do so.

Manager definitions constitute one of the novel features of the Gutenberg system. All processes in the system are instantiated from manager definitions. Thus, a manager definition provides information necessary for instantiating the manager process, as, for example, a capability for the file containing the executable image (object module) of the process, and the initiation protocol (see section 3.2) for determining the manner in which ports are connected to manager processes. It also includes a capability for a subdirectory containing the privileges that all processes instantiated from the manager definition will initially possess.

One other component of a manager definition node is a set of *port descriptors.* This set corresponds to the set of operations defined within the manager. Each port description contains a *generic operation name* (a name specified by the user at manager creation time), and the type of port through which a user requests this operation.

Capabilities in cd-nodes inherit all the properties of the capability directory in that they are stable and sharable but are not owned by processes. These capabilities are called *stable* capabilities and can only reside in the capability directory.

Capabilities in Gutenberg consist of three parts: a specific kernel primitive, a list of parameters for the primitive, and a list of primitives that can be used to manipulate the capability itself, which are called *capcaps*, for capabilities on a capability.

A capability permits a process which possesses it to invoke the specific kernel primitive it contains. This primitive is also called *primary* kernel primitive in order to distinguish it from

Subdirectory cd-node Contains a set of capabilities, and has the following attributes:

subdirectory id* system-wide unique identifier of the subdirectory.

use count* the number of stable and transient subdirectory capabilities (including those in manager definition cd-nodes) pointing to this cd-node.

active directory count* the number of processes having this cd-node as their active directory.

Manager Definition Node Provides information necessary for instantiating a manager process. The manager definition cd-node has nine attributes.

manager id* system-wide unique identifier of the manager cd-node.

initial active directory a subdirectory capability pointing to the subdirectory cd-node that will become the active directory of the manager process when it is initiated.

initial process image a privilege (represented by a cooperation class capability) for a file containing the object code to be executed when the process is initiated.

manager dependency indicates whether the existence of a manager process is dependent on the existence of ports connected to it.

initiation protocol indicates when a new manager process is created or an existing one is connected to when a port to the manager is created.

port descriptors is the list of (port type, generic operation name) pairs for operation capabilities that may be linked to this manager cd-node. The port type specifies whether it is a Send, Receive or Send-Receive port as well as the format of the arguments that may be passed over the port as part of a message or request-details.

manager use count* the number of manager capabilities pointing to this cd-node.

operation use count* the number of operation capabilities linked to this cd-node.

port use count* the number of ports associated with this cd-node.

Figure 3: Attributes of Capability Directory Node

Operation Capability Provides the privilege to create ports. Operation capabilities have six attributes:

- **operation name** user-specified name of the operation. This name becomes the object operation of created ports. This name also serves to identify the operation capability.
- **generic operation name*** name of the corresponding operation defined in the manager definition cd-node. This name can be the same as or different from the operation name.
- **port type*** specifies the port type for this operation that may be linked to the corresponding manager definition cd-node. The port type can either be Send (S), Receive (R), or Send-Receive (SR).
- **message format*** specifies the arguments that may be passed over the port as part of the message, and the request-details in case of Send-Receive port type.
- **cooperation class(es)** restrict how the port will be connected to a manager process.
- **manager id*** a pointer, global name, to the manager to which this operation capability is linked.

The capcaps of the operation capability are: COPY, TRANSFER, REGISTER, REMOVE, HOLD, MERGE, MODIFY-CAP, MODIFY-CAPCAP, and VIEW-CAP.

Figure 4: Attributes of an Operation Capability

the other kernel primitives that manipulate the capability itself.

The capcaps determine how the capability may be modified and used. Capcaps include the privilege to *transfer* (to another process), *copy*, *register* (make stable), *hold* (make transient), *merge* (with other mergeable capabilities), *view*, and *modify* the capability. Each capcap may be active, in which case the corresponding kernel primitive may be invoked for the capability, or inactive, in which case the corresponding kernel primitive cannot be invoked on the capability. Not every capcap makes sense for each type of capability. When we discuss the specific capabilities next we point out the capcaps that are applicable.

The parameter list may include names of cd-nodes as well as other capabilities (most notably, the cooperation class capability which is discussed later).

2.3.2 Types of Capabilities

There are four different types of capabilities that may be stored in the capability directory: *operation*, *subdirectory*, *manager definition*, and *cooperation class capabilities.*

An operation capability (figure 4) represents the privilege to create a port for use in requesting a particular operation on a given user-defined object type. Thus, the primary kernel primitive of the operation capability is **Create-port**. One parameter of the operation capability is the operation name, which becomes the operation requested via ports created from this capability. This name also serves to identify the operation capability in the subdirectory in which the capability is contained. Another parameter of the capability is the name of a manager definition cd-node in the capability directory that the operation capability is *linked* to. This manager definition is used by the kernel to determine whether a newly created port is to be connected to a new server process instantiated from the manager definition or to an already existing one. It is also used by the kernel in conjunction with a third parameter, the generic operation name, to determine whether the requested operation is currently supported by the manager; this is checked by examining whether the operation generic name is part of the port descriptors in the manager definition.

The primary kernel primitive in the manager definition capability (figure 5) is **Create-operation**, which is used to create operation capabilities, linked to the manager definition named in the capability. Since an operation capability can be used to create a port to access a protected object, a manager definition capability signifies the privilege to provide other processes with specific types of access to their objects. This effectively is the privilege to control access to the object's type.

The primary kernel primitive associated with the subdirectory capability (figure 6) is **Change-directory**. The Change-directory primitive is used by a process to change its active directory to the subdirectory named in the subdirectory capability.

The subdirectory capability also contains a set of *subdirectory rights*. When a subdirectory capability is exercised to make a subdirectory active, the subdirectory rights override the capcaps of each individual capability in the subdirectory, and this further restricts the use of the capabilities registered in the subdirectory. This restriction during the changing of the active directory, referred to as *privilege filtering*, allows a fine granularity of control over the use of capabilities within an active directory. This is vital for supporting an effective mechanism that allows processes to switch from one protection domain to another dynamically. In this situation, when a process wants to switch to a new protection domain, it has to traverse the capability directory and change to a new active directory. While traversing the capability directory, a process may have to visit intermediate subdirectories which contain capabilities that the process need not be authorized to exercise or even view. By deactivating all rights except the CHANGE-DIR along the path between the

Manager Definition Capability Provides the privilege to create operation capabilities. The manager definition capability has the following attributes:

- **manager definition name** user-specified name used to identify the manager definition cd-node to which the created operation capabilities are linked to. This name also serves to identify the manager definition capability.
- **cooperation class(es)** restrict who may create operation capabilities linked to the definition manager. Possession of one of the specified cooperation class capabilities is required when exercising the **Create-operation** primitive.
- **manager id*** a pointer, global name, to the manager definition cd-node corresponding to this capability.

The capcaps of the manager definition capability are: COPY, TRANSFER, HOLD, REGISTER, REMOVE, DESTROY-NODE, MERGE, MODIFY-CAP, MODIFY-NODE, MODIFY-CAPCAP, VIEW-CAP, and VIEW-NODE.

Figure 5: Attributes of a Manager Definition Capability

Subdirectory Capability Provides the privilege to change the process' active directory. The attributes are:

subdirectory name user-specified name of the subdirectory cd-node which becomes the process' active directory when the **change-directory** privilege is exercised. This name also serves to identify the subdirectory capability.

cooperation class(es) used to restrict who may make the subdirectory active. When exercising the **Change-directory** privilege, a process must possess one of specified cooperation class capabilities or else the primitive is illegal.

subdirectory right restricts how cd-nodes and capabilities contained in the subdirectory may be used; each right may be ON (active) or OFF (inactive); the rights are: TRANSFER, COPY, REGISTER, REMOVE, HOLD, MERGE, VIEW-CAP, VIEW-NODE, MODIFY, DESTROY-MANAGER-NODE, DESTROY-DIR-NODE, CHANGE-DIRECTORY, CREATE-PORT, and CREATE-TYPE.

subdirectory id* a pointer, global name, to the subdirectory cd-node corresponding to this capability.

The capcaps of the subdirectory capability are: COPY, TRANSFER, REGISTER, REMOVE, HOLD, DESTROY-NODE, MERGE MODIFY-CAP, MODIFY-CAPCAP, VIEW-NODE and VIEW-CAP.

Figure 6: Attributes of a Subdirectory Capability

Cooperation Class Provides a wild card privilege that may be merged with any other capability type. It has two attributes:

class name user-specified name used to identify the cooperation class in the current protection domain.

class id* system wide unique identification of cooperation class.

The capcaps of the cooperation class capability are: COPY, TRANSFER, HOLD, REGISTER, REMOVE, MERGE, VIEW-CAP, MODIFY-CAP, and MODIFY-CAPCAP

Figure 7: Attributes of Cooperation Class Capability

initial and goal subdirectory, the mechanism for switching to a new domain becomes simple, and the security of the system is not compromised.

The fourth type of capability, the cooperation class capability, represents the privilege of a process to participate in a cooperative activity identified by a unique identifier, the *class id* (figure 7). The cooperation class capability may be associated with any other capability type, thus providing a wild card privilege. Hence, the primitive associated with this capability is ANY denoting its wild card property. When a cooperation class capability is associated with either a subdirectory or a manager definition capability, it restricts the invocation of the corresponding primary primitive to the processes which possess the cooperation class capability. In effect, such processes become members of the cooperative activity represented by the capability. In this way cooperation class capabilities complement the role of capcaps by determining how these capabilities may be exercised. In the case in which a cooperation class capability is associated with an operation capability, it specifies a cooperative activity, for example a communication with a manager of a shared object. Here, the cooperation class capability can be used as a synchronization token or to identify either an instance of a manager or a particular instance of an object. It could also be used to identify a transaction, a file, or a process. It can be used to classify the users in the system into groups and divisions for administrative reasons. New cooperation class capabilities can be created, on request, by the kernel.

2.4 Transient Capabilities

Port capabilities and copies of stable capabilities from the capability directory are the transient capabilities. Two features distinguish the transient capabilities from the stable capabilities: they are owned by a single process, and therefore cannot be shared, and their existence is dependent on the existence of the process that owns them. All the transient capabilities a process owns exist only for the duration of the existence of the process and are destroyed when the process terminates. The transient capabilities possessed by a process, are stored in the process' ***capability list***, or ***c-list***. Capabilities in the c-list, as previously stated, together with the active directory of a process, form the protection domain of the process. Thus, a process, in addition to the capabilities in its active directory, may exercise the capabilities in its c-list.

A transient capability comes into existence when a process moves or copies a capability from its active directory, into its c-list (using a primitive called HOLD), when it receives the capability from another process via a port, when it creates a capability by invoking the proper create primitive, or when it creates a port. In the last instance, two port capabilities are generated to access the created port, one for the client of the port and the other for the server of the port. A process moves a capability in its active directory into its c-list in order not to lose the capability when it changes its active directory to another subdirectory.

The part of the c-list in which the kernel maintains the port capabilities of a process is called the *p-list*. The port capabilities are *exclusive* and as such, port capabilities are inherently transient, cannot be shared or copied. However, either the client or the server process may transfer its port capability to another process. The transferring may be either temporary (the semantics of a lend with the ownership retained) with the SEND-RECEIVE primitive or permanent with the SEND primitive. In the latter case, the ownership of the port is transferred along with the port capability.

The only capcap associated with a port capability is *transfer*. The transfer capcap in the client's port capability is set to the value of the corresponding transfer capcap in the operation capability used. If the used operation capability is stable, then the transfer right of the client's active directory is also taken into consideration. The transfer capcap in the server's port capability is always enabled. A server process may transfer a port that it serves, to another server as long as the port functionality is preserved. This is essential for supporting the implementation of load balancing algorithms and realizing a hierarchy of object managers, features which can improve the performance, flexibility and reliability of a distributed system. A common example of a hierarchy of managers is a manager structured based on the master/slave model, in which a process can only create a port for requesting an operation to the master process. The master process decides which slave process should service the request and passes the port capability to it. A process may reset the transfer capcap of the port capability when it passes the capability to another process, to prevent further transferring.

Transient capabilities contribute to the flexible use of capabilities in Gutenberg without compromising the security of the system because the c-list is saved in the PCB and may only be manipulated through kernel primitives.

The kernel primitives that manipulate the capabilities both within a c-list and an active directory, fall into two classes: *generic* and *special primitives*. Generic primitives are further classified into *constructive* in that they do not affect the participating capabilities, and *destructive*. Constructive primitives are designated by the ending **-C** attached to their names. Recall that a number of capcaps and rights correspond to each primitive, and must be active in the capability on which the primitive is invoked, and must be allowed by the rights of the active directory of the invoking process.

The eleven generic kernel primitives are: *Create, Register, Register-C, Remove, Hold, Hold-C, Drop, View, Modify, Merge* and *Merge-C*. Here is a brief description of their functionality (Details concerning the primitives may be found in [3]).

Create These primitives create a capability. **Create-port**, and **Create-operation** are instances of this generic primitive. A **Create** always creates a transient capability which may then be registered or transferred with all or part of its privileges retained. Creating an operation capability requires a manager capability, and creating a port capability requires an operation capability. Other **Create** primitives require no privilege.

Register and Register-C These primitives make a transient capability, or one derived from transient capabilities, stable. A process may reset part of the capcaps and/or rights of a capability, when it registers the capability. The purpose of these primitives is two-fold: to allow a process to store a capability for future reference in another session; and, to allow a process to share a capability with other processes which are not currently instantiated, but share access to a subdirectory.

Remove These primitives delete a stable capability from the active directory.

Drop These primitives delete a transient capability from the c-list.

Hold and Hold-C These primitives make a stable capability, or one derived from stable capabilities, transient. A process may reset part of the capcaps and/or rights of a capability, when it holds the capability. The purpose of these primitives is to allow a process to retain a capability from its active directory when it changes its active directory to another subdirectory.

View These primitives bring a copy of a transient or stable capability, or a cd-node into the address space of a process. Partial views are also facilitated in the case of a subdirectory; it is possible to bring capabilities of a specified type into the address space of the process. The purpose of these primitives is to allow a process to examine capabilities it possesses, and, if desired, use this information to modify or create new capabilities.

Modify These primitives allow a process to modify an existing transient or stable capability, or a cd-node.

Merge and Merge-C These primitives allow a process to merge two compatible capabilities to obtain another. Two capabilities are compatible if they are of the same type and have identical non-modifiable attributes (attributes whose values cannot be changed with the **Modify** primitive).

As has previously been discussed, the manager definition and subdirectory capabilities are slightly different from other capabilities. These capabilities contain pointers to manager definition and subdirectory cd-nodes, respectively. When one of these capabilities is created with the **Create** primitive, the cd-node is created as well. These cd-nodes exist in the system until either they are explicitly destroyed by using the proper **Destroy** primitives, or all of the capabilities that point to them are deleted using the **Remove** or **Drop** primitives.

The kernel uses the c-list and the active directory of a process to check whether it has a legitimate privilege for executing a primitive it has requested. Therefore, the consistency and availability of the capability directory is fundamental to the correct operation of the protection mechanism. As is commonly done with resources in distributed systems, the capability directory is physically distributed, though still logically unified, across the distributed system as we discuss below.

3 USER-DEFINED OBJECTS

Recall that the Gutenberg kernel is not involved in the protection of objects that are purely local to a process. User-defined objects are those objects managed by one process but accessible by other processes via operations requested using ports. Here, we discuss how user-defined objects are created, shared, and protected.

3.1 Type Creation

User-defined types in Gutenberg are represented by manager definition cd-nodes (figure 3). A manager definition is created by a process invoking the **Create-manager** primitive. Recall that no special privilege is required to invoke this primitive. In invoking the primitive, a process must provide five parameters: A cooperation class capability identifying the file containing the executable image for the process; a subdirectory capability identifying the *default directory,* the subdirectory which becomes the active directory of any process instantiated from this manager definition; the *operation list,* the specification of the operations that are implemented by the manager; the *manager initiation protocol,* indicating how manager processes are instantiated; and, the *manager dependency* indicating whether a manager process will be destroyed when all ports connected to it are destroyed.

Upon successful execution of the **Create-manager** primitive, the kernel places a manager definition capability, pointing at the new manager definition and containing its name, in the process's c-list. After the manager definition and the corresponding manager definition capability are created, the process may use the manager capability to invoke the **Create-operation** primitive to create the operation capabilities for the type. These operation capabilities can then be stored in the capability directory or distributed over ports to processes wishing to use the type.

3.2 Manager Initiation Protocols

The manager initiation protocol specified when creating a manager definition determines the manner in which ports are connected to the manager processes instantiated from the definition. It specifies whether all the object instances of a type are managed by one process or each object is managed by different processes. There are three initiation protocols in Gutenberg: *conservative, creative* and *class conservative.* In the conservative manager initiation protocol, a manager process is instantiated from the manager definition only if there is no other manager process executing in the system which was instantiated using this manager definition. If such a process already exists, the port being created is attached to this process. This protocol provides the means to produce a manager process that manages all the objects of a type, and to automatically connect port-creating processes to this manager. Using this protocol, the manager can be informed of the object being accessed at port-creation time using a cooperation class capability. Instantiating more than one conservative manager requires creating more than one manager definition.

The creative protocol creates a new manager process from the creative manager definition cd-node for each new port created. This protocol allows a process to create a port to a new process under all situations. This protocol allows a process to isolate the newly created manager in order to ensure that the manager cannot leak information. However, it cannot support multi-port interconnections between a specific client and a specific server.

The third protocol is the class conservative manager initiation protocol. It allows new managers to be instantiated selectively based on the cooperation class capability supplied at port creation time. The class conservative manager is typically designed to manage one object of the type, and may serve multiple ports from any number of processes.

In the class conservative protocol, when a port is created using an operation capability, the kernel checks to see if a process associated with the specified class id has been instantiated from the manager definition pointed to by the operation capability. If so, the port is connected to this manager. If not, a new manager is instantiated and associated with the specified class id.

Both conservative protocols allow any two processes to communicate indirectly through a conservative process, although they cannot establish direct, full duplex interconnections. However, using these protocols and by allowing processes to pass port use privileges via ports, the one-to-one process intercommunication topology adopted by Gutenberg can be expanded to arbitrary topologies. For a more detailed description of manager initiation protocols, see [12].

3.3 Object Protection and Sharing

Once a type is created, distributing privileges to allow other processes to use the object type corresponds, in Gutenberg, to distributing operation capabilities linked to the manager definition. As has previously been discussed, for a process to access a shared resource, a port is needed between itself and the process managing the object. Establishing a port involves checking for an operation capability in the active directory or c-list of the requesting process. Thereafter the kernel performs access authorization for a user-defined operation simply by checking that the requesting process has the privilege to access the port associated with the operation. Thus, creating and accessing user-defined objects involves using kernel-defined capabilities to authorize access to kernel-defined objects, and does not involve checking user-defined capabilities as in other capability-based systems.

In Gutenberg, process interconnections can change dynamically through the transfer of capabilities on ports. There are three ways in which one process can transfer some of its capabilities to another; The transferred capabilities are always placed on the receiving's process c-list. Since the kernel is the manager of the capabilities and ports, it monitors the transfer of capabilities between processes.

The first method of capability transfer is the transfer of a port capability. The sending process loses the port capability, and therefore the privilege to execute the object operation associated with the transferred port; the receiving process obtains this privilege.

The second method of capability transfer is the transfer of an operation capability (which can be associated with a cooperation class) that can be used to create any number of ports to access an object (identified by the cooperation class).

The third method of capability transfer is by registering the capability to be transferred in a subdirectory and transferring the subdirectory capability that points to it. Using this method a process may transfer a number of capabilities at once with the minimum communication overhead.

In all three methods, capabilities are transferred either by SEND as part of the message or by SEND-RECEIVE as part of the request details. These two mechanisms of capability transfer are distinguished by the semantics of the transferring. The transfer by SEND is permanent, whereas the transfer by SEND-RECEIVE is temporary and the transferred capabilities can be held only while the recipient is processing the request to which it pertains. When the recipient executes the SEND that satisfies the request, the kernel automatically returns the *outstanding* capabilities to the process which executed the SEND-RECEIVE. For more detailed discussion of the privilege transferring mechanisms as well as their implications see [8].

4 DISTRIBUTED GUTENBERG KERNEL

An instance of the Gutenberg kernel is running on each site in the system. From the objects that the kernel maintains only the capability directory needs to be distributed. The other three objects, namely processes, ports and transient capabilities, are naturally distributed since processes always execute on a single site, and ports and transient capabilities can only be used by the process which possesses them.

The capability directory is partitioned and replicated to ensure availability and reliability. Manager definitions are replicated only at the sites in which manager processes from these can possibly be instantiated. Subdirectories are replicated in locations where they are expected to be used and in a number of other locations in a manner that meet the resiliency requirement and balance the distribution of local directory storage space. The site where a cd-node is created has a copy of that cd-node and also keeps track of which sites have copies of that cd-node.

The set of capabilities and cd-nodes from the capability directory that resides on a site forms that site's *local directory*. The site's local directory dynamically expands and contracts to accommodate the needs of any process executing on the site. *Dynamic adjustments* pertain only to the migration of subdirectories; modifications to manager definitions are expected to be relatively rare to justify their migration. A subdiretcory is migrated only when processes make repetitive use of the capabilties located there, justifying the move. Even then, only (lockable) portions of the subdirectory are copied in a *lazy copy* fashion.

Synchronization of access to components of the capability directory is achieved with the use of two locking-based concurrency control schemes with the granularity of locks being on portions of cd-nodes. For example, a subdirectory is associated with three locks; one is used for the set of operation and cooperation class capabilities; the second for subdirectory capabilities; and the third for manager definition capabilities.

Two concurrency control schemes used are the *primary two-phase locking scheme* [2] where a single copy is designated as the site to gain all locks, and the *basic two-phase locking scheme* [2] where to lock a cd-node for reading it is necessary to lock only the local copy whereas a write requires obtaining the write lock on all replicates of the cd-node. Which scheme is used for a specific cd-node depends on the likelihood of its predominant access by processes on one site or on multiple sites. A *dynamic recategorization* allows the kernel to adapt to the system behaviour by changing the concurrency control mechanism for a cd-node in response to the nature of its use.

A two-phase commit protocol is used to achieve a reliable distributed commitment and ensure atomicity. For a more detailed discussion of issues involved in the distribution and the approach adopted see [7].

5 A COMPARISON WITH RELATED SYSTEMS

A large part of the improvement in programming languages has come from the promotion of data and procedural abstraction as a major tool for structuring modules. This led to the adoption of abstraction and encapsulation mechanisms in Gutenberg, as in many other systems, e.g. Argus [6], NIL [13]. While in these two systems, the mechanism for dynamic module interconnection control is built within a programming language, the approach taken in Gutenberg separates process interconnection control from programming languages. Gutenberg supports the dynamic control of process interconnections through the use of capabilities. However, it is different from the other protection-oriented operating systems such as Hydra [16] and iMAX [4], in that it adopts a non-uniform object-orientation.

A few systems provide port-based communication facilities using functional addressing [12], but none ties protection so closely to communication as Gutenberg does. The Accent system is port-based and supports asynchronous communication with process transparency [9]. Communication in Gutenberg is also similar to the mechanisms used in Intel iAPX-432 [4], DEMOS [1] and NIL [13]. In all these systems, apart from NIL, even though a communication link could be typed, thus restricting its use, they do not support the concept of restricting access to shared objects by restricting the creation of communication channels is not supported directly, as in Gutenberg.

As mentioned earlier, a unique feature of Gutenberg is the capability directory, which contains stable capabilities in a unified structure controlled by the kernel. Other systems, such as Hydra, iAPX 432, and CAL [5] allow capabilities to be stored in inactive objects (i.e., data structures, as opposed to processes) that are not kernel objects. The problem of how to allow such objects to be permanently stored in secondary memory is noted in [5]. Having a unified structure for stable capabilities that is separate from user-managed data facilitates their management by the kernel and their use by application processes.

6 CONCLUSION

The Gutenberg system is a novel attempt to facilitate the design and structuring of distributed computations in an understandable and reliable manner that is suitable for validation. The crux of the Gutenberg approach is the use of port-based communication, non-uniform object-orientation and decentralized access authorization using ports and the capability directory. This nonprocedural directory provides an abstract view of the functional building blocks of a large system of distributed cooperating modules and should serve the goal of understandability and verifiability.

In this paper we discussed the design of the Gutenberg kernel. In particular, we presented the kernel primitives, i.e., kernel-implemented operations for manipulating the capability directory and ports. Since process creation and destruction are byproducts of port operations, there are no explicit primitives to deal with processes in Gutenberg. Gutenberg does allow users to construct managers for user defined objects. Such manager definitions are registered in the capability directory.

A Privilege in Gutenberg is represented by capabilities which can have two levels of persistence, transient for those capabilities which persist only as long as an owning process exists, and stable for those capabilities in the capability directory, whose existence is independent of processes. Gutenberg has mechanisms for achieving transfer of privileges represented by both transient and stable capabilities. Some of the highlights of these mechanisms include:

- Using both unidirectional and bidirectional communication primitives and associating them with permanent (unidirectional) and temporary (bidirectional) granting of privileges in order to provide flexibility in privilege granting while keeping the kernel simple.
- Typing ports with respect to the ability to transfer privileges on them in order to expedite communication in cases where no privilege transfer can be made.
- Restricting ports to connecting one client process with one server process in order to simplify interprocess communication in general and the transfer of privileges in particular.

Design and implementation of a Gutenberg kernel (built on top of UNIX) is currently nearing completion. Experimentation with this kernel should provide qualitative evaluation of the advantages of the Gutenberg approach.

References

[1] Baskett, F., Howard, J., Montague, J., 'Task Communication in DEMOS,' *Proceedings of the 6th ACM Symposium on Operating System Principles*, November, 1977.

[2] Bernstein, P., Goodman, N., 'Concurrency Control in Distributed Database Systems,' *ACM Computer Surveys*, vol. 13, no. 2, June 1983.

[3] Chrysanthis, P.K., Ramamritham, K., Stemple, D.W., Vinter, S.T., 'The Gutenberg Operating System Kernel,' Dept. of Computer and Information Science Technical Report 86-06, University of Massachusetts, February, 1986.

[4] Cox, G., Corwin, W., Lai, K., Pollack, F., 'A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment,' Intel Corporation, 1981.

[5] Lampson, B. W., Sturgis, H. E., 'Reflections on an Operating System Design,' *Communications of the ACM*, vol. 19, no. 5, May, 1976.

[6] Liskov, B., Scheifler, R., 'Guardians and Actions: Linguistic Support for Robust, Distributed Programs,' *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, January, 1982.

[7] Ramamritham, Stemple, D., Vinter, S. T., 'Decentralized Access Control in a Distributed System,' *Proceedings of the 5th International conference on Distributed Computing Systems*, May 1985.

[8] Ramamritham, K., Briggs, D., Stemple, D., Vinter, S. T., 'Privilege Transfer and Revocation in a Port-Based System,' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, May 1986.

[9] Rashid, R., Robertson, G., 'Accent: A Communication Oriented Network Operating System Kernel,' Carnegie-Mellon University Technical Report, April, 1981.

[10] Ritchie, D. and Thompson, K., 'The UNIX Time-Sharing System,' *Communications of the ACM*, vol. 17, no. 7, July, 1974.

[11] Stemple, D., Ramamritham, K., Vinter, S., 'Operating System Support for Abstract Database Types,' *Proceedings of the 2nd International Conference on Databases*, September, 1983.

[12] Stemple, D., Vinter, S., Ramamritham, K., 'Functional Addressing in Gutenberg: Interprocess Communication Without Process Identifiers,' to appear in *IEEE Transactions on Software Engineering*, 1986.

[13] Strom, R., Yemini, S., 'NIL: An Integrated Language and System for distributed Programming,' *Proceedings of SIGPLAN '83, Symposium on Programming Languages*, August, 1983.

[14] Vinter, S. T., 'A Protection Oriented Distributed Kernel,' Ph.D. Thesis, University of Massachusetts, August 1985.

[15] Vinter, S. T., Ramamritham, K., Stemple, D., 'Recoverable Communicating Actions,' *Proceedings of the fifth International Conference on Distributed Computing Systems*, May 1986.

[16] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., Pollack, F., 'HYDRA: The Kernel of a Multiprocessor Operating System,' *Communications of the ACM*, vol. 17, no. 6, June 1974.

CARAT: a Testbed for the Performance Evaluation of Distributed Database Systems[1]

Walt Kohler and Bao-Chyuan Jenq[2]

Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003

Abstract

This paper describes the goals, design, and current implementation of CARAT, an operational distributed software testbed. CARAT is a simplified but complete transaction processing environment. It was designed to be a flexible tool for the testing and performance evaluation of distributed concurrency control, deadlock detection and avoidance, and recovery mechanisms used in distributed database systems. CARAT contains all the major functional components of a distributed transaction processing system (transaction management, data management, log management, communication management, and catalog management) in enough detail so that the performance results will be realistic. Some of the protocols currently implemented in CARAT and discussed in this paper are: a two-phase locking protocol with distributed deadlock detection, a distributed version of optimistic concurrency control, before-image and after-image journaling mechanisms for transaction recovery, and a two-phase commit protocol for global consistency of distributed transactions.

1 Introduction

During the past decade, concurrency control and recovery in transaction-oriented database systems has been the subject of intensive research [GRAY79, BERN81, KOHL81, HAER83]. Even though a wide variety of concurrency control and recovery algorithms have been proposed, there is currently little quantitative knowledge on how the choice of algorithms impacts the performance of the overall transaction processing system [STON83].

There are three major approaches to system performance evaluation: simulation, analytical modeling, and measurement. Simulation studies are most widely used primarily because system parameters and configurations are relatively easy to change and any desired level of detail can be captured. However, it often requires a tremendous amount of computation to gather convincing simulation results for complex systems. Furthermore, the results can be misleading if some significant factors are ignored by the simulation model.

Analytical modeling is relatively inexpensive to perform and often provides good insight into system behavior, if an appropriate model can be found. But analytic models for complicated systems are difficult to construct and solve without introducing simplifying assumptions that may be inappropriate.

Both analytical and simulation models require estimates of the system resource requirements of the primitive elements of the algorithm as implemented in the system under study. Usually they can only be obtained by extensive measurement of a prototype implementation. Performance measurement of a real system is more conclusive than either simulation or analytical analysis. However, performance data is time comsuming to collect and reduce and it is not feasible to generalize the results of performance measurement to different system configurations and designs without using some type of model.

One way to obtain the model parameters needed for analytical and simulation studies is through the implementation and measurement of a *testbed* system. A testbed system is an operational prototype system that is simple and flexible enough to permit the designer to "plug in" various algorithms or strategies and then measure and study their impact on system performance. As discussed in a special issue of *Computer* (Vol. 15, No. 10, 1983), testbeds are becoming recognized as an important research tool because "We expect experimentation with distributed systems to provide

1. an improved understanding of the functional requirements and operational behavior of distributed systems,
2. measurements from which quantitative results about distributed systems can be derived,
3. an integrated environment in which interrelations of solutions to individual problems can be evaluated, and
4. a design environment in which design decisions can be based on both theoretical and empirical studies." [BERG82, page 9]

We believe that an evaluation approach based on a combination of performance measurement and modeling studies is necessary in order to characterize and understand the impact

[1] This work was supported by the National Science Foundation, grant numbers ECS-8120931, ECS-8340029, and SDB-8418216.

[2] Bao-Chyuan Jenq is now with Digital Equipment Corporation, Nashua, New Hampshire.

of different concurrency control and recovery algorithms on the performance of a distributed database system. The testbed system we have implemented for the performance measurement of concurrency control and recovery algorithms is called **CARAT** (**C**oncurrency **A**nd **R**ecovery **A**lgorithm **T**estbed). Some of the protocols currently implemented in CARAT and used in performance measurement experiments are:

- a two-phase locking protocol with distributed deadlock detection based on probes
- a distributed version of optimistic concurrency control
- before-image and after-image journaling mechanisms for transaction recovery
- a two-phase commit protocol for global consistency of distributed transactions.

Performance comparisons of concurrency control algorithms have been carried out by a large number of researchers using simulation and analytical models, for example, by Irani and Lin [IRAN79], Ries and Stonebraker [RIES79], Lin [LIN81], Thomasian [THOM82], Galler [GALL82], Menasce and Nakanishi [MENA82], and Carey and Stonebraker [CARE84]. But there are important issues that have received little consideration in previous studies:

- There has been little empirical comparison of different concurrency control and recovery algorithms in distributed database systems, as pointed out by Stonebraker et al.[STON83]. Moreover, even though system resource requirements for the basic components of concurrency control and recovery algorithms are necessary for realistic performance modeling studies, few results have been reported.
- The validity of the various assumptions made in the models have not been carefully examined. In fact, the previously reported results have been contradictory. Refer to the recent modeling work of Agrawal, Carey, and Livny [AGRA85a] for an examination of the discrepancies.
- Most of the previous modeling studies considered only the performance of concurrency control algorithms in a centralized database system. Few studies have addressed the distributed database environment [GARC79, LIN81, NAKA82].
- Although concurrency control and recovery mechanisms are intimately related, except for the recent modeling work of Agrawal and DeWitt [AGRA85b] they have been treated as two separate problems when studying their impact on database system performance.

In this paper we describe the design and implementation of the CARAT testbed. Our goal is to show how we constructed the testbed and to identify our major design decisions. Since our initial performance studies have concentrated on transaction management issues, i.e., concurrency control and recovery protocols, we have separated transaction management from data management as much as possible. A discussion of our performance measurement methodology and experimental results can be found in [KOHL86]. We have also developed and validated an analytical queueing network model for the CARAT distributed transaction processing system [JENQ86b]. The current paper serves as an introduction to the testbed used in those studies. The next section begins with an overview of the physical design of CARAT.

2 Overview of the CARAT Architecture

The *process* and *communication* structure are the two major architectural issues in developing a testbed for the efficient execution of distributed transactions. In determining the process structure, there are three primary considerations. First, it is important to minimize communication overhead and context switching. Second, it should support a high degree of concurrency among multiple transactions, called inter-transaction concurrency. Finally, to exploit the processing parallelism available in a distributed system, it should be possible for requests from a single transaction to be executed sequentially or in parallel at multiple nodes, this is called intra-transaction concurrency.

CARAT is implemented as a set of cooperating server processes which communicate via a uniform and efficient message passing mechanism. After experimenting with an early prototype [GARC83], we decided to implement CARAT using two levels of servers at each node. Figure 1 illustrates the process and message structure of CARAT for two nodes, but the architecture generalizes to any number of nodes. The top level server process, called the TM server, is the Transaction Manager. At the next lower level there is a pool of Data Managers, designated the DM servers. This structure cleanly divides most of the functions of transaction management and data management between the two server categories. (See the book by Ceri and Pelagatti [CERI84] for a discussion of distributed transaction management issues and architectures.) The TM and DM servers work cooperatively to service transaction requests from user application processes. Each user process, labeled TR, submits transaction requests sequentially. In our experiments each node runs on a single processor.

One function of the TM server is to act as the inter-process communication agent for user processes. Inter-node communication links are established at system start-up time among the TM servers and are maintained forever. There is no communication link between a TM server and a remote DM server. Since all inter-process connections are maintained, there is no connect or disconnect overhead for individual transactions. Each request to a remote DM server is routed through a remote TM server. This also reduces the required number of inter-node links, which would otherwise become overwhelming for an environment with many users at each node.

A user application process issues transaction requests only to its local TM server. The local TM server is called the Coordinator TM. A local transaction request is routed by the Coordinator TM to a local DM server while a remote request is sent to a remote (slave) TM server. The DM servers execute the transaction requests and send response messages to their local TM server. For responses to local requests, the TM server then routes the messages to the initiating user processes, while responses to remote requests are routed to the user processes

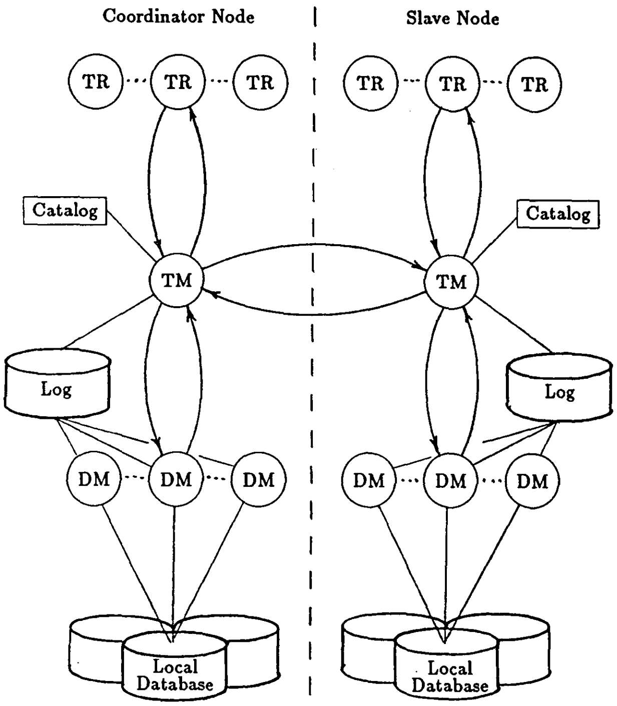


Figure 1: CARAT process and message structure.

through the initiating TM server.

A DM server for a particular transaction is dynamically allocated by the TM server from the DM server pool. The association of a DM server with a transaction is maintained throughout the lifetime of the transaction. For a distributed transaction, there is a DM server agent at each of the participating nodes.

Inter-transaction concurrency can be achieved by having multiple DM's active concurrently at one or more nodes on behalf of different transactions. The degree of inter-transaction concurrency at a node is thus bounded by the number of DM servers at the node. Intra-transaction concurrency can be achieved if a user transaction issues multiple requests in parallel for different DM server nodes.

We believe that the two-level server structure is well-suited for the distributed workstation and database server environment. The TM server may become a bottleneck when the message arrival rate is high. However, our experience so far has shown that the serialization delay due to the TM server is not a significant factor for a wide range of message request rates. Besides, our testbed architecture allows the TM server to control the effective multiprogramming level by simply queueing the message requests from transactions. By controlling the multiprogramming level, TM is able to control the level of conflict among transactions. This process structure also maps well into an environment where each node is a multiprocessor or tightly-coupled cluster. Each TM or DM process could run in a separate processor.

CARAT contains the functional components: *transaction management*, *data management*, *log management*, *communication management*, and *catalog management*. Transaction management, inter-node communication and catalog management are the responsibilities of the TM server processes. The data management function is handled by the DM server processes, which also hold the primary responsibility for log management if recovery based on logging is used. The CARAT system has been implemented for a high-speed network of DEC VAX/VMS[3] systems with DECnet/Ethernet connections . In the next section we will describe the communication management component. This will be followed by a section that sketches the execution of a transaction and then sections that describe the implementation of each of the other functional components.

3 Communication Management

To reduce programming effort and increase flexibility, a port interface is implemented by the communication management module. A uniform call interface is provided by the port interface for both intra-node and inter-node process-to-process communication. There is at least one port associated with each of the CARAT system processes (TM's and DM's) and the user transaction processes (TR's). The coupling of ports and processes enables us to categorize the port types by the associated process types. Thus, there are three basic port types in CARAT: TM, DM and TR. Each of the ports is named by port type, port number, and node number.

In essence, a port is an abstraction of a first-in-first-out queue containing messages destined for the associated process. Messages received either through inter-node or intra-node communication are all appended to the tail of the port queue. A process is transparently interrupted when a message arrives and it is then automatically resumed after the message has been placed on the queue. Both synchronous and asynchronous primitives for sending and receiving messages are provided by the communication module. Our experience has shown that asynchronous send and receive are essential for the operation of the TM server process due to its multi-threaded nature. Each of the server processes defines its port at system start-up time. By defining a port, the process has essentially set up an incoming channel from the port. Outgoing communication channels are established by connecting to the ports owned by the other communicating processes.

Intra-node process-to-process communication is implemented using the mailbox facility provided by the VAX/VMS operating system. Mailboxes support synchronous and asynchronous communication between processes by buffering messages in system dynamic memory. When using synchronous mailbox communication, the process that issues the receive blocks and waits, if it finds no message in the mailbox. It remains blocked until a message is sent to the mailbox by some other process. Likewise, a process sending synchronously is blocked until the message it sent is received by another process connected to the mailbox. Asynchronous communication, on the other hand, allows a sending process to continue before the message is read by a receiver and a receiving process to continue even if there is no message in the input queue.

For inter-node process-to-process communication, the underlying communication mechanism is based on DECnet virtual circuits. This shields undesirable conditions such as lost messages,

[3] DEC, VAX, VMS, and DECnet are trademarks of Digital Equipment Corporation.

message duplication, and out-of-order message delivery from the servers. Communication link failures and node crashes can be observed by DECnet and the involved communicating parties can be notified of the failures. DECnet supplied failure information can be used to supplement the TM server's timeout based failure detection mechanism. The only inter-node communication in the current CARAT architecture is between TM servers.

4 Model of CARAT Transaction Execution

A transaction is basically a sequence of host language statements, including terminal input/output statements, and database DML statements bounded by the transaction delimiters TBEGIN and TEND. In a distributed database system based on message-passing, each of these DML commands may be carried by a message to a DM server which will interpret and then execute the command. This approach requires one request/response message per DML command in addition to the processing overhead associated with the run-time interpretation of the command. To reduce message overhead, several DML commands with parameters can be grouped into a *request* and carried by a single message. To reduce the run-time interpretation overhead, the requests can be compiled.

In CARAT a request is compiled into a small object module and stored with the DM server. Requests are invoked at run time by a message that carries the request number and the actual parameters used by the request. Requests are the basic units of execution and distribution among participating nodes.

There are five basic message types issued by a CARAT transaction: TBEGIN, DBOPEN, TDO, T_ABORT, and TEND. A user process initiates a transaction by sending a TBEGIN message to its local TM server. This TM becomes the Coordinator TM for the transaction. A TBEGIN message is followed by a DBOPEN message and a sequence of TDO messages, each of which contains a transaction request. A DBOPEN message carries the description of the data objects (e.g., name of the area or file) upon which the subsequent TDO messages will operate. Upon receiving a DBOPEN message, the TM server determines the location of the data object by consulting its local catalog. The catalog is replicated at each site. If the data object resides locally, it will allocate a local DM server to serve the transaction if one is not already allocated and then forward the DBOPEN request to the DM server. For remote data objects, the TM forwards the DBOPEN message to the TM server where the data object resides. The remote TM server is called the Slave TM. Transactions initiated at another node are called foreign transactions for the Slave TM server. When a Slave TM receives a request from a foreign transaction, it assigns it a DM server, if it does not have one already, and then forwards the request. After processing the request, a DBOPEN_K acknowledgment is returned to the transaction. One of the parameters included in the DBOPEN_K message is the location of the data object for use in subsequent TDO requests.[4] Note that there is exactly one DM server which acts as an agent for a transaction at every node the transaction visits. The use of a catalog by the TM coordinator provides *location transparency* to transactions. This allows transparent file migration and simplifies application programming.

The TDO message carries a request number, a location indicator, and the actual parameters for the request. Each TDO message is routed by the coordinating TM server to either a local DM server, using a DOSTEP message, or a remote TM server, using a REMDO message. For TDO messages which expect data to be returned to the requesting transaction, an acknowledgment message, TDO_K, will carry the response data. Our current implementation assumes that the entire response set can fit into the message buffer.

The user process issues a TEND message to commit a transaction. To coordinate execution of distributed transactions, the cooperating TM and DM servers execute a commit protocol.

An example of the flow of messages that might occur during the execution of a simple CARAT transaction is shown in Figure 2. The sequence of events are time ordered from the top of the figure to the bottom. Each column represents the user process or a system process and the interchange of a message is shown by an arrow.

```
   TR         TM         Local DM     Remote TM      Remote DM

TBEGIN -->
       <--  TBEGIN_K
DBOPEN -->
             DBOPEN  -->
                     <--  DBOPEN_K
       <--  DBOPEN_K
DBOPEN -->
            REM_DBOPEN -------------->
                                       DBOPEN    -->
                                                 <--  DBOPEN_K
                     <----------------  REM_DBOPEN_K
       <--  DBOPEN_K
 TDO   --> (local request)
            DOSTEP   -->
                        [DB IO]{LOG IO}
                     <--  DOSTEP_K
       <--  TDO_K
  .     .     .
  .     .     .
  .     .     .
 TDO    --> (Remote request)
            REMDO    ---------------->
                                       DOSTEP   -->
                                                  [DB IO]{LOG IO}
                                                <--  DOSTEP_K
                     <----------------  REMDO_K
       <--  TDO_K
 TEND  -->
            PREPARE  -->
            PREPARE  ---------------->  PREPARE  -->
                        {DB IO}{LOG IO}           {DB IO}{LOG IO}
                     <--  PREPARE_K             <--  PREPARE_K
                     <----------------  PREPARE_K
           [LOG IO]
            COMMIT   -->
            COMMIT   ---------------->  COMMIT   -->
                         [LOG IO]                    [LOG IO]
                     <--  COMMIT_K              <--  COMMIT_K
                     <----------------  COMMIT_K
       <--  TEND_K
```

Figure 2: Flow of messages for a simple CARAT transaction.

[4] The "_K" on a message name is used to indicate an acknowledgment message.

5 Transaction Management

The basic function of transaction management is to coordinate distributed transaction execution to guarantee that all the participating nodes reach the same decision about the outcome of each distributed transaction, despite node, communication or transaction failures. This is achieved by implementing a commit protocol. Another responsibility of this module is to provide an interface to user transactions and to route requests and responses between the transaction, TR, and the local DM or remote TM processes based upon the transaction state information it maintains and the message type. In the following subsections we will discuss transaction state management, the two-phase commit protocol, and the failure handling mechanisms implemented in CARAT.

5.1 Transaction state management

Upon receiving a TBEGIN message, the transaction management component of the TM process registers the transaction by assigning it a globally unique transaction identification, TS. TS is a timestamp [LAMP78] formed by concatenating three numbers: a local counter, a node number, and a retry number. The local counter represents *logical time* and is incremented by each local transaction registration. TS is returned to the transaction in the TBEGIN_K acknowledgment message. All future messages sent by the transaction contain TS as identification. To maintain the correct partial ordering of transactions in the distributed system, the local counter is brought up-to-date every time it receives a message from another node.

The transaction management module keeps track of the transaction state of every active transaction. Each TM server process maintains two tables for transaction state information. The **Coordinator Table** contains *global* state information on each active transaction that was initiated locally. The **Slave Table** contains *local* state information on each active foreign transaction that has opened a database file and executed requests at that node.

The *global state* of a transaction is either: *inactive*, *active*, *collecting*, *committing*, or *aborting*. A TBEGIN message received by the Coordinator TM marks the beginning of the transaction's *active* state. The transaction normally remains in the active state until a TEND message is received by the TM coordinator from the transaction. While in the active state, the TM coordinator accepts and forwards DBOPEN and TDO messages for processing by local or remote DM managers. A TEND message triggers the TM coordinator into the transaction commit protocol.

The Slave Table maintains the *local state* of foreign transactions. There are six local states: *inactive*, *active*, *preparing*, *prepared*, *committing*, or *aborting*. An entry is maintained for each foreign transaction that is not inactive at the node. The local state becomes *active* when a REM_DBOPEN message is received and a DM server is assigned to the foreign transaction. Under normal operation the state remains active until a PREPARE message is received from the Coordinator TM process. This marks the beginning of the *preparing* state and a PREPARE message is sent to the assigned DM process. The *prepared* state is entered when the DM process returns a PREPARED_K message. This tells the Slave TM that the DM has safely stored the information to undo (or redo, depending on the recovery method used) the transaction in stable storage. A transaction in the *prepared* state means that the slave has given up its right to unilaterally abort the transaction. It must wait for the Coordinator TM server's commit or abort decision.

The transaction management component is also responsible for coordinating failure recovery. The three basic types of failures that are usually considered as recoverable are [GRAY79, HAER83]: *transaction failure*, *system failure*, and *media failure*. We assume that *transaction* failures do not corrupt the volatile or non-volatile database or the log, that *system* failures do not corrupt the non-volatile database or log, and that *media* failures do not corrupt the log or the database checkpoint. Transaction failures occur with the highest frequency and they are typically caused by user abort and deadlock abort. Before failures can be handled they must be detected. User abort failures can be detected by the underlying operating system and message system. Deadlock failures are detected by the deadlock detection component of lock based concurrency control schemes.

Since our performance experiments have only been concerned with transaction deadlock abort, we have not yet implemented a sophisticated user and node failure detection mechanism. The basic mechanism employed for user abort and node crash detection in the current implementation is message timeout. For example, if the Coordinator TM does not receive a message from an active TR within a predefined timeout interval, the TM coordinator assumes the transaction has failed and initiates an abort procedure. Similarly, a remote node crash or communication network failure is detected by the inability to send a message to the node or by timeout waiting for the next message or acknowledgment from the node.

The logic of the transaction manager (TM) is specified using a finite state machine representation. Depending on a transaction's current state and the type of input message received on behalf of the transaction, it may make a transition to another state or stay in the same state, and it may send messages to a DM server, other TM servers, or the user transaction. The implementation of the TM module is based on this finite state machine representation. This approach proved to be of great value in the debugging and modification of the system. A complete description of the state transition tables can be found in [JENQ86a].

5.2 Commit Protocol

The conventional centralized two-phase commit protocol [GRAY79] has been implemented in CARAT. For each transaction, the Coordinator TM is the the controller. The coordinator communicates with the participating slave TM's to reach unanimous agreement on the outcome (commit, or abort and rollback) of the transaction. During execution of the commit protocol, log records with recovery information are written to the stable storage so that the effects of the transactions can be correctly recovered from system failures in which the volatile memory is lost.

Performance improvements to the basic two-phase commit protocol can be achieved by reducing the number of inter-node messages and the number of log writes in the distributed environment. In addition, since read-only transactions are likely to be dominant in most application environments, it is desirable to optimize the protocol for read-only transactions. Two more efficient commit protocols, called Presumed Abort 2PC and Presumed Commit 2PC, have been proposed and implemented in IBM system R* [MOHA83]. Two-phase commit protocols are vulnerable to coordinator failures and to network partitions that isolate the coordinator. We plan to study additional protocols in the future.

5.3 Failure Handling

Following the work of Skeen and Stonebraker [SKEE81], we classify the failure handling mechanisms into two protocols: the *termination protocol* and the *recovery protocol.* The *termination protocol* is invoked at failure detection time to guarantee transaction atomicity. It attempts to terminate consistently (commit or abort, depending on the transaction's current state) all affected transactions at all the participating nodes.

If a failure is detected before the commit decision is made, i.e., while the transaction is in the *active* or *collecting* state, the TM coordinator sends an ABORT messages to the local DM server process and all the participating Slave TM servers. The slave TM server then forwards an ABORT message to the participating DM server to roll back the transaction.

However, if a node crash is detected by the Coordinator TM while a transaction is in the *committing* state, which corresponds to the second phase of the two-phase commit protocol, then the transaction will be left in the *committing* state until the slave node recovers and informs the coordinator it is functioning again. Then the commit message will be forwarded to the slave.

For communication failures, the underlying DECnet virtual circuit communication layer will first try to send messages through an alternate communication path. If one exists, the communicating processes will be allowed to continue; otherwise, the Coordinator TM follows the protocol for a failed slave.

If a Slave TM server detects that the Coordinator TM server for one of its transactions has failed, it will either abort the transaction or wait for the coordinator to recover. The decision depends on the local state of the transaction stored in the slave table. If the local state is *prepared*, then it must remain in the *prepared* state until the coordinator node recovers and sends a COMMIT or ABORT message. However, if the slave is not *prepared*, the transaction is immediately aborted.

A node which has failed must execute a *recovery protocol* before it resumes communication with other nodes. The major functions of the recovery protocol are to restart the system processes, the TM and DM's in the case of CARAT, and to re-establish consistent transaction states for all transactions affected by the failure. The information in the system log is sufficient to re-establish a consistent state and it is assumed to survive the failure.

We have taken a simple approach in the current implementation of CARAT. All transactions that were active at the time of the node crash but not in the second phase of the two-phase commit, i.e., not in the *committing* state in the coordinator table or the *prepared* state in the slave table, are rolled back.

6 Data Management and Concurrency Control

The DM servers manipulate the database on behalf of user application processes. The implementation of the DM servers can be partitioned into five major modules: **Request Processor**, **DML Processor**, **Buffer Manager**, **Concurrency Control**, and **File IO**.

The **Request Processor** module is the front-end for the DM server. It processes the transaction requests received in DOSTEP messages from the Transaction Manager. It contains a set of request-handling routines that map the request number and input parameters into a sequence of DML statements to be executed by the DML Manager. Each request from the TM server is dispatched to a corresponding request-handling routine. In our current implementation, the request handlers are statically defined and linked into the Request Processor module.

The **DML Processor** module supports database DML routines. We are currently using a simple CODASYL database processor because we have access to the source code and it is easy to modify those parts that impact our transaction processing experiments. Other data models could be supported, but we believe that the concurrency control and recovery issues can be studied independently of the data model. The DML Processor routines are called to execute CODASYL DML statements using currency, set information and data definitions specified in the database schemas. A simple database protection mechanism is provided by checking passwords contained in the DBOPEN statements before opening the database files for access. Currently, our distributed database is just a collection of independent database fragments, one at each node. But other structures, including replication, can be supported by our architecture.

The DML Processor calls routines in the **Buffer Manager** module to fetch the database pages it requires. To reduce disk IO, the Buffer Manager manages a private buffer, i.e., private to each DM server, for high traffic data pages. A least-recently-used (LRU) strategy is employed for buffer page replacement. The size and management of the buffer can be adjusted to accommodate different concurrency control and journaling strategies. For the lock based concurrency control scheme, a fixed size private buffer is managed by each DM server. Modified pages are written back to the on-disk database when the buffer slots are re-assigned to other data pages. The Buffer. Manager is also called to flush the modified data pages during the first phase of the two-phase commit execution. For the optimistic concurrency control approach, the size of the private buffers are extended to accommodate all modified pages, since no updates to the on-disk database is allowed before a transaction is committed. A global buffer mechanism, in which one or more shared buffers are used to further reduce the disk IO traffic for concurrent transactions will be considered for future implementation.

Low level IO routines for file operations are contained in the **File-IO** module. This module supports both synchronous and asynchronous write to non-volatile disk storage. Asynchronous write is used for database read/write operations while synchronous write, also called force-write, is primarily used for journaling operations to ensure that the log records are safely stored on disks before a process can continue executing other operations.

The **Concurrency Control** module contains most of the routines for local and distributed concurrency control and deadlock detection. Several mechanisms have been implemented and tested. A two-phase locking protocol with distributed deadlock detection [ESWA76, CHAND83] and a distributed version of Kung's and Robinson's optimistic concurrency control [KUNG81] are described in the next two subsections.

6.1 Two-Phase Locking with Distributed Deadlock Detection

Two-phase locking (2PL) in CARAT is supported by a **Lock Manager** that uses a hashing technique to manage a lock table containing Lock-Grant queues and Resource-Wait queues. There is one lock table at each node. It is used to record the locks on resources at the node. Five lock modes are supported [GRAY79]: Concurrent Read (CR), Concurrent Write (CW), Protected Read (PR), Protected Write (PW), and Exclusive (EX). A lock request is queued for a resource if the requested lock mode is not compatible with the locks held by other transactions or with other queued lock requests for the resource. The Buffer Manager issues a read or write lock request before reading a page into its buffer. All locks are held until a transaction has committed or rolled back.

Both local deadlocks and global deadlocks are resolved by detection. Local deadlocks are detected by searching cycles in the Transaction-Wait-For-Graph that is encoded in a two dimensional array. One approach to distributed deadlock detection that is implemented in CARAT is based on the *probe algorithm* proposed by Chandy and Misra [CHND82]. In this case, the TM server at each node acts as a controller for distributed deadlock detection. A PROBE message is an ordered pair (initiator, receiver), where the initiator denotes the transaction that initiates the probe computation and the receiver denotes the transaction which is in the dependent set of the initiator.

Our implementation of distributed deadlock detection using probes will be described using the following notation:

- TM(n) denotes the TM server and DM(Ti,n) the DM server for transaction Ti at node n.
- A transaction Ti is said to be in the Remote-Wait state, i.e., communication wait state, at node n if the DM(Ti,n) server for Ti is waiting for a request or response from a remote node. The remote node will be denoted by RWN(Ti,n).
- At each node the dependencies of transactions, e.g., Ti on Tj, are registered in an array DEPENDENT(Ti,Tj) in order to filter out redundant PROBE messages.

DM(Ti,n) executes the following protocol:

```
If DM(Ti,n) is blocked on a lock request then
    If Ti is locally dependent on itself then
        Declare Ti as the deadlock victim
    else
        If (Ti is dependent on Tj) and (Tj at node n is in
            Remote-Wait state) then
            Send an INIT-PROBE(Ti,Tj) message to TM(n)
        endif
        Idle, i.e., wait for lock release
    endif
endif
```

On receiving an INIT-PROBE(Ti,Tj) message from DM(Ti,n), TM(n) sends a PROBE(Ti,Tj) message to TM(m), where m is the remote node from which Tj at node n is waiting for a request or response to an request, i.e., RWN(Tj,n).

TM(m), on receiving a PROBE(Ti,Tj) message from TM(n) executes the following protocol:

```
If (Tj is idle) and (DEPENDENT(Ti,Tj) = FALSE) then
    DEPENDENT(Ti,Tj) := TRUE
    If (Ti = Tj) or (Tj is locally dependent on Ti) then
        Declare Ti as the deadlock victim
    else
        If Tj is in Remote-Wait state at node m then
            Send PROBE(Ti,Tj) to RWN(Tj,m)
        else
            For each Tk that blocks Tj and Tk is in
                Remote-Wait state at node m
                Send PROBE(Ti,Tk) message to TM at node
                RWN(Tk,m)
            endfor
        endif
    endif
else
    Ignore probe message
endif
```

To illustrate the implementation of distributed deadlock detection in CARAT, a deadlock that was detected in one of the performance experiments is shown in Figure 3. In the figure, each transaction Ti is identified by its associated timestamps q.n, where q is the local counter and n is the node number. The sequence of the events is listed below.

1. DM(7.1,1) for transaction 7.1 at node 1 is in Remote-Wait state waiting for DM(7.1,2) at node 2;
2. DM(7.1,2) for transaction 7.1 at node 2 is blocked on a lock request by DM(2.2,2) for transaction 2.2;
3. DM(6.2,2) for transaction 6.2 at node 2 is in Remote-Wait state waiting for DM(6.2,1) at node 1;
4. DM(7.2,2) for transaction 7.2 at node 2 is blocked on a lock request by DM(6.2,2) for transaction 6.2;
5. DM(6.2,1) for transaction 6.2 at node 1 is blocked on a lock request by DM(7.1,1) for transaction 7.1;
6. DM(2.2,2) for transaction 2.2 at node 2 is blocked by on a lock request by DM(7.2,2) for transaction 7.2. Since transaction 2.2 is dependent on transaction 6.2 and 6.2 is in Remote-Wait state, an INIT-PROBE(2.2,6.2) message is sent to TM(2); A
7. On receiving INIT-PROBE(2.2,6.2) from DM(2.2,2), TM(2) sends PROBE(2.2,6.2) to TM(1);
8. On receiving PROBE(2.2,6.2), TM(1) executes the probe protocol and sends PROBE(2.2,7.1) to TM(2);
9. On receiving PROBE(2.2,7.1), TM(2) executes the probe protocol. Since 7.1 is locally dependent on 2.2, transaction 2.2 is declared the deadlock victim.

6.2 Optimistic Concurrency Control

In the optimistic approach (OPT)[KUNG81], a transaction consists of *three phases*: a *read phase*, a *validation phase*, and possibly, a *write phase*. Our implementation of the optimistic approach is based on the use of two counters at each node: 1) NVN, the node validation number, and 2) NWN, the node write number. At the beginning of the read phase at each node transaction Tj visits, Tj reads the value of NWN into START-WN(Tj). START-WN(Tj) is the highest assigned transaction write number, NWN, when the transaction starts. During the read phase, each transaction performs read and write operations by managing data in the private buffers of the DM servers and its read-set and write-set are maintained in arrays of resource names.

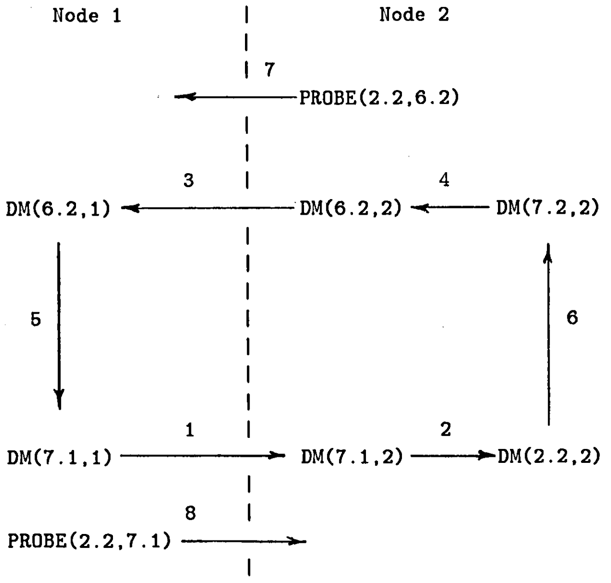


Figure 3: An Example of distributed deadlock detection in CARAT.

At the end of the read phase, the transaction is validated within a critical section. For distributed transactions, validation proceeds independently and in parallel at each node within separate critical sections. The DM(Tj,n) server at node n becomes ready to enter the validation phase for transaction Tj when it receives a PREPARE message from its TM server. If transaction Tj is validated successfully at node n, the node validation number, NVN, is incremented by one. The write-sets of validated transactions are kept in a VALIDATED-WRITE-TABLE (VWT), since they may be needed for the validation of some concurrent transactions.

Transaction Tj is validated within a critical section at node n as follows: First, the current value of NVN is assigned to VAL-VN(Tj). Then the read-set of Tj is checked against the write-set of each transaction Ti that completed writing after Tj started its read phase. Tj will fail validation if its read-set intersects with the write-set of at least one of the Ti's. As discussed in [CERI84, page 236], Tj must also be validated against other *active* transactions that have successfully validated but have not yet completed writing to the database.[5] Therefore, for each Ti such that VAL-VN(Ti) < VAL-VN(Tj) and Ti has not yet completed writing, we compare the union of the read-set and write-set of Tj with the write-set of Ti. If the sets intersect, Tj fails validation and the transaction is aborted. If Tj does not fail validation, the node validation counter NVN is incremented by one before leaving the critical section.

A transaction Tj enters the write phase only if it is an update transaction and it has been validated successfully at each node where it was active. The two-phase commit protocol is employed to ensure global consistency as described in [BHAR82]. The parts of the transaction at each node, called subtransactions, are validated separately. In the PREPARE phase, each subtransaction that succeeded in validation locally returns a positive acknowledgment, i.e., a "Yes" vote, via a PREPARE_K/OK message to its TM server. The Coordinator TM server commits the transaction only if each subtransaction of the transaction validates successfully. In order to guarantee global consistency in the face of failures, the after-image log records of successfully validated transactions are force-written to the log file before a PREPARE_K/OK message is returned by the DM server. The effects of a committing transaction can be recovered by rolling forward the database using the after-image log records. At the end of the write phase, the value of NWN is incremented by one and assigned to FINISH-WN(Tj). Note that the value of NWN is incremented and assigned to Tj only after all writes are completed. Therefore, a transaction Ti that has START-WN(Ti) ≥ FINISH-WN(Tj) is able to see all the updates produced by Tj.

7 Log Management

When a transaction is aborted, for example, because of concurrency control conflict or node failure, the effects of the transaction have to be removed by a recovery mechanism provided by the database system. One approach is to update-in-place the data objects altered by the transactions. In this case, each data object has a fixed location in non-volatile storage and each site maintains, perhaps on a storage device with different failure modes, a sequence of log records containing a before-image, an after-image, or both, of the data objects modified by the transactions to undo and redo the transactions [GRAY79, HAER83]. The log is manipulated as an append-only file on a per node basis. The log record for a data object has to be written to non-volatile storage before the data object is modified at its home location, which is the reason the mechanism is often called a write-ahead-log (WAL).

The journaling facility currently implemented in CARAT is based on this write-ahead-logging mechanism. The TM and DM servers at each node share a single system log file for before-image (BI) or after-image (AI) log records and data objects and two-phase commit protocol records. journaling can be done at various levels of data object granularity, however, only page level journaling is currently implemented.

When two-phase locking is used with before-image journaling, designated as 2PL/BI, before-image data pages are force-written to the logging disk by the DM servers before the data pages are modified in the buffers and, therefore, before the data buffers are written back to the database. The resulting before-image log can

[5] We discovered that the approach described in [CERI84, page 235-236] is not adequate to handle all cases. We introduced the node validation number, NVN, and modified the validation procedure as described here to correct the problem.

be used to roll back the database if the transaction aborts.

For the optimistic concurrency control approach with after-image journaling, designated as OPT/AI, log records of modified pages are force-written to the logging disk during the prepare phase of the two-phase commit execution, i.e., before the DM server returns a PREPARE_K message to the TM server. Note that, for the optimistic approach, all the modified data pages are buffered in a virtual memory buffer and are flushed out to the database only when the COMMIT message is received by the DM server.If the transaction does not commit, the buffer is just discarded.

To safely record the transaction state against system failures, the Coordinator TM server, as the coordinator for a distributed transaction, force-writes a COMMIT or ABORT log record to the system log at the end of the first phase of the two-phase commit, after the fate of the transaction is determined. Also, a PREPARED log record is force-written to the system log when a DM server reaches the prepared state. A PREPARED log record contains the coordinator node number to resolve the fate of the transaction at system recovery time.

The TM server at a recovering node executes a *recovery protocol* before it resumes transaction processing. First, the system log file is examined for the states of the transactions at node crash time. Active transactions are rolled back, for example, by using the before-images in the system log. For committing transactions a COMMIT message is sent to the Slave TM servers and, for each Prepared transaction, an inquiry message is sent to its Coordinator TM server for the final outcome of the transaction.

8 Catalog Management

Catalog management is introduced to support location transparency and replication transparency [LIND80]. CARAT uses a distributed catalog mechanism consisting of local catalogs managed by each of the TM server processes to provide location transparency to user transactions. In the current implementation, the mapping of file names and locations in the catalog is static, since no file migration or file deletion is allowed and each node has the complete global knowledge of the mapping. However, catalog management schemes such as those described by Lindsay [LIND80] can be implemented in the future.

A DBOPEN message carries the name of the data objects (i.e., file names) to be accessed. Upon receiving a DBOPEN message, the TM server determines the location of the data object by consulting its local catalog. If the data object resides locally and no DM server was allocated for the transaction, it allocates a DM server as the local agent for the transaction. The DM server then checks the protection and opens the file for access. For remote data objects it forwards the DBOPEN message to the remote TM server which allocates a DM server at that node for the transaction or, if a DM server has been already allocated, forward the DBOPEN message to the DM server to open the file. Note that there is at most one DM server for a transaction at each node.

9 Summary

We have described the design and implementation of CARAT, a distributed testbed for use in the performance evaluation of integrated concurrency control and recovery algorithms. The testbed approach has enabled us to "plug in" various algorithms and protocols, verify their correctness, and measure their impact on system performance. Some of the protocols that we have implemented and used in performance tests are: two-phase locking with distributed deadlock detection based on probes, a distributed version of optimistic concurrency control, before-image and after-image journaling mechanisms for transaction recovery, and a two-phase commit protocol for global consistency of distributed transactions. Other companion papers [KOHL86, JENQ86b] describe the use of CARAT in performance measurement and modeling studies. Our experience has shown that the empirical and analytical studies are synergistic. We are now in the process of gathering additional measurements for different hardware configurations (more disks and more nodes), other combinations of concurrency control and recovery algorithms, and other buffering schemes. We are also extending our modeling efforts in these areas.

Acknowledgments. A prototype version of the CARAT testbed was implemented in the Corporate Research Group at Digital Equipment Corporation by Hector Garcia-Molina, Frank Germano, Walt Kohler, and others between 1980 and 1982 . CARAT has been redesigned and reimplemented at the University of Massachusetts under the direction of Walt Kohler, Jack Stankovic, and Don Towsley with support from National Science Foundation, grants ECS-8120931, ECS8340029, and SDB-8418216.

References

[AGRA85a] R. Agrawal, M. J. Carey, and M. Livny, "Models for Studying Concurrency Control Performance: Alternatives and Implications," *ACM SIGMOD International Conference on Management of Data*, 1985, pp. 108-121.

[AGRA85b] R. Agrawal and D. J. DeWitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," *ACM Transactions on Database Systems*, Vol. 10, No. 4, December 1985, pp. 529-564.

[BERG82] H. K. Berg, "Distributed System Testbeds," *Computer*, Vol. 15, No. 10, October 1983, pp. 9-11.

[BERN81] P. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Vol. 13, No. 2, June 1981.

[BHAR82] B. Bhargava, "Resiliency Features of the Optimistic Concurrency Control Approach for Distributed Database Systems," *Second Symposium on Reliability in Distributed Software and Database Systems*, July, 1982.

[CARE84] M. Carey and M. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," *Tenth International Conference on Very Large Data Bases*, August 1984.

[CERI84] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.

[CHND82] K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", in *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Onterio, Canada, August 1982.

[ESWA76] K. P. Eswaren, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of ACM*, Vol. 19, No. 11, November, 1976.

[GALL82] B. Galler, "Concurrency Control Performance Issues," Ph.D. dissertation, University of Toronto, September 1982.

[GARC79] H. Garcia-Molina, "Performance of Update Algorithms for Replicated Data in a Distributed Database," Ph.D. Dissertation, Computer Science Department, Stanford University, 1979.

[GARC83] H. Garcia-Molina, F. Germano, Jr., and W. H. Kohler, "Architectural Overview of a Distributed Software Testbed," *Proc. Sixteenth Hawaii Intl. Conf. on System Sciences*, January 1983.

[GRAY79] J. N. Gray, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmuller, Editors, Springer - Verlag, 1979, pp. 393-481.

[HAER83] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, Vol. 15, No. 4, December 1983.

[IRAN79] K. B. Irani and H. Lin, "Queueing Network Models for Concurrent Transaction Processing in a Database System," *ACM SIGMOD International Conference on Management of Data*, 1979, pp. 134-142.

[JENQ86a] B. P. Jenq, "Performance Measurement, Modelling, and Evaluation of Integrated Concurrency Control and Recovery Algorithms in Distributed Database Systems," Ph. D. Dissertation, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, February 1986.

[JENQ86b] B. P. Jenq, W. Kohler, and D. Towsley, "A Queueing Network Model for a Distributed Database Testbed System," Technical Report CS-86-142, Department of Electrical and Computer Engineering, University of Massachusetts, June 1986.

[KOHL81] W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 149-183.

[KOHL86] W. H. Kohler and B. P. Jenq, "Performance Evaluation of Integrated Concurrency Control and Recovery Algorithms Using a Distributed Transaction Processing Testbed," *Sixth International Conference on Distributed Computer Systems*, June 1986, pp. 130-139.

[KUNG81] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transaction on Database Systems*, Vol. 6, No. 2, June 1981.

[LAMP78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July 1978, pp. 558-564.

[LIND80] B. Lindsay, "Object Naming and Catalog Management for a Distributed Database Manager," IBM Research Report, RJ2914 (36689), 1980.

[LIN81] W. K. Lin, "Performance Evaluation of Two Concurrency Control Mechanisms in a Distributed Database System," *ACM SIGMOD International Conference on Management of Data*, 1981, pp. 84-92.

[MOHA83] C. Mohan and B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," IBM Research Report, RJ 3881, 1983.

[MENA82] D. Menasce and T. Nakanishi, "Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management Systems," *Information Systems*, Vol. 7, No. 1, 1982.

[NAKA82] T. Nakanishi and D. Menasce, "Performance Evaluation of a Two-Phase Commit Based Protocol for DDBs," *ACM Principles of Database Systems*, March 1982.

[SKEE81] D. Skeen and M. Stonebraker, "A Formal Model of Crash Recovery in a Distributed System," *Proc. Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1981, pp. 129-142.

[RIES79] D. R. Ries and M. R. Stonebraker, "Locking Granularity Revisited," *ACM Transaction on Database System*, June 1979.

[STON83] M. Stonebraker, et al., "Performance Analysis of Distributed Data Base Systems," *Proceedings Third Symp. on Reliability in Distributed Software and Database Systems*, October 1983, pp. 135-138.

[THOM82] A. Thomasian, "An Iterative Solution to the Queueing Network Model of a DBMS with Dynamic Locking," *13th Computer Measurement Group Conference*, December 1982, pp. 252-261.

REQUEST II - A DISTRIBUTED DATABASE SYSTEM FOR LOCAL AREA NETWORKS

Marek Rusinkiewicz and Dimitrios Georgakopoulos

Department of Computer Science
University of Houston - University Park

ABSTRACT

The design and implementation of a distributed database management system, organized as a collection of a central database and several member databases is described. The central database contains copies of all data items that reside in member databases, and (possibly) some additional data that are not replicated. All updates are applied first to the master copy of the data item at a member database, and then propagated to the central site. Distributed concurrency control uses the Two-Phase Locking with primary copies. Protection from the loss of data consistency, due to site or network failures, is provided by a spooling mechanism which is used to keep logs of all incomplete update operations. The controlled data replication scheme used in the system simplifies the implementation of transaction processing, concurrency control and fault recovery facilities.

INTRODUCTION

For many database applications distributed systems constitute an attractive alternative by offering improved availability of data, increased reliability, and response times better than those possible in functionally equivalent centralized systems. Recent developments in the Local Area Networks (LAN) lead to a significant reduction in the communication overhead, which used to constitute a major bottleneck in computer networks. Thus the development of practical and reliable distributed database systems (DDBMS) became feasible [1]. REQUEST II is an example of a DDBMS such system designed to manage databases whose data is distributed over the multiple sites of a local area network.

The distributed configuration of the system fits naturally into hierarchical organizational structures of many modern corporations. These tree-like structures usually consist of functionally or operationally separate units with varying level of autonomy (divisions, branches, departments, etc.). The units are frequently allowed to create and maintain their own operational data. In such environments access to a single pool of information is required, while the data sources are distributed in a close geographic area. At any level of the corporation hierarchy, an efficient mechanism should be provided allowing to access not only the local data, but also the data belonging to the subordinate units.

REQUEST II provides a logically integrated view of data with distribution transparency and controlled data replication to meet the above requirements. Distribution transparency is a feature expected from modern distributed database system and allows users to interact with REQUEST II exactly as if it were not distributed. Replication of data is controlled by a static scheme which reflects the hierarchical structure of data and allows to improve response time, fault tolerance and data availability. Under the proposed replication scheme all members of a hierarchy, are provided with a local copy of data owned by their direct subordinates. This hierarchical organization applies to the data distribution and replication scheme but not to the underlying data model. In fact a REQUEST II database is a collection of (possibly replicated) relations.

Many general solutions to the problems of failure and recovery in distributed systems [2] are not directly applicable to the design of (efficient) distributed databases. The main reason for this is the number of data items involved in a transaction and the fact that the relevant data items may be determined dynamically during the transaction processing. Majority of solutions proposed for distributed database systems [3] are tuned towards long haul, low bandwidth communication networks. Since the design trade-offs are different in LANs, the solutions suitable for this type of environments would have to be developed.

CH2345-7/86/0000/1179$01.00 © 1986 IEEE

In a Local Area Network such as Ethernet or ring network, many of "classical" reliability issues such as network partitioning need not to be considered. Instead, the achievement of a high level of performance becomes more important. Furthermore, it is possible to impose some data architecture constraints to allow the use of efficient transaction processing algorithms that improve the response time and simplify fault error recovery. REQUEST II constitutes an attempt to provide an efficient solution for a LAN-based DDBMS with a reasonable set of restricting assumptions.

The remaining part of the paper is organized as follows. The next section describes the data architecture and outlines the nodal database system used by REQUEST II. The following sections introduce the techniques used by REQUEST II to solve the most difficult problems in distributed data management, namely transaction processing, concurrency control and fault tolerance. The last section describes the system implementation. The discussion concludes with some comments on further system development.

DATA ARCHITECTURE

Data distribution and replication

A distributed database under REQUEST II can be viewed as a collection of a central database and several member databases. A copy of each data item can be stored in at most two locations: the central database and one of the member databases. The central database contains copies of all data items that reside in member databases, and (possibly) some additional data that is not replicated. Each member database contains copies of its local data items, which are used as master copies for concurrency control and recovery purposes. The member databases are considered to be the owners of that data. The central database owns only the data that is not stored in any member database. No data item resides in more than one member database (Figure 1).

The configuration of the central database (CDB) and the member databases (MDBs) can be described using the concept of data objects - collections of data items constituting the contents of databases. The master copy of a data object D, will be denoted by Di. If only one copy of a data object exists, it is considered to be a master copy. Based on the previous discussion of the data distribution and replication, we may say that the objects (Di, i=1..n) are stored in the corresponding member databases (MBDi,i=1..n). The central database (CDB) contains the duplicate copies of data objects (Di', i=1,2,...n) and the non-replicated data object (D0) which is stored only in the central database.

Thus, the data distribution and replication scheme of REQUEST II satisfies the following conditions:

central database:
$CDB = \{D0, D1', D2', \ldots, Dn'\}$

member databases:
$MDBi = \{Di\},\ i = 1, \ldots, n$

replication rules:
$Di' = Di,\ i = 1, \ldots, n$
$Di \wedge Dj = \{\},\ i <> j,\ i,j = 1, \ldots, n$

where $\wedge$ stands for intersection and $\{\}$ represents the empty set.

Extension of this scheme to a clustered distributed database environment is straightforward. Each member database can be thought of as a central database of an underlying (lower) level in a hierarchy, thus allowing to implement a hierarchically structured database collection. A REQUEST II cluster can be defined as a collection of database network sites (nodes) whose data items follow the above model of data distribution and replication (Figure 2).

A database network node is allowed to participate in at most two clusters, i.e. a node that is a member database in one cluster can constitute a central database in another cluster. In a clustered environment a copy of an object located in the lowest-level member database is considered to be the master copy of this object.

Distributed dictionary structure

The meta-information about data objects is stored in a hierarchical system of database dictionaries (catalogs). REQUEST II uses a simple general rule for dictionary replication: copies of table dictionaries reside where a copy of the corresponding data table is located.

A node dictionary resides at each site and contains information about all active nodes in the clusters of the distributed system, in which the site participates as a member and/or a central node. The entry for each node contains the information necessary to access the

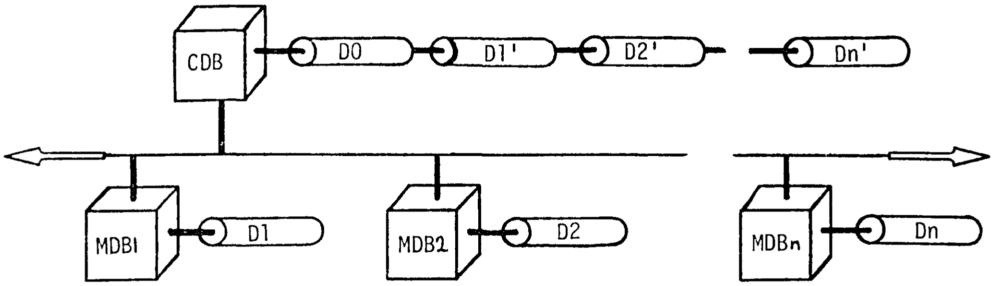


Figure 1. Data Architecture for a Two-level Hierarchy

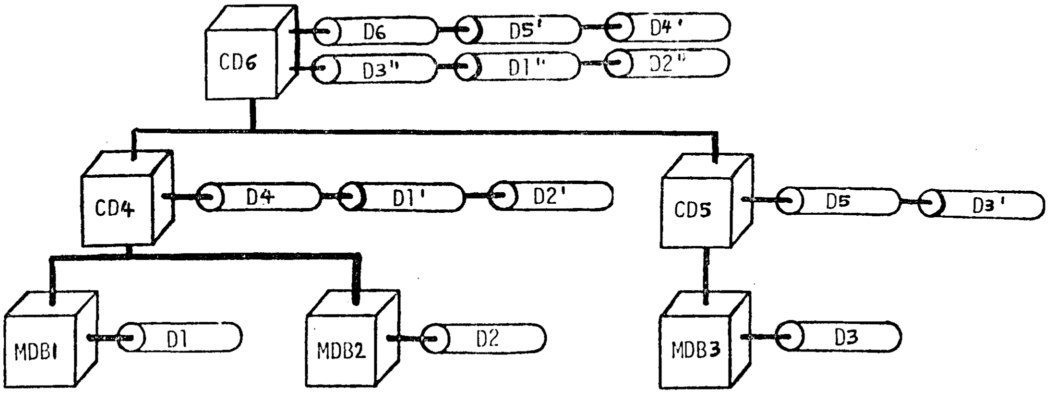


Figure 2. Multi-level Data Distribution and Replication Scheme

data stored at this node, including the logical node ID and the access information used by REQUEST II to perform remote operations.

The database dictionary lists all the databases defined in the distributed system, and provides access information (addresses) for their dictionary, security and data files. The database dictionary is fully replicated, i.e. a copy of it exists at each node. Table dictionaries describe the structure of every relation in the system. In each cluster, there are at most two copies of each table dictionary located together with the corresponding data files. REQUEST II dictionaries can be manipulated only by special routines which clearly distinguish dictionary relations from data relations.

The nodal Database Management System

REQUEST [4, 5] is a relational multi-user DBMS used as a "nodal" Database Management System for both central and member databases. REQUEST supports a wide variety of user interfaces, including several non-procedural query languages such as SQL, QUEL and Query-By-Example, a report generator, a screen-oriented application development tool, a host language interface, an integrated data dictionary and an on-line HELP facility. Most user interfaces are menu driven.

A database under REQUEST stores three kinds of relations: dictionary, data and security tables. The integrated dictionary system is hierarchically structured and consists of a database dictionary and a number of table dictionaries. For each database the information is kept describing all existing tables, including the owner, the table logical record length, table cardinality, table type (i.e. the information whether the relation is a view or a base table) and the physical organization of data files. A table dictionary file contains description of all fields defined in that table including field name and length, field type (i.e. integer, real, string, money or date), maximum and minimum values, default value (if applicable), default display format, and key information. A table security file contains the list of users authorized to access the table together with the allowed operations. Security file can be modified only through a GRANT-REVOKE authorization mechanism. Data files can be internally stored and accessed either as hash files or an indexed sequential files. REQUEST access method uses multi-key extendible hashing [6] and sequential organizations with secondary (dense) indexing for data files.

The local concurrency control under REQUEST uses the Two-Phase Locking [7] with intent locks [8]. REQUEST uses only a restricted form of transactions in which locks can be imposed on linearly ordered database object classes. This restricted form of transaction allows efficient implementation of a deadlock-prevention strategy.

When the user enters and executes a query, the query language interface (UFI) calls the language processor to parse, optimize and interpret the query. Before a parse tree is created, all relation and attribute variables are validated, and the domains of all constants used in the query are checked. At this stage it is determined whether the query is local or remote.

A nodal REQUEST can be viewed as a two level abstraction consisting of user interfaces (upper level) and the run-time unit (lower level). The run-time unit is capable of serving requests independently of the request source (local user or another site in the distributed system). User interfaces generate requests to be satisfied either at the local or at a remote site. The "centralized" REQUEST has been converted to a "nodal" database management system by modifying the interface between the run-time unit and the user access requests, so it became flexible enough to support both the calls from local users interfaces and the calls from the distributed system software.

Currently the distributed environment of REQUEST II, allows basic DDL, DML and SQL data retrieval functions to be performed interactively from any site in a network-transparent manner.

TRANSACTION PROCESSING

A single execution of a Data Definition, Data Manipulation or Query command constitutes a transaction. Transactions can be classified as queries (read-only) or updates (read-write) and apply either to the dictionary tables or to both dictionary and data tables. In order to support a multi-user, multi-site environment it is necessary to schedule conflicting transactions using some concurrency control mechanism. The concept of serializability [9] is employed to assure that conflicting read and write operations are scheduled according to some serialization order. Serializability requires that whenever transactions execute concurrently, their effect must be identical to some serial execution of these transactions.

Nodal DBMSs use the "Two-Phase Locking" to resolve local conflicts, while the distributed system extends this basic locking scheme to satisfy the distributed environment requirements. Two different locking methods are employed in REQUEST II. The "Basic Method" is used for the database dictionary copies and the "Primary Copy" method is used for table dictionaries and data items. [9]

Under the "Basic method", a read-lock is set at the local copy, if it is available. Otherwise, an arbitrary copy of the data item is read-locked. For update operations, write-locks are requested from all copies of the data item and the transaction is allowed to proceed only after all of them are granted. This reader-preference solution was adopted since, the changes to the data definition scheme do not occur frequently. To eliminate the possibility of a deadlock priorities are assigned to all database dictionary copies. Deadlocks are prevented by requiring all transactions to request their locks on copies of the database dictionary, in the a priori assigned order. Under the "Primary Copy" method both the read-locks and write-locks are imposed only at a designated master (primary) copy.

REQUEST II requires all transactions to lock distributed database objects in the following order:
- database,
- table (optional),
- data record (optional),
- table header (optional).

This linear ordering defines a hierarchy of all classes of lockable database objects. During the transaction execution a tree of locked objects of different granularity is produced. Deadlock avoidance techniques have been used, to carefully analyze the lock/unlock sequences in all transactions types and assure that deadlocks can not occur. Deadlock prevention in REQUEST II relays on two assumptions: the linear ordering of all classes of lockable objects in the restricted REQUEST II transactions and intent locking of these objects. Intention mode is used to lock all the ancestors of a node which is to be locked in shared or exclusive mode.

Transaction processing algorithms depend on the availability of data (data replication). For each transaction a transaction processing plan is determined by a Transaction Manager. Transaction Managers locate the remote and local copies of database items and manipulate them by calling the corresponding (local or remote) Data Managers. The distributed system obtains the information about data location from its dictionary system and executes the transaction as follows.

1. Updates: All updates are performed first at the site that stores the primary copy of a data item. Then, if there are other copies to be updated, the the primary copy site assumes the responsibility for propagating updates to the other sites involved. The primary copy of the updated object remains locked until all copies are updated.

2. Dictionary Queries: If the dictionary is available at the local site it is read locally. Otherwise read requests are forwarded to the site that holds the primary copy of the dictionary. Database dictionary queries need to lock only the local copy of the database dictionary, while for table queries the primary copy of the table dictionary must be locked.

3. Data Queries: If all data tables (and therefore dictionaries) required to evaluate a query are available locally then the query is processed. Otherwise, the query is submited to the central site, evaluated there and the response table is sent back to the originating site. Data queries lock the primary copies of the data tables.

Processing of data queries is facilitated by the data replication scheme used in REQUEST II, under which no decomposition of compound queries is required. This approach may have one drawback, namely little use is made of potential for parallel processing within a single transaction (although different transactions can be processed in parallel). Parallel processing requires the decomposition of a query into sub-queries, to be evaluated in parallel at different sites. The query evaluation cost consists of the I/O cost, the CPU cost and the communication cost. Studies of local area networks have shown that communication overhead is reasonably low and therefore the communication time does not necessarily dominate the cost of a query evaluation. This observation is not true in packet switching networks where the communication overhead is a dominant factor [10, 11].

As an example illustrating the use of our locking protocols, let us consider the processing of an interactive transaction performing a data record update. In the discussion below, S stands for shared lock, X means exclusive lock and IX means intent exclusive lock.

Data record update:

```
Get database and table names;
LOCK(IX) the local copy of the database;
Search local database dictionary for
database and table names;
If database or table name are not found
then error;
If the local site is not the primary copy
site of the data then
   SUBMIT update request to the primary
   copy site;
   UNLOCK(IX) the local database copy;
   Wait for a response;
else {primary copy site}
   LOCK(IX) the primary (local) copy of
   the table;
   Get the new data value(s);
   LOCK(X) the primary (local) copy of the
   data record to be updated;
   Perform the update on the primary
   (local) copy and propagate the update
   to the other (remote) copy;
   UNLOCK(X) the primary copy of the
   updated data record;
   UNLOCK(IX) the primary copy of table;
   UNLOCK(IX) the local database copy;
```

Other classes of transactions, i.e. data queries, dictionary queries, table field updates, table updates, database updates and authorization checks are performed in a similar way.

FAULT TOLERANCE

One of the frequently mentioned benefits of distributed database systems is the improved reliability and data availability. However, in order to achieve this benefit, the system must be able to operate correctly (preserve database consistency) even in the presence of failures. There are several types of failures that could occur in a distributed system. Two main categories are communication network failures and site failures. Undetected failures may severely damage the integrity of the system. Thus a failure detection mechanism must be provided to notify the system when a failure occurs. Failure detection and notification mechanisms usually reside in the communication subsystem. Distributed Database Systems logically belong to the (ISO) application layer of communication subsystems [12] and they rely on the network error detection mechanisms [13].

If a site fails during an update, the distributed system may be left in an inconsistent state. To protect distributed databases from this type of inconsistencies, a log of an operation must be recorded in a stable storage, before the execution of any update operation. In REQUEST II a spooling mechanism is used to maintain logs. A spooler is a process with access to the secondary storage, that serves as a first-in, first-out message queue. Any message sent to a node is first delivered to a spooler. The spooler secondary storage is managed using conventional DBMS reliability techniques (i.e. locking and message invalidation) to guarantee the integrity of these messages. Log entries are uniquely identified within the site by (local) timestamps. Other sites can refer to a specific log entry by providing its timestamp to the spooler where the log is kept, e.g. to commit an operation. Log files can be inspected by the recovery procedures to Undo/Redo any update that left the database inconsistent.

A Two-Phase Commitment Protocol

The commit protocol employed by REQUEST II to help maintain the consistency of the distributed database is a modification of the "presumed abort" two-phase commit protocol [14]. The protocol ensures that an update is either committed at all sites or not committed at all, even if a failure occurs during its execution. A transaction is aborted and the the user is informed if during an update operation the site holding the primary copy of a data object fails, before the commit message of the transaction is accepted for processing at that site. This assumption implies that sites holding primary copies of database objects must be operational at least up to the point where the transaction is accepted (marked as committed) but they can fail anytime before, during or after the effects of the update operation are actually installed in their local databases.

The REQUEST II two-phase commit protocol uses one process called the coordinator, which is connected to the user application and a set of subordinate processes running in cohort sites. The coordinator is responsible for preparing and committing an update in the site where the primary copy of the data item is stored (primary cohort site). When this task is completed the coordinator "forgets" the operation and informs the user that a transaction has been committed, even though the actual database updates are not yet installed in all copies. Thus, the response time as perceived by the user is significantly reduced. The primary cohort site becomes responsible for performing the update on the primary copy and propagating the update to the remaining copies of the data item.

Algorithm: Two-phase Commit

Step 1: When a transaction at some site of the system requests an update operation that involves remote sites, it automatically becomes a coordinator for this transaction. The coordinator constructs a message composed of the site ID and the remote DM operations to be performed. Then a PREWRITE(site ID, DM operations) message is sent by the coordinator to the spooler of the cohort site which stores the primary copy of the data item (primary cohort site). If the primary copy of the data item is stored in the local site then the primary cohort site is the coordinator's site.

Step 2: When a PREWRITE message arrives at the primary cohort site, the spooler receives the message and constructs a Log_uid based on its local clock value. This Log_uid uniquely identifies the log at the primary cohort site. Then, the spooler creates a log entry consisting of the Log_uid, the DM operations, the number of non updated copies of the data item to be updated (target count), and the set of nodes that store the yet to be updated copies of that item (target set). The target count is initially equal to the total number of copies for the data item. Its purpose is to serve as a commit flag. If its value is less than the total number of copies of the data item to be updated, then the transaction is considered committed else it is uncommitted. The target set initially contains all nodes that store copies of that item. It is used as a checkpoint for the transaction progress at the various nodes involved in its execution. Finally, the primary cohort site locks properly the spooler storage and inserts the log entry. At this point a PREWRITE_ACK(Log_uid) message is sent back to the coordinator.

Step 3: If the coordinator does not receive the PREWRITE_ACK message from the primary cohort site (the site is down), then it aborts the transaction and informs the user. Otherwise it sends a COMMIT(Log_uid) message to the primary cohort site.

Step 4: When the COMMIT(Log_uid) message, is received by the primary cohort site, its spooler locates the log entry using the Log_uid as an index, and extracts the DM operations to be performed from the log. The target count is decremented by one and simultaneously a COMMIT_ACCEPTED_ACK message is sent back to the coordinator (i.e. transaction is accepted for completion). The above two actions are performed in a single indivisible step (i.e. similar to the "test and set" instruction found in all types of computers). Finally, the DM operations are forwarded to the local DM for execution, and the target set is modified by removing the local site ID. The purpose of the COMMIT_ACCEPTED_ACK message is to inform the user site that the transaction will be eventually completed under the control of the primary cohort site.

Step 5: When the coordinator receives the COMMIT_ACCEPTED_ACK message it assumes that the update has been completed and "forgets" about it. The primary cohort site assumes the responsibility for propagating the updates to the remaining copies of the data item. This task is accomplished by determining the IDs of the cohort sites which keep non updated copies of the data item (i.e. target set). For every such site, the target count is decremented, and an update message is sent to the corresponding remote DM. When an acknowledgement is received the target set is modified by removing the remote site ID. These updates are performed in accordance with a one-phase commit protocol. When the target set becomes empty, the INVALIDATE(Log_uid) operation is performed to remove the log entry from the spooler storage.

The above process can be simplified taking into account data replication and the location of the primary copy in relation to the user's site, resulting in reduction of the processing and communication overhead.

Crash recovery

When a site resumes running and prior to the execution of any transaction, the recovery routine accesses the spooler storage (local and remote) and if possible, finishes all incomplete updates violating the consistency of the database in the recovering node. Crash Recovery routine is essentially a ReDo procedure that examines the log files and and performs all the necessary updates. The requirement for designing a REDO recovery process is to implement most recovery operations in such a way that may be repeated without producing any additional inconsistency. This can be achieved by keeping additional information in the logs and/or allowing the recovery process to perform read-only DM operations on the distributed database. The latter implies the use of monolithic recovery locks which are necessary to preserve consistency and will not be discussed here.

In the first phase the recovery process examines its local spooler storage and all uncommitted log entries (i.e. entries where the target count of a data item is equal to the total number of copies of that item) are discarded.

In the second phase, the recovery process completes the committed transactions by executing the DM operations which are recorded in the log entries (i.e. entries where the target set is not empty). The effects of those DM operations are directed only to the nodes which are found in the target set of each log. Before the remote execution of a DM operation the target count for that data item is decremented. After the DM operation is executed, the target set is modified by removing the ID of the node where the effects of the DM operations were installed. Installing the effects of an incomplete transaction is performed in a pre-defined order, on a node by node basis. At each node, the recovery DM operations are executed in a sequential order for each log entry. This sequential order is reflected in the structure of the logs, where log entries are associated with timestamps. Therefore, later events follow the earlier ones when the log is examined sequentially.

If a log is stored locally (i.e. the recovering node is a primary node) its effects are installed in all nodes found in the target set of that log entry. For every log entry its target set contains only the sites where the recorded DM operations are not yet installed. Thus, all copies of data items whose primary site is recovering will be eventually updated and all related inconsistencies will be resolved. Since we assumed that no updates to a data item are allowed if its primary node is not available, the log at the primary node must contain all unfinished DM operations related to this data item.

Whenever the recovering node is found in the target set of a log entry at a remote spooler, the DM operation(s) of this log entry are applied only to the recovering node. The purpose of this type of recovery is to remove inconsistencies by "informing" the recovering node about changes that occurred to database objects having their primary copies at different sites while the recovering site was not accessible. In addition, inconsistencies in the distributed database dictionary are resolved this way (e.g. if a table whose primary node is different than the recovering node has been deleted, the database dictionary copy at the recovering node must be modified to reflect this change).

In the final phase all invalid logs (i.e. entries for which the target set empty) are removed.

If a node failure occurs during a crash recovery, the recovery process need to be repeated. Most recovery operations are repeatable (e.g. if the current log that is examined by the recovery process indicates an "update data record" DM operation, the recovery process is designed to verify the existence of a valid version of the record to be updated before it actually attempts the update). Therefore, a failure of the recovery process cannot cause any additional inconsistency.

The proposed spooler structure scheme requires that the log files entries that reside at different nodes correspond to independent database objects, namely, primary copies which by definition reside in at most one site. Thus, in the case of a multiple site failure, the processing of the log files located at different member sites by different recovery processes could be performed in any order.

The proposed protocol provides complete protection against single site failures and many multi-site failures. This is achieved at the expense of limiting somewhat data availability.

SYSTEM IMPLEMENTATION

Inter-site process communication alternatives

When a user requests an operation on remote data, the request is passed to and executed at the site where the data is stored. Several alternative organizations can be considered for implementing the execution of remote operation requests.

1. One solution is to provide a single service process (server) that accepts and services requests from remote processes (i.e instances of REQUEST II on remote machines). A server of this type must multiplex itself among multiple client requests and maintain transaction state information from one transaction to another. The problem with such organization of a server is the throughput degradation due to delays associated with page faults and database I/O operations.

2. Another solution is to create a separate process to service each incoming request. However, creating and initializing a process is a time consuming operation which adds overhead to the execution of a request. A more serious drawback is that such organization does not preserve the notion of a transaction, thus all requests belonging to a transaction must be associated by some mechanism that ties them together. Data access authentication is a problem too. Each request must be individually authenticated, which is a potentially expensive operation [15].

3. Another solution is to create a process for each user on his first request and retain that process for subsequent use for the duration of the session. This reduces the process set-up cost necessary to service multiple requests for data. Since a process belongs to a single remote user and since all requests that are part of a transaction originate from the same user, there is no need to transmit the user ID and perform user authentication every time.

4. Another possibility is to initialize the server as a single process. The server process handles all incoming transaction requests until its performance degrades. At this point a new instance of the server is spawned to share the load with the "old" instance of the server. The spawning of new instances of server can be controlled statically, by setting an upper limit to the number of transactions that can be serviced concurrently by a single server instance, or dynamically, by monitoring the server load [16].

In our implementation the system sites communicate using the DECNET transparent communication facility. The client process (Transaction Manager) and the server process (Data Manager or Spooler) are connected by a virtual circuit (logical channel). When a process requests an activity at another site for the first time, a virtual circuit between the sites is established. This is accomplished by remotely executing a command procedure which activates the REQUEST II programs. The virtual circuit and the server processes are retained for the duration of the user session. The processes can now communicate with each other by performing remote procedure calls. A process constructs and sends a message describing the function to be performed and its parameters, and then waits for replay. Remote procedure calls are used to perform most of the inter-site operations except for transferring large amounts of data which is done by explicitly invoking the DECNET file transfer facilities.

In detecting network or node failures REQUEST II relies upon this send/receive interface to report if a process or a processor at the other end has failed. Communication failures are reported to the processes on both ends of the virtual circuit. The DECNET provides the necessary failure detection and reporting capabilities.

A pilot implementation of the system runs on five VAX-11 computers using the DECNET facilities and the VMS mailbox mechanism for inter-nodal communication [17]. In our further work, we plan to compare the efficiency and the overhead of the four outlined organizations of the inter-site process communication, and evaluate their suitability to implement a distributed database systems using the testbed approach. Another direction of our work, is to adapt the database management system to an environment in which several Local Area Network clusters are connected through gateways. In particular, we are investigating a distributed database system which can operate in a heterogeneous network and OS environment consisting of DECNET/VMS and UNIX/3BNET.

REFERENCES

[1] Stonebraker M. and E. Neuhold, "A Distributed Data Base Version of INGRES", Second Berkeley Workshop on Distributed Data Management and Computer Networks, 1977.

[2] Barbara Liskov and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM Transactions on Programming Languages and Systems, (Vol. 5, No 3), July 1983.

[3] Hammer, M. and D. Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM TODS, December 1981.

[4] B. Czejdo and M. Rusinkiewicz, "REQUEST: a Testbed Relational Database Management System for Instructional and Research Purposes", AFIPS Conference Proceedings,(Volume 53), July 1984, The NCC 84.

[5] Rusinkiewicz,M.,and B.Czejdo, "Query Transformation in Heterogeneous Distributed Database Systems", Proceedings of 5-th International Conference on Distributed Computing Systems, Denver, May 1985.

[6] Kelley, K. and M.Rusinkiewicz, "Implementation of Multi-key Extendible Hashing as an Access Method for a Relational DBMS", to appear in Proceedings of the Second International Conference on Data Engineering, Los Angeles, February 1986.

[7] Eswaran K., Gray J., Lorie R. and Traiger I., "The Notion of Consistency and Predicate Locks in a Data Base System", Communication of ACM, (Vol 19, No 11), November 1976.

[8] Date,C.J., An Introduction to Database Systems Volume II, Addison-Wesley, 1983.

[9] Bernstein P., and Goodman N., "Concurrency Control in Distributed Databases", ACM Computing Surveys, (Volume 13, No 2), June 1981.

[10] Epstein R., Stonebraker M., and Wong E., "Distributed Query Processing in a Relational Data Base System", Proceedings of the ACM SIGMOD International Conference on Management of Data, June 1978.

[11] Agrawal,R., M.Carey, and M. Livny, "Models for Studying Concurrency Control Performance: Alternatives and Implications", Proceedings of the ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 1985.

[12] Zimmerman,H., "OSI Reference Model - The ISO Model of Architecture for Open System Interconnection", IEEE Trans. Commun., (Vol. 23, No 2), February 1980.

[13] Thomas A. Joseph and Kenneth P. Birman, "Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems", ACM Transactions on Computer Systems, (Vol. 4, No 1), February 1986.

[14] C.Mohan, B.Lindsay, "Efficient Commit Protocols for a Tree of Processes Model of Distributed Transactions", ACM Proceedings of the 2nd SIGACT/SIGOPS Symposium on Principles of Distributed Computing, August 1983.

[15] Lindsay B., Hass L., Mohan C., Wilms P. and Yost R., " Computation and Communication in R*: A Distributed Data Manager", ACM Transactions on Computer Systems", (Vol 2, No 1), February 1984.

[16] Ousterhout J., Scelza D. and Sindhu P., "MEDUSA: An experiment in Distributed Operating System Structure", Communication of ACM, (Vol 23, No 2), February 1980.

[17] Digital Equipment Corporation, "Guide to Networking on VAX/VMS" VAX/VMS Version 4.0 Manual, September 1984.

A PROTOCOL FOR FAILURE AND RECOVERY DETECTION TO SUPPORT PARTITIONED OPERATION IN DISTRIBUTED DATABASE SYSTEMS

Jung K. Kim and Geneva G. Belford
Department of Computer Science
University of Illinois at Urbana-Champaign

ABSTRACT

Data replication in distributed database systems is designed to achieve high reliability and availability. In case of network partitioning, the mutual consistency of data may become doubtful unless the transaction processing mechanism puts appropriate limits on its operation. To achieve high availability, transactions must be processed despite failures. An optimistic approach allows transactions to be executed in each partition and the database is reconciled when the partitions are merged. To coordinate the partition merge process correctly, the system must know the partitioning and merging points. This paper presents a new protocol for failure and recovery detection, which correctly supports partitioned operation and subsequent merging.

1. Introduction

To achieve high reliability in a distributed database system, component failures must be tolerated. Since unexpected failures may occur at any time, the transaction processing mechanism must be robust against them. In a distributed database system, concurrent accesses to a data object must be synchronized and recovery from failure has to preserve the consistency and integrity of the database. If data is replicated, an update to a data object must be *atomic*; i.e., it should be performed on all copies or none. Also, transactions should not be delayed indefinitely in the face of failure. It is therefore important that failures and recoveries are detected accurately and efficiently.

When failure occurs it is possible that one group of sites cannot communicate with another group. This type of failure is called *network partitioning*. If network partitioning occurs, the mutual consistency of data items may become doubtful unless the transaction processing mechanism appropriately limits its operation. When data redundancy is used in order to increase the availability of data, transactions must be processed despite failure. Several approaches to operation in the face of network partitioning have been proposed. One approach allows all transactions to be processed, and later, when partitions are merged, any data inconsistency is resolved. (See [1], [12], [9], etc.) This approach is called *optimistic*, since it is efficient only when there are few inconsistencies to be resolved after the partitions are merged.

In order to support partitioned operation, a mechanism to detect changes in the status of the network is necessary. Recently, a number of protocols for failure and recovery detection have been proposed (e.g., [5], [10], [11], [4], [6]). Most of these protocols are based upon periodic checking of the system status, which requires considerable overhead. Even when the system is lightly loaded or idle, the status of the network is periodically checked. This paper presents a new failure and recovery detection algorithm, which may be used to support optimistic partitioned operation. In this algorithm, each site monitors other sites only when necessary. Specifically, if a transaction includes operations to be carried out at remote sites, its home site then checks the status of the other sites. A *Status Monitor* (SM) resides in each site to detect status changes (failure or recovery) of other sites and to coordinate the construction of a consistent view of the membership of a partition. In addition, our protocol handles the problem of synchronizing multiple failure/recovery detection in a particularly robust way. This protocol consists of two subprotocols – the Failure and Recovery Detection Protocol (FRDP) and the Status Change Protocol (SCP).

In section 2, background material, basic assumptions about communication facilities, and the system model are presented. The transaction management mechanism is briefly described in section 3. Section 4 presents the detailed description of the FRDP and the SCP. Section 5 discusses the feasibility of our protocols and gives arguments that they work correctly. Finally, our conclusions are in section 6.

2. Background

A *transaction* is defined here as a process that manages a sequence of read and write actions to be executed atomically for one application. A maximal subset of sites which can communicate with each other forms a *partition group* (or *partition*, for short). In the optimistic approach, while the network is partitioned, the system processes all transactions, but commitment becomes temporary until partitions are merged, even though actual copies of data items are updated. This means that some transactions may have to be rolled back after partitions are merged. Logical merging of the transactions processed in different partitions must be done in such a way that the resulting transaction schedule is serializable. Serializability means that concurrent transactions are executed in a manner equivalent to a serial execution [8]. To enforce serializability, a transaction that reads a copy of a data item in one partition must logically precede a transaction that updates the same data item in another partition. A *serializability conflict* is defined as a situation that occurs when a transaction reads or updates the copy of a data item in one partition and

CH2345-7/86/0000/1189$01.00 © 1986 IEEE

another transaction updates the copy in another partition. Thus, if an update to the same data item occurs in more than one partition, there is a serializability conflict.

In Davidson's protocol [1] for recovery after partitions are merged, possible inconsistencies among data item values are detected by using a precedence graph, which checks for the serializability of the executions of transactions across partitions. A concurrency control mechanism is assumed to guarantee the serializability of the execution of transactions in each partition. A transaction is assumed to read a data item before updating it. This assumption facilitates serializability conflict detection in the precedence graph, since the graph can be reduced considerably. It is also assumed that each partition has a unique coordinator. When partitions are merged, the coordinator in each partition is notified and local transaction executions in each partition are stopped. The coordinator then derives the total ordering of the transaction execution history in its partition. Then a global precedence graph is generated from these histories. Conflicting transactions are detected as a cycle in the precedence graph. If there is such a cycle, conflicting transactions are rolled back until the cycle is eliminated. Then the remaining transactions are executed at sites that have not yet executed them yielding a consistent database. A new coordinator of the merged partition is then elected.

To coordinate such a partition merge process, the system must know when the partitioning occurred and when the merge process is initiated. (By "when" we mean not clock time, but the point within the execution schedule of transactions.) Thus, some surveillance mechanism which sees the status changes is necessary. The only way to detect a status change is to send messages and use timeouts, because a global snapshot of the network status is not possible. A site may periodically send messages to check for network status changes or the status may be checked only when necessary. If a surveillance mechanism which operates only as needed is adequate to support partitioned operation, then we can save on overhead. Essentially, we only need to know about status changes that can affect the consistency of the database. If the system is partitioned but the transactions can be processed without any problem (i.e., there is no conflict), we do not need to know that partitioning has occurred. Our protocol provides only those network status change points that are needed to restore the database to a consistent state after the partitions are merged.

Site failures and communication link failures are considered in our protocol. A site failure occurs when a site in the network goes down, and a communication link failure occurs due to a transmission medium fault. Generally, it is not possible to distinguish between a site crash and a partitioning. Therefore, when it is not possible to communicate with a site, network partitioning is assumed. When a site failure occurs, it is assumed to be *pure*, meaning that the processor simply stops running and no garbage messages are produced.

Since our protocol does not detect all network status changes, it is necessary to define a *view* of a partition. A *logical partition* view (or just *view*) at a site is defined as being the subset of sites with which communication is thought to be possible [7]. A site may think that it cannot communicate with certain other sites even if they are available. Similarly, a site may be unavailable when it is thought to be available. But, if communication is needed with such a site, our protocol will detect the status change. Thus, a site's view of the network may be updated only when knowing the status of the network is necessary. A site S_i is considered to be UP at site S_j if S_i and S_j are in the same logical partition in S_j's view. Otherwise, it is considered to be DOWN. A site status table is maintained by each site to maintain its current view.

In order to detect conflicts and to resolve them after partitions are merged, each partition is assumed to have a unique coordinator. Garcia-Molina discussed elections of a coordinator in a distributed system [3]. To avoid the question of electing a coordinator in a partition, an ordering of the sites is assumed. For example, the sites may be numbered 1,...n, and we assume $S_1 < S_2 <...< S_n$. The site which is lowest in order becomes the coordinator when a new partition is created. This ordering need not be related to the physical topology of the network.

Data items are assumed to be partially replicated, so that some sites may have copies of a data item while other sites may not. It is assumed that transactions follow the 2–*Phase locking* protocol [2]; also that a write operation requires that all copies be locked and a read operation requires that the copy read be locked. This guarantees that serializability within a partition is not violated. Furthermore, the following are assumed.

- All messages are delivered to their destinations in the order in which they are sent.
- Message delivery is not guaranteed. The sender only knows of the delivery of its message by an explicit acknowledgement.
- The maximum and minimum network message transmission delay, T_{MAX} and T_{MIN}, can be defined.
- The relation 'can communicate with' is symmetric and transitive.

3. Transaction Management while the Network is Partitioned

The system is assumed to be running an optimistic algorithm such as Davidson's protocol [1]. When a transaction reads or updates data items and all copies of those items are not in the same partition, then its commitment is temporary until the sites that have the other copies join the partition. After temporary commitment, the transaction sends its execution history to the coordinator of the partition. And the coordinator sends an acknowledgement to the transaction. (This allows the transaction to recognize when the coordinator has failed or a partitioning has cut it off from the coordinator. See section 5.) The coordinator uses these transaction execution histories to produce a totally ordered transaction execution history within the partition, so that conflicting updates can be resolved when partitions are merged.

Even a *read—only* transaction may be involved in a serializability conflict, if it is processed in a partition which does not have all copies of the data items read by it. Therefore a site executing any transaction involving data items with copies in other partitions has to monitor the status of other sites that have copies of the data items used until commitment of the transaction becomes permanent.

Whenever the transaction performs an action (read or write), it references a data dictionary to get the identifiers of the sites which have copies of the data items used. Then:
– For all of those sites marked UP in the current view, the transaction sends a *watch* request to the SM. The SM then begins monitoring the sites for failure. The transaction sends a *watch–release* to the SM when it completes (temporary commitment).
– If not all of those sites are marked UP, then the transaction requests a watch for recovery of the sites marked DOWN and does not release those watch requests until its commitment becomes permanent. In this way, recovery of the sites in the other partitions is detected.

If a status change is detected by the SM, transaction processing is stopped and updating of the network view is carried out.

4. The Protocol Description

In this section, a detailed description of the proposed protocol is presented. Our protocol consists of two subprotocols – the Failure and Recovery Detection Protocol (FRDP), and the Status Change Protocol (SCP). Before presenting the FRDP and the SCP, we describe the messages and data structures needed.

4.1 Messages and Data Structures

To facilitate the operation of the FRDP and the SCP, the following messages are used.
- The site status checking message (CHECK) – This message is used to check the status of a site during normal FRDP operation.
- The status response message (RESPONSE) – This message is used to respond to the CHECK.
- The network status checking message (NETCHECK) – This message is used to determine the status of the network after a site status change is detected.
- The network status response message (NETRESPONSE) – This message is used to respond to the NETCHECK.
- The status inconsistency report message (ALERT) – This message is used to alert sites to the problem that inconsistent view vectors might be generated in the partition during SCP execution.
- The message to acknowledge the ALERT (ACK)
- The message carrying the view vector, V_vector_i, which contains the partition view at site i – Component k of V_vector_i, $V\_vector_i(k)$, represents the status of site k, UP or DOWN, as viewed by site i.

- The message to acknowledge that the view vector was received (VIEWACK)

In order to handle arriving messages, the following queues are maintained by the SM:
- The C–queue – This queue is used to receive the CHECKs and the NETCHECKs.
- The R–queue – This queue is used to receive the RESPONSEs, the ALERTs, and the NETRESPONSEs.
- The T–queue – This queue is used to receive requests from local transactions.

To maintain a current, consistent view of a partition and to support the exchange of status messages, each site maintains a Site Status Table (SST). The SST is shared with local transactions but is updated only by the SM. Each entry in the SST represents a site (S_i) in the network and has four fields:

1) Request_Set : The set of local transactions requesting message exchange with site S_i

2) Sent_Flag : Set when a CHECK is sent to site S_i

3) Received_Flag : Set when a RESPONSE is received from site S_i

4) Status : UP if S_i is available, DOWN if not

In order to detect failures, two timers are used:
- BROADCAST – This timer is used by the FRDP and the SCP.
- VCHECK – This timer is used by the SCP for robustness.

In order that a consistent timeout period be used throughout the network, it is assumed that each site owns a physical clock and that all clocks run at close enough to the same rate so that there is no discernible difference between timeout periods.

The following variables are used by the SCP protocol at each site.
- L_{ID} – is used to synchronize multiple detections of a status change.
- D_{VAL} – is used to store the values carried by ALERTs and NETRESPONSEs.
- *D_Record* – is used to keep track of possible problem sites.
- ST (State variable) – is used to keep track of the state of the protocol execution: The state can be "Normal", "Init_Collection", "Part_Collection", or "View_Sent". When the SM is executing the FRDP protocol, ST is set to "Normal". When the SM initiates the SCP to collect the current status information for partition reconfiguration, ST is set to "Init_Collection". ST = "Part_Collection" means that the SM is participating in a collection. When a new view vector is sent by the SM, ST is set to "View_Sent".

4.2 Description of the Failure and Recovery Detection Protocol

When a transaction sends a request (*Read* or *Write* request) to remote sites, it sends a request for a *watch* on the sites involved to the SM. When the SM receives a watch request from a local transaction, it puts the transaction identifier (TR_j) into the Request_Sets of the sites from which that transaction is requesting service:
SST.Request_Set$[S_i]$ = SST.Request_Set$[S_i] \cup TR_j$.

While a local transaction is waiting for service from remote sites, its SM periodically sends CHECKs to the sites involved and receives RESPONSEs from those sites. After the transaction receives its service from all remote sites that are UP, it sends a *watch release* to the SM to indicate that it does not need to check the status of those sites any more. The SM then deletes this transaction identifier (TR_j) from the Request_Sets of the sites:
SST.Request_Set[S_i] = SST.Request_Set[S_i] - TR_j.

When the SM sends out a CHECK, the timer BROADCAST is restarted. The Received_Flags of all sites in the SST are reset and the Sent_Flags for those sites to which the CHECK is sent are set. When a RESPONSE is received, the SM sets the Received_Flag of the source site entry in the SST. The SM responds with a RESPONSE as rapidly as possible upon receiving a CHECK. When BROADCAST expires, the SM checks whether the sites with set Sent_Flags responded. If all sites marked UP responded and none marked DOWN responded, then the SM generates the next CHECK and sends it to the sites. If any site marked DOWN responded or any site marked UP did not respond, then a status change is detected and the SCP is executed. While the SM is waiting for BROADCAST to expire, it may receive CHECKs as well as RESPONSEs from remote sites. These are handled as described above.

If a CHECK is received after BROADCAST expires but before the next CHECK is generated, the SM cannot respond with a RESPONSE immediately, but must wait until after the next CHECK is generated. Then, in order for the SM to respond quickly, before it sends out the CHECK it processes the C-queue. Let T_{check} and T_{msg} denote the times needed to check the site responses and to generate the next CHECK, respectively. And let T_{cq} be the maximum time to process the C-queue. (Note that the actual time is proportional to the length of the C-queue.) Let $T_{GEN} = T_{check} + T_{msg} + T_{cq}$. Thus, the timeout period of BROADCAST must be set to at least $2T_{MAX} + T_{GEN}$ in order to ensure that responses from normally operating sites will be received before BROADCAST times out.

The actions of the SM during normal FRDP operation may be then summarized as follows.

- **Event:** timer BROADCAST expires.

1. The SM checks the SST to see whether all sites to which CHECKs were sent have responded. If all sites marked UP responded and all sites marked DOWN did not, go to step 2. (Otherwise, a status change is suspected and normal FRDP operation stops. The SCP is then executed.)
2. The next CHECK is generated if there is a site in the SST with a non-empty Request_Set.
3. The C-queue is processed. At this time, the SM responds with RESPONSEs to all CHECKs received after BROADCAST expired.
4. If a CHECK has been generated, the SM sends this message to all sites with non-empty Request_Set and restarts BROADCAST.

- **Event:** Receipt of a watch request

The SM puts the transaction identifier into the Request_Set of each of the sites for which a watch was requested. If the SM is not currently sending CHECKs to any sites, then CHECKs are sent to the sites with non-empty Request_Set and BROADCAST is restarted.

- **Event:** Receipt of a watch release

The SM deletes the transaction identifier from the Request_Sets of the sites.

- **Event:** Receipt of a CHECK

The SM responds with a RESPONSE to the source site.

- **Event:** Receipt of a RESPONSE

The SM sets the Received_Flag of the source site entry in the SST.

- **Event:** Receipt of a NETCHECK

FRDP operation stops. See SCP discussion for details.

4.3 Description of the Status Change Protocol

In summary, the SCP works as follows. When the SM detects failure or recovery, it suspends the locally executing transactions and broadcasts a NETCHECK to collect up-to-date status information from all sites. After collecting the responses (NETRESPONSEs) from the sites, it creates a new view vector and sends it to all sites in the new partition.

An SM which detects failure or recovery becomes a system reconfiguration initiator (SRI) for the SCP, and the SM's of other sites become cohorts. (Notice that the SRI is not the database merge coordinator, but only an initiator for system reconfiguration. The site with the lowest rank in the logical partition which is formed after system reconfiguration is automatically assigned as the coordinator.) Many sites can simultaneously be SRI's, because status changes may be detected nearly simultaneously by different sites and each site that detects a status change executes the SCP. Therefore, multiple executions of the SCP must be synchronized. To accomplish this, the SM of each site tries to be a cohort of the SCP execution that was initiated by the SRI at the site of the lowest rank among the SRI's in the partition.

Unless some messages can be lost during the execution of the SCP, SRI's with higher site rank will abort their executions upon receiving the NETCHECK from the SRI of lower site rank. But it is quite possible that inconsistent view vectors can be delivered to different sites if some messages are lost.

[Example 4.1] Let us suppose the following situation. Two sites, i and j, become SRI's nearly simultaneously and the rank of i is lower than that of j. Site j broadcasts a NETCHECK and then site i broadcasts a NETCHECK. All cohorts except site i receive the NETCHECK from site j, but the NETCHECK sent to site i from site j is lost. Before the NETCHECK from site i reaches the cohorts, all cohorts respond to site j. They all accept site j as the proper SRI. The NETCHECK from site i then arrives at all cohorts except site j, while the NETCHECK sent to site j from site i has been lost. All cohorts realize that the rank of site i is lower than that of site j and they now accept site i as the proper SRI. Site j generates a new view vector which has site i marked DOWN and sends it to the cohorts. Site i generates a new view vector which has site j marked DOWN and sends it to the cohorts. Now two inconsistent view vectors have been sent by two different SRI's. All sites except site j accept the new view vector from site i, but site j's view vector is different from that of the others. This is an undesirable situation since two different views which are not disjoint make partition management difficult and the error will soon be detected and need to be rectified. □

Therefore, the SCP protocol must satisfy the following condition to guarantee consistency of views.

[Condition 4.1] For any two sites S_i and S_j with ST = "Normal", if S_j is marked UP in *V_vector*$_i$, then S_i must be marked UP in *V_vector*$_j$ and vice versa. □

Condition 4.1 guarantees that a site is marked UP in only one partition view at any one time; that is, the sets of sites marked UP in any two views are either identical or disjoint. In order to enforce this condition, when a cohort receives NETCHECKs from the SRI's as in example 4.1, it sends ALERTs to the SRI's that may initiate different views informing them of the problem, so that the SRI with higher rank aborts its SCP execution; that is, if the SRI with higher rank did not receive a NETCHECK from the SRI with lower rank, then it aborts its SCP execution upon receiving an ALERT. When the SRI sends out a new view vector it waits for acknowledgements from cohorts. If all cohorts acknowledge, then the SRI sets its state to "Normal". While the SRI is waiting for acknowledgements, if it receives a NETCHECK, then it assumes that the network status has changed or inconsistency has been detected. It then participates in the execution of SCP by the site that sent the NETCHECK. This mechanism enforces condition 4.1. (See Chapter 5 for proof.)

Since the SRI can fail during the execution of the SCP, the protocol must be robust against this possibility. That is, the protocol must guarantee that its execution will finish successfully despite SRI failure. The timer VCHECK is used to detect SRI failure and to support the resolution of inconsistent views.

The actions of the SM during SCP execution are described as follows.

- **Event:** A status change is detected by site i running the FRDP.

1) The SM stops FRDP execution, sets ST to "Init_Collection", and stores its site rank in L_{ID}.
2) The SM initializes each component of *V_vector*$_i$ to DOWN.
3) The SM then broadcasts a NETCHECK to all sites in the network and starts the timer BROADCAST to $2T_{MAX} + T_{GEN}$. (It is assumed that T_{GEN}, although derived from FRDP event timings, is more than adequate time for the generation of a NETRESPONSE.) Normal FRDP operation is resumed after the successful execution of the the SCP.

- **Event:** Receipt of a NETCHECK at site i from site k.

For simplicity, let k be the rank of site k. Upon receiving a NETCHECK, the SM performs different operations depending on its state:

○ *Case* 1: ST = "Init_Collection" - The SM has initialized the collection of status information.

1) The SM compares k with L_{ID}, its site rank, and takes appropriate action:

○ *Case* 1.1: $k < L_{ID}$

The SM stores k in L_{ID}, and aborts its SRI activity by setting ST to "Part_Collection". Now it considers itself participating in the collection initiated by the site k.

The SM starts the timer VCHECK. The timeout period of VCHECK is $3T_{MAX} + 2T_{GEN} - T_{MIN}$. (See chapter 5 for the justification of this timeout period.)

○ *Case* 1.2: $k > L_{ID}$

The SM sets *V_vector*$_i(k)$ = UP.

2) The SM then sends site k a NETRESPONSE with parameter D_{VAL} set to L_{ID}. The inclusion of parameter D_{VAL} causes site k to abort its SCP execution in case 1.2.

○ *Case* 2: ST = "Part_Collection" - The SM is participating in the collection initiated by another SRI.

The SM compares k with L_{ID}, the lowest SRI site rank seen thus far.

○ *Case* 2.1: $k < L_{ID}$

In this case, inconsistency may occur as described in Example 4.1.

The SM sends an ALERT to site k with parameter D_{VAL} set to L_{ID}, and an ALERT to the site whose rank is equal to L_{ID} with parameter D_{VAL} set to k. The ALERT causes the site with higher rank to abort its SCP execution if it had not received a NETCHECK from site k, and causes site k to mark the site with higher rank UP in case site k did not receive a NETCHECK from that site.

The SM stores L_{ID} in *D_Record*. The entry for the site with rank equal to L_{ID} in *D_Record* will be removed if the SM receives an ACK from that site.

The SM stores k in L_{ID}.

The SM starts the timer VCHECK.

○ *Case* 2.2: $k > L_{ID}$

The SM sends site k a NETRESPONSE with parameter D_{VAL} set to L_{ID}. Receipt of the parameter D_{VAL} causes site k to abort its SCP execution.

○ *Case* 3: ST = "Normal" - The SM is not executing the SCP.

1) The SM stops its normal FRDP operation.
2) The SM stores k in L_{ID}
3) ST is set to "Part_Collection".
4) The SM sends site k a NETRESPONSE with parameter D_{VAL} set to L_{ID}.
5) The SM starts VCHECK.

○ *Case* 4: ST = "View_Sent" - The SM sent a new view vector to cohorts and is waiting for their acknowledgements.

In this case, receipt of a NETCHECK means that the network status has changed or inconsistency has been detected by a cohort.

1) The SM stores k in L_{ID}.
2) ST is set to "Part_Collection".
3) The SM sends site k a NETRESPONSE with parameter D_{VAL} set to L_{ID}.
4) The SM starts VCHECK.

- **Event:** Receipt of a NETRESPONSE at site i from site k.

○ *Case* 1: ST = "Init_Collection"
1) The SM compares D_{VAL} with L_{ID}
○ *Case* 1.1: $D_{VAL} = L_{ID}$
The SM sets $V\_vector_i(k) =$ UP.
○ *Case* 1.2: $D_{VAL} < L_{ID}$ - Another SRI with a lower site rank is executing the SCP.
The SM aborts its SCP execution by setting ST to "Part_Collection".
The SM stores D_{VAL} in L_{ID}.
The SM starts the timer VCHECK.
○ *Case* 2: ST = "Part_Collection"
The SM just discards the NETRESPONSE.

- **Event:** Receipt of an ALERT from site k.

○ *Case* 1: ST = "Init_Collection"
The SM compares D_{VAL} with L_{ID}.
○ *Case* 1.1: $D_{VAL} < L_{ID}$ - The SM did not receive a
The SM aborts its SCP execution by setting ST to "Part_Collection".
The SM stores D_{VAL} in L_{iD}
The SM starts the timer VCHECK.
The SM sends an ACK to site k.
○ *Case* 1.2: $D_{VAL} > L_{ID}$ - The SM did not receive a NETCHECK from the SRI with higher rank.
The SM sets the view vector entries for site D_{VAL} and site k to UP.
○ *Case* 2: ST = "Part_Collection" - The SM already received a NETCHECK from the SRI with lower rank.
1) The SM just discards the ALERT.
2) The SM sends an ACK to site k.

- **Event:** BROADCAST expires.

○ *Case* 1. ST = "Init_Collection" - Normal case
1) The SM generates a new view vector according to the responses from the cohorts.
2) The SM sends V_vector_i to all sites UP in its view.
3) The SM sets ST = "View_Sent".
4) The SM starts BROADCAST.
○ *Case* 2. ST = "Part_Collection" - The SM has aborted the execution of the SCP.
The SM ignores the signal from BROADCAST and waits for a new view vector.
○ *Case* 3. ST = "View_Sent" - The SM has sent a new view vector to cohorts.
If the SM did not receive VIEWACKs from all sites marked UP in its view, it assumes that there has been a problem and initiates the SCP again.
If the SM received VIEWACKs from all sites marked UP in its view, it sets ST = "Normal" and returns to FRDP operation.

- **Event:** Receipt of a new view vector from site k.

○ *Case* 1: ST = "Part_Collection"
○ *Case* 1.1: $k \neq L_{ID}$ - The SM realizes there is an inconsistency.
The SM resets all necessary data structures and initiates the SCP.
○ *Case* 1.2: $k = L_{ID}$
The SM checks *D_Record* to see if there is a possibility of inconsistency. If *D_Record* is empty then the SM updates its Site Status Table (SST) with the new view vector, sets ST = "Normal", sends a VIEWACK to site k, and resumes FRDP execution. Otherwise, it assumes that there has been a inconsistency problem and initiates the SCP.
○ *Case* 2: ST = "Normal" or "Init_Collection"
The SM discards the new view vector received.
○ *Case* 3: ST = "View_Sent" - This can only happen if an ALERT to this SM notifying it of a lower ranked SRI was lost.
The SM accepts the new view vector, sends a VIEWACK to the sender, sets ST = "Normal", and resumes FRDP execution.

- **Event:** VCHECK expires. - The SM has been waiting for a new view vector.

○ *Case* 1: ST = "Part_Collection"
The SM assumes that the SRI has failed, resets all necessary data structures, and executes the SCP as when a status change is detected.
○ *Case* 2: ST = "Normal"
The SM just discards the signal from VCHECK.

- **Event:** Receipt of an ACK from site k.

○ *Case* 1: ST = "Part_Collection"
The SM deletes site k's entry from *D_Record*.
○ *Case* 2: ST = "Normal" or "Init_Collection"
The SM just discards the ACK.

- **Event:** Receipt of a VIEWACK.

The SM records the acknowledgement from the sender.

- **Event:** Receipt of a CHECK or RESPONSE used by the FRDP.

The SM just discards them.

4.4 Starting or Restarting sites

Since both recovery from a crash and partition merging are treated as partition merging, a site just coming on line or recovering after a crash executes the SCP, informing all sites with which it can communicate of its recovery. When a site rejoins the network, it must execute the network merge process since it does not know whether it becomes a member of the same partition which it was in before the failure.

5. Discussion

We believe that implementation of our protocol is quite feasible for support of the optimistic approach to partitioned operation. Since our protocol provides the necessary network status change points, a partition merge process to restore the database to a consistent state can be done correctly.

Before we prove properties of our protocol, a justification of the timeout period of the timer VCHECK, which is used by the SCP to detect failure of the SRI, is presented. The timeout period of VCHECK must safely allow enough time for the proper operation of the protocol, so that false failure detection is avoided.

[Assertion 5.1] The timeout period of VCHECK is safe.

[Proof] The timer VCHECK is used in four cases.

Case 1. (VCHECK is started after sending a NETRESPONSE to the SRI.)

Suppose that site i becomes a cohort of the SCP initiated by site j when i receives a NETCHECK. Site i responds with a NETRESPONSE to the NETCHECK and starts VCHECK. Site i expects to receive a new view vector from site j before VCHECK expires. Suppose that site j sends out the NETCHECK to sites and starts the timer BROADCAST at time t_1. Site i receives the NETCHECK at time t_2. The minimum time period between t_1 and t_2 is T_{MIN}. Thus, the NETCHECK is received at site i no earlier than $t_1 + T_{MIN}$. Therefore, $t_1 + T_{MIN} \leq t_2 \leq t_1 + T_{MAX}$. Site j's timer BROADCAST expires at $t_1 + 2T_{MAX} + T_{GEN} = t_3$. Therefore, the maximum time between t_2 and t_3 is $2T_{MAX} + T_{GEN} - T_{MIN}$. After BROADCAST expires, site j takes at most T_{GEN} to generate the view vector and the view vector is delivered within T_{MAX} to site i. Therefore, under normal operation the new view vector must be received at site i before $t_2 + 3T_{MAX} + 2T_{GEN} - T_{MIN}$.

Case 2. (VCHECK is started after sending an ALERT to the sites which may be involved in the generation of inconsistent view vectors.)

Suppose that ALERTs are sent to site i and j from site k at time t_1, and $i < j$. Site j is expected to take at most T_{GEN} to generate an ACK and the ACK takes at most T_{MAX} to be delivered to site k before time $t_1 + 2T_{MAX} + T_{GEN} = t_2$. The new view vector from site i will be delivered at site k before $t_1 + 3T_{MAX} + 2T_{GEN} - T_{MIN} = t_3$ as in *Case* 1. Since $T_{MAX} + T_{GEN} - T_{MIN} > 0$, $t_3 > t_2$. Thus, the new view vector and ACK must be received at site k before $t_1 + 3T_{MAX} + 2T_{GEN} - T_{MIN}$.

Case 3. (VCHECK is started when a site j receives a NETRESPONSE which indicates that another SRI with a lower site rank is executing the SCP.)

Suppose site i is the SRI with the lower site rank and a NETRESPONSE arrives at site j from site k at t_2. If site i sent out a NETCHECK at t_1, then it must arrive at all cohorts before $t_1 + T_{MAX}$. By assuming that NETCHECK from i to k took T_{MIN} and the NETRESPONSE from k took T_{MIN}, the new view vector from site i must be received at site j by $t_2 + 2T_{MAX} + T_{GEN} - 2T_{MIN} + T_{GEN} + T_{MAX} = 3T_{MAX} + 2T_{GEN} - 2T_{MIN}$.

Case 4. (VCHECK is started when an ALERT is received.)

Suppose that site i sent a NETCHECK at time t_1 and site k sent an ALERT at time t_2. If the NETCHECK took T_{MIN} and the ALERT took T_{MIN} to be delivered, then site i's BROADCAST will expire at some time before $t_2 + 2T_{MAX} + T_{GEN} - 2T_{MIN}$. Site i will take at most T_{GEN} to generate the new view vector and the new view vector will take at most T_{MAX} to be delivered. Therefore, the new view vector must arrive before $t_2 + 3T_{MAX} + 2T_{GEN} - 2T_{MIN}$.

Therefore, the timeout period of VCHECK is safe for all four cases. □

Next, we demonstrate the property of our protocol that all sites in a newly formed partition have a consistent view.

[Assertion 5.2] The Status Change Protocol satisfies condition 4.1

[Proof] Suppose condition 4.1 is violated. Since views are only created by SRI's, there must be two different SRI's, S_1 and S_2, $S_1 < S_2$, each of which assumed the other was DOWN. Furthermore, they both received acknowledgements successfully from all of their cohorts after they sent out the new view vectors. In this case, (a) there must be a site S_k which responded to both SRI's with NETRESPONSEs, or (b) ALERTs that were sent to the SRI's were lost so that the SRI's were not aware of the problem. From the SCP specification, if the SM of S_k receives a NETCHECK from S_2 when it had already received a NETCHECK from S_1, then it sends an ALERT to both S_1 and S_2. Therefore, case (a) is impossible. Assuming case (b), the SM which sent ALERTs initiates the SCP upon receiving a view vector from S_2 before it receives a new view vector from S_1, and S_1 will receive a NETCHECK from the SM before its BROADCAST expires. If the NETCHECK was lost, S_1 would execute the SCP again since it did not receive acknowledgements from all of its cohorts. If the SM which sent ALERTs receives a new vector from S_1 before it receives a new vector from S_2, then it checks whether an ACK from S_2 is received already and if not it initiates the SCP. Also, S_2 will not set its ST = "Normal" unless it receives acknowledgements from all of its cohorts. Therefore, the assumption contradicts the specification of the SCP. □

Let us elaborate on our comment that our protocol provides the necessary status change points. The problem is that when partitions are merged, the merge algorithm must know when partitioning occurred in terms of transaction history.

For simplicity, let partitioning be ***pairwise***, meaning partitioning subdivides one old partition P_k into two subpartitions – P_i which contains the coordinator of P_k, and P_j.

Definition 5.1 A *partitioning point* for the site S_j, $LT_k(S_j)$, $S_j \in P_k$, is the last transaction processed at site S_j in P_k before P_k is partitioned.

Definition 5.2 A *partitioning point* I_i^k of partition P_i, which was formed when P_k was partitioned, is the set of $LT_k(S_j)$, for all S_j in P_i.

When P_i and P_j are merged to re-form P_k, the coordinator of P_j sends I_j^k to the coordinator of P_i (i.e. of P_k) so that conflicting transactions may be serialized beginning at that point.

[Assertion 5.3] If partition P_k is subdivided into subpartitions P_i and P_j as defined above, then the FRDP and the SCP correctly determine I_i^k and I_j^k.

[Informal Proof] We must show that when partitioning occurs and there is a possibility of a serializability conflict, the partitioning is detected by our protocol and the coordinator obtains the necessary partitioning point information. To see this, there are four possible cases to consider, depending upon whether or not possibly conflicting transactions are processed in the partitions while P_k is partitioned:

Case 1. Transactions are processed in both P_i and P_j.

When a transaction in either partition executes an action on a data item having a copy at a remote site (in the other partition), it sends a watch to the SM. If a site fails, the FRDP then detects it, transactions are halted, and the SCP updates the network view. The coordinators of both partitions are elected and they collect the partitioning points, I_j^k and I_i^k from the sites.

Case 2. Transactions are processed in P_i but not in P_j.

P_i notices the partitioning, and the coordinator of P_i is elected and it collects I_i^k from the sites in P_i as above. When P_i and P_j are merged, the coordinator of P_i requests the execution history from P_j and no conflicts need to be resolved.

Case 3. Transactions are processed in P_j but not in P_i.

P_j notices the partitioning and its coordinator collects I_j^k. The coordinator of P_k, which is the same as the coordinator of P_i, does not know about the partitioning since no transactions involving sites in P_j are processed. When P_i and P_j are merged, the coordinator of P_j sends I_j^k to the coordinator of P_i. The coordinator of P_i gets the partitioning point by checking the history of the transactions processed in P_k against I_j^k. It finds that there has been no possibility of a serialization conflict.

Case 4. No transactions are processed in P_i and P_j.

The coordinator of the merged partition will recognize that there was no possibility of serialization conflict.□

6. Conclusion

We have presented a new protocol which supports optimistic partitioned operation by detecting failures and recoveries. We believe that our protocol is adequate to support optimistic partitioned operation.

Protocols based on periodic status checking require considerable overhead. Even when the system is lightly loaded or idle, the status of the network is periodically checked. Therefore, the system may unnecessarily work to detect failures and to reconfigure the system.

With our protocol, a status change may not be detected right after a site crashes or the network partitions, but it is detected later and system reconfiguration is initiated when the cooperation of that site is needed by a transaction. However, the whole system does not have to pay the cost of periodic checking.

In addition, our protocol handles the problem of synchronizing multiple failure/recovery detection in a particularly robust way.

ACKNOWLEDGEMENT

This work was supported in part by NASA under Contract No. NAG 1 - 613.

REFERENCES

[1] S.B. Davidson, "Optimism and Consistency in Partitioned Distributed Database Systems," ACM Trans. on Database Systems, Vol. 9, No. 3, Sept. 1984, pp. 456–481.

[2] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", CACM, Vol. 19, No. 11, 1976.

[3] H. Garcia–Molina, "Elections is a Distributed Computing System," IEEE Trans. on Computers, Vol. C–31, No. 1, Jan. 1982.

[4] M. Hammer and D. Shipman, "Reliability Mechanism for SDD–1: A System for Distributed Database," ACM Trans. on Database Syst., Vol. 5, No. 4, Dec. 1980, pp. 431–466.

[5] W. Kim, "AUDITOR: A framework for highly available DB/DC systems," Proc. 2nd Symp. Reliability in Distributed Software and Database Systems, 1982.

[6] A.V. Ma and G.G. Belford, "A Failure and Recovery Detection Protocol for Optimistic Partitioned Operation in Distributed Database Systems," Proc. 6th Int. Conf. on Distributed Computing Syst., May 1986.

[7] D.A. Menasce and R.R. Muntz, "Locking Protocol for Resource Coordination in Distributed Databases," Proc. SIGMOD Int.Conf. on Management of Data, 1978, pp. 1–14.

[8] C.H. Papadimitriou, "The Serializability of Concurrent Database Updates," J. ACM, Vol. 26, No. 4, Oct. 1979, pp. 631–653.

[9] D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, J.M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of Mutual Inconsistency in Distributed Systems," IEEE Trans. on Software Engr., Vol. SE–5, No. 3, May 1983.

[10] B. Walter, "A Robust and Efficient Protocol for Checking the Availability of Remote Sites," Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks, 1982, pp. 45–67.

[11] B. Walter, "Network Partitioning and Symmetric Surveillance Protocols," Proc. 5th Int. Conf. on Distributed Computing Systems, May 1985, pp. 124–129.

[12] D.D. Wright, "Managing Distributed Databases in Partitioned Networks," Ph.D. Thesis, Dept. of Computer Science, Cornell University, Sept. 1983.

Replication in Distributed Systems: The Eden Experience

Jerre D. Noe, Andrew B. Proudfoot, and Calton Pu

Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

Two different methods for replicating objects were implemented on the Eden local area network. One was at the object level, and it provided easier implementation and maintenance at some cost in performance. The other, at the kernel level, gave better transparency and performance from the client's view but was more difficult to implement. Both approaches maintained object availability in spite of crashes. Their implementation gave insight into the conflict between performance and transparency, the cost of naming replicated resources, and some inescapable effects on system users who do not want replication.

1. Introduction

In a distributed system that supports applications using multiple machines, the probability of failure-free execution is the product of the probabilities of success for the related sub-tasks running on the individual machines. Two approaches may be taken to increase the chance of success for a distributed application; either increase the reliability of the nodes or provide replicas of crucial resources. This paper is concerned with the latter approach, for which the following are the design goals:

1. The loss of copies of replicated resources should not prevent operation.
2. The use of replicas should not alter the client interface.
3. There should be minimum performance penalty for users of replication and none for non-users.

1.1. Background

Hardware replication has been an accepted design approach for a long time, but software replication is more recent. A non-exhaustive list of early efforts on software replication in distributed systems includes one that led to a commercial product, Tandem[1], which added replication of processes to its replicated hardware structure. Bernstein and colleagues[2], in an early design paper on the SDD-1 System, made use of distributed atomic transactions for updating replicas, using methods that depended upon a global clock. Lampson and Sturgis[3] introduced the concept of stable storage, using replicas of unreliable disk files in a careful manner to provide reliable storage, which was used to support atomic transactions. These ideas were incorporated in XDFS, the Xerox Distributed File System[4], where replicas were cached for performance reasons. The Cambridge File System[5] used backup replicas on disk, with transactions for atomic update, but the design was so conservative in its attempts to cause no penalty for non-users of replication that it was restricted to single-site transaction, a limitation later removed. An interesting comparison of CFS and XDFS is given by Mitchell and Dion[6].

Distributed INGRES provided replicas for databases as an add-on feature. The design is described by Stonebraker[7]. LOCUS[8], another commercially available system, supported replication of Unix™ files with a primary-copy update transaction

*This work was supported in part by the National Science Foundation under Grant No. MCS-800411.

CH2345-7/86/0000/1197$01.00 © 1986 IEEE

mechanism. Walker, et al., reported that concurrency control in their transactions was made more complicated by replication because the primary storage site no longer knew about all the activity for a given file.

Since the early experiments, many proposals and analyses of schemes for concurrency control and crash recovery have appeared in the literature, including some survey articles[2, 9]. One of the trends developing for such work has been the adaptation of distributed database tools for atomic transactions and their availability as part of the operating system. Several new designs and prototype implementations are being explored in which the resources for control of distributed transactions, replication and crash recovery are, to varying degrees, being built into the underlying systems[10, 11, 12, 13], but at this point few observed results have been reported. It is in this context that our experience with replication in the Eden system is of interest.

1.2. The Experiments and Summary of Observations

The Eden distributed system, which began operating in April 1983, has provided a means to implement and test various schemes for the control and use of replication. This paper reports on the observations from two such experiments. One approach, described previously[14], implemented replication at the object level. Another[15] was done at the level of the kernel which creates and supports Eden's object environment. These experiments arose through curiosity about the two approaches and were not initially conceived as a comparison, hence they differ in detail. Nonetheless they do provide an opportunity in retrospect to contrast some of their features.

With either approach, two features must be added to the non-replicated system: First, where formerly one object was dealt with, now a set of names of the copies of an object must be recognized and handled. Secondly, a mapping must exist between this set of names and a *single* name used by the client; without this level of indirection any change in the number or identity of replicas would require notifying all clients of the change. When these two functions, mapping and the use of multiple names, are implemented at object-level we mean that they are accomplished by objects without changing the underlying kernel or its interface. Alternatively, both functions may be embedded within the kernel, hence the term kernel-level implementation. These approaches differ in the degree of transparency to the user of replication, in the cost of programming the replication system and the execution cost for users and nonusers of the facilities.

We shall first state the main observations that result from the experiments. More complete discussions appear in the following sections of the paper.

- Both approaches succeeded: objects were made available in spite of crashes.
- The kernel-level implementation was more difficult and required more programming effort.
- The object-level implementation was easier due to modularization, location-independent invocations and high-level language support, but it requires some degree of kernel support beyond the initial Eden design.
- The kernel-level replication gave more transparency from the client's view.
- Kernel size was increased about 20% by including replication control.
- In a test case requiring writing to two disk copies, both implementations allowed replicated data to be read and written in less than twice the time required for ordinary non-replicated objects, due to concurrency in the updates.

Our experience led us to believe:

- There is an inescapable cost to the nonuser of replication, although careful design can keep it small.
- Transparency and performance are in conflict.
- If the underlying network operating system cannot deal with network partitions, it is difficult but not impossible to do so in replication systems that are added.
- Naming of replicated resources has a cost: Either the clients must hold names of multiple access routes, or some form of

dynamic binding, such as broadcast, must be used.

- One need not implement transaction management especially for the purpose of replication; specialized use of a general transaction tool can handle it.**

1.3. Structure of the paper

Having declared our principal findings above, we shall describe the evidence that led us to these conclusions. Some understanding of the Eden system is needed in order to proceed; this information is provided in Section 2. Section 3 describes the kernel-level approach, and section 4 summarizes the object-level approach. Section 5 provides detailed reasoning behind our observations. Section 6 seeks to relate our findings to what we understand about the "new-wave" of systems that are incorporating replication or transaction-control mechanisms. Section 7 summarizes the paper.

2. The Eden System

Eden is a distributed system used as a test-bed for experimental research on implementation of distributed computer systems and the construction of distributed applications. Many of its technical features have been described[16], and recently assessed[17]. The implementation on which the experiments were run consisted of sixteen Sun workstations, of which four are disk servers that provide virtual disk to the other twelve.

2.1. Logical features of Eden

Eden objects encapsulate data, the procedures that operate upon them, and active processes. Objects are defined by the user rather than being restricted to a few system defined types. All objects are named by system-wide unique identifiers called capabilities. All communication between objects is through invocation, a form of remote procedure call, with the same syntax used for local and remote invocations.

A high level Eden Programming Language (EPL) [18] is used for defining Eden objects. EPL is based on Concurrent Euclid[19] with extensions to handle capabilities and to deal with sending and receiving invocations. It provides "light-weight" processes and monitors that the application programmer may use to control concurrency.

Invocations are location independent and objects are mobile. Objects may move from one node to another or they may be fixed to specific locations, under the control of kernel calls. Eden objects always appear active from the viewpoint of the invoking object. To achieve this, Eden objects have two forms: the active form and the passive representation. The former is in volatile memory with a virtual processor assigned to it; the latter is the long term state of the object, recorded on disk. Creation of a new object establishes an active form which, at the discretion of the designer of the object, can *checkpoint* itself (i.e., record its passive representation on disk). This checkpoint to disk is the only atomic action that is built into the Eden kernel. It is implemented so that it either succeeds or has no effect. The object designer may also make checkpoint available as an invocation so that other objects may request the active form to checkpoint itself. The active form may vanish due either to a crash or to deliberate deactivation. A subsequent invocation of the object will cause the kernel to fail to find the active form and therefore to search for the passive representation and create a new active form which then responds to the invocation. Thus an object always appears active from the viewpoint of the invoker, but the response time may vary depending upon whether or not reactivation had to take place. As originally designed, each object instance could have at most one active form and one passive representation. This is the feature that becomes changed by replication, as subsequent sections will show. Replects, in the kernel-level implementation, provide a single active form and multiple passive representations; R2D2 replicates both the active forms and passive representations.

**Note that this says nothing about the use of general transaction mechanisms for *all* applications, as opposed to taking advantage of application semantics. That distinction requires further research.

2.2. Implementation features of Eden

The kernel of the Eden distributed operating system is composed of two sets of cooperating processes. Running on each node is a process called the Host that is responsible for creating objects, protecting capabilities from forgery, locating objects, and passing invocation messages among objects. The second type of process is the *permanent object database* (POD), a copy of which may run on any node, although it makes most sense to run them on the nodes attached to physical disks. The normal Eden POD stores the passive representations, ensures that at most one active form and passive representation exists, and cooperates with the Hosts to create an active form for an invoked object if none exists.

Eden objects may be thought of as values of abstract types. They will perform certain actions in response to invocations, as well as internal actions. The code implementing these abstract types, i.e., the concrete types, is encapsulated in TypeStore objects and this code may be "replugged" (e.g. to provide a faster implementation) as long as the externally available invocations are preserved. TypeStore objects have all the features of ordinary Eden objects but in addition, their data consists of executable code. Every object includes a capability for the TypeStore that contains its code. In principle, all objects of the same type share the TypeStore object; in practice, for implementation reasons, one copy of the code in a TypeStore is cached on each node that contains one or more objects of that type. Obviously, if instances of Eden objects are replicated, the TypeStore must also be replicated in order to protect against crashes, and updates should be automatically handled by the same mechanism.

3. Kernel-Level Replication: Replects

The Replect approach, the kernel-level implementation of replication in Eden, seeks to overcome the loss of passive representations stored in unavailable PODs. This is done by creating multiple passive representations on separate machines, thus increasing the probability that a passive representation will be available for activation if an object is invoked. From the viewpoint of the client there is no visible change between invoking a Replect or an ordinary Eden object; at most one active form exists and it is invoked using one capability. The difference is visible, of course, to the client that creates the Replect; the number of copies is under its control. TypeStore replication is straightforward; a TypeStore object is just another Replect.

3.1. Replect implementation

The Replect approach required changes to the kernel; a few to the Host and an almost complete rewrite of the POD.

When a Replect is to be altered, the update must be treated as a distributed transaction in order to allow competing actions (activations, checkpoints, typestore reads) to take place without mutual interference and in order to protect against crashes that might leave inconsistent copies. Similarly, transactions are required during activation to ensure that only one active form is produced. In both cases, one POD is selected to act as the transaction manager. During update the transaction manager locks the other participating POD's passive representations, writes a commit record, causes participating PODs to write shadow copies and/or writes in-place entries in the headers for the copies (with an "undo" log for crash recovery). It then uses a two-phase commit protocol to complete the transaction. If the transaction manager crashes when the slaves are in the ready-to-commit state, the slaves are blocked until the transaction manager recovers.

Gifford's voting scheme[20] is used both for activation and for updating. Given a number of copies of an object, Gifford's voting method makes use of a read-quorum and a write-quorum chosen so that their sum exceeds the total number of copies. This overlap ensures that when a read-quorum is read, at least one copy of the latest version is found. There are minor differences between the scheme actually used and the approach described by Gifford, but no major change in the concepts.

Gifford assumed an underlying general transaction mechanism. The Replect implementation did not have one to draw upon and provision of it accounted for a fairly large part of the implementation effort. However, by not exporting a general transaction tool to Eden users, shortcuts in the Replect implementation were possible. The transaction manager code is in a POD with no interface to objects and the transaction deals only with single objects that are replicated. In contrast, a generally usable transaction tool would have to deal with a list of objects involved in the transaction. Instead, the kernel code within the POD deals only with a list of all the PODS that have copies for a given Replect.

Very few changes were made in the Host kernel code, involving only about a one or two percent increase in the total amount of Host code. POD code, however, had to be extensively rewritten and its size increased by about 20%. The Replect approach took a graduate student the better part of a year elapsed time in order to become acquainted with the problem, and define the implementation, followed by about six months of coding, debugging, and testing.

3.2. Replect system performance

To assess performance of Replects, measurements were made under three different conditions:

1. The Eden object case: With the standard Eden system, not modified for Replects.
2. The 1-copy Replect case: With the Replect system, using single-copy Replects. This assessed the penalty for using Replect kernel code for accessing non-replicated objects, rather than branching to standard code.
3. The multi-copy case: With the Replect system, using three copies of the passive representation, with read quorum and write quorum equal to two. ($n = 3$; $n_r = n_w = 2$). TypeStore used $n = 3$, $n_r = 1$ and $n_w = 3$.

Multiple measurements were made, to provide data at a confidence level of 98% with the confidence interval varying for the different cases, but in the neighborhood of plus or minus 10%. The measures of interest were the elapsed times for activation, read invocation and write invocation involving checkpoint of the modified objects. ByteStore objects containing 1000 bytes of data were used.

Activation of Replects required 10% to 13% more than activation of a standard Eden object. This was for the most favorable activation case in which current copies of the passive representation and TypeStore were located at the host where the ByteStore was to be activated. The standard Eden object required a mean time of 1.94 seconds; the single-copy Replect took 2.13 seconds; the multi-copy case required 2.20 seconds.

In the very unfavorable activation case, in which neither the passive representation nor the TypeStore were on the host where activation was to occur, activation time was much longer; 32.5 seconds for the Eden object case and 34.4 seconds for the multi-copy Replect. This involved locating the TypeStore, having a copy sent over the network, running the activation protocol, finding and copying the remote passive representation.

Remote read invocation of an active object should be, and was, no greater for the Replect case than for the single Eden object. A "do-nothing" invocation requires 0.084 seconds. The test invocation read 1000 bytes. The Eden object mean invocation time for reading was 0.237 seconds at 98% confidence level with, in this case, a confidence interval of plus or minus 2%. The mean times for the single and multiple Replect invocations fell within this range.

A write invocation to active objects required 13% more time for a single-copy Replect and 27% more for the multi-copy case compared to the standard Eden object. The Eden object required 1.81 seconds; the single-copy Replect took 2.05 seconds and the multi-copy Replect used 2.30 seconds. Concurrent writing of the two copies of the passive representation kept the elapsed time well below double the single object figure.

4. Object-Level Replication

A previous paper[21] describes an Eden object level implementation of replication. The emphasis in that report is on the Regeneration method of increasing availability, and its advantages and disadvantages relative to other data replication methods. For example, Regeneration differs from Voting in the number of replicas necessary for each access. To read or update a resource replicated with Regeneration, one copy is sufficient. If copies are inaccessible, new and up-to-date copies are created elsewhere to substitute the inaccessible ones.

At the heart of the object level replication is a replicated directory, described in section 4.1. In the replicated directory, the user may specify the number of copies for each resource; the copies are placed on machines with independent failure modes, with user over-ride. Replication of general objects is described in section 4.2. Implementation effort and performance are summarized in section 4.3.

4.1. The Replicated Directory

As we have mentioned in section 1.2, a mapping must exist between a single resource name used by the client and the set of copies. In the Replect implementation (section 3.1), the mapping is stored in each POD containing a copy, and a broadcast mechanism is used to find them. Since there is no such broadcast mechanism at the object level, a mapping must be constructed. Furthermore, the mapping must be itself replicated for availability; otherwise a replicated resource would be as available as a non-replicated mapping. For object-level replication, the Eden Replicated Resource Distributed Database (R2D2) was designed and implemented to serve as the replicated mapping.

R2D2 maps pathnames (character strings similar to Unix pathnames) into sets of capabilities. Each level in the path corresponds to a directory, which may be replicated with several copies. Each copy of a replicated directory is unaware of its siblings; a specialized transaction manager maintains the consistency of replicas despite concurrent access and crashes. Since pathnames are organized in a hierarchy, R2D2 has a tree structure with replicated nodes.

R2D2 supports typical operations on a mapping such as Lookup, Add, Replace, and Delete. Each request is handed on to the R2D2 transaction manager, which traverses the tree and finds the appropriate directory on which the operation must be performed. For example, *Lookup("users/alice/testfile")* is a read operation serviced by one copy of the directory named *"users/alice"*. In contrast, a write operation like Add is sent to all copies. If any of the set is unavailable, it is replaced by Regeneration, which creates new copies on operable machines to replace the missing ones. This ensures that the transaction never fails due to crashes, as long as sufficient machine resources remain to handle the required number of copies.

If a missing copy of the lowest directory is replaced this requires altering the next higher directory, to eliminate the capability of the missing copy and replace it with the newly created one. This approach proceeds recursively up to the root of the addressing structure. Note, however, that protection against a crash of the root must be handled in some other way. The manner in which "hardening" the root against crashes is done depends upon other requirements that one might wish to meet, such as performance or transparency at the client interface. One might give clients a fixed set of capabilities, with enough copies of the root to give a satisfactory level of availability without regeneration, but that requires all clients to include code to deal with the set of capabilities. Alternatively, one might arrange to broadcast for access to a copy of the root, or to use the Replect scheme. These alternatives were not explored in the R2D2 implementation, and a Replect was used for the root.

The root directory receives all R2D2 requests and forwards them to R2D2 transaction managers for processing. A centralized lock manager in the root synchronizes concurrent updates. At the time R2D2 transaction manager receives an update request, it is given an implicit lock on the name being updated. To avoid deadlocks, conflicting updates are returned to clients for later resubmission.

4.2. Eden Resource Management System

Although R2D2 provides transparent access to replicated directories, other invocations to client-defined objects are not directly supported by R2D2 transaction manager. The Eden Resource Management System (ERMS)[14], has been designed and implemented to provide atomic object access within nested transactions. Here, we delineate ERMS transaction features concentrating on its client access to replicated resources.

From within transactions, ERMS clients access their resources, which are controlled by a transaction manager. The ERMS transaction manager is more general than the R2D2 transaction manager, since ERMS handles groups of objects of any type. ERMS has adopted locking concurrency control, and a lock manager (separate from the R2D2 lock manager which handles single replicated objects) synchronizes resource access.

For update atomicity, ERMS uses R2D2 as the authoritative mapping; only committed versions of resources are stored in R2D2. At the beginning of a resource update, the ERMS transaction manager creates a new version, which is a single object directly invoked by the client. At commit time, the ERMS transaction manager creates the necessary number of copies, co-located with the old version if possible, and installs the new copies in R2D2, substituting the old versions.

4.3. Discussion

The design of R2D2 took about the same time as for Replects. The coding and debugging took approximately three man months. Additional design and implementation effort was dedicated to ERMS. Significant code sharing between R2D2 and ERMS, such as the locking concurrency control, brought ERMS online in another three months.

ERMS provides other functions beyond handling client access to replicated resources. For example, ERMS nested transactions are implemented with nested concurrency control and crash recovery, both independent of replication. Also, ERMS uses one transaction manager per transaction to simplify implementation, introducing some additional communication cost. Consequently, ERMS resource access time is dominated by features unrelated to replication.

In this paper, we use R2D2 as an example of object level replication, and compare R2D2 performance measurements with those of a non-replicated directory with the same tree structure. Multiple measurements were made to provide data at a confidence level of 98%, with the confidence interval at plus or minus 4%. For the read case, a mean value of 0.23 seconds was obtained for the non-replicated directory and 0.42 seconds for the replicated resource. The increase is due to the extra invocations to the R2D2 root and its transaction manager. This could be decreased by combining these two objects into one, but it would be done at the cost of decreased modularity and probably more difficulty in debugging and maintaining R2D2.

For updates in which no regeneration of missing copies was necessary, the two-copy resource required more time than for the non-replicated directory by a factor of 1.5. The single directory required 1.43 seconds and R2D2 needed 2.18 seconds to update two copies.

5. Observations Based on the Implementations

Availability, transparency, and performance were affected by the level at which replication was implemented, by the concurrency control and crash recovery methods used, and by features of the underlying Eden systems. Implementation problems also were affected by these factors. The following sections comment on these facets.

5.1. Availability

Both implementations succeeded in making objects available in spite of crashes of nodes in the network. In the object-level case, the disappearance of one of the file-servers in the Sun network for several days had no noticeable effect on the objects replicated through use of R2D2; the resources were automatically regenerated on surviving file-servers. In the Replect case, PODs were deliberately switched on and off without decreasing availability of the replicated resources. (An advantage of the Eden system is that logically separate experimental versions of the kernels can be run simultaneously with the "official" version, interacting only through competition for memory, Ethernet time, and CPU cycles.)

To measure the change in availability due to replication would require either long-term observations with natural system loads, or use of synthetic loads chosen to represent particular classes of system use. Such studies are important for an understanding of *policies* for use of replication, to answer questions such as the number of copies to use, how they should be distributed and which objects should be replicated. We restricted our concerns to the *mechanisms* for replication, and made no attempt to make quantitative measurements of availability. A separate simulation study to examine availability and policy questions have been conducted and recently reported[22].

5.2. Transparency

The issues of transparency and performance are in conflict. From the transparency standpoint one would like to keep a consistent, single-name, user interface for replicated and non-replicated objects. One way to do this, whether working at object level or kernel level, is to treat the non-replicated case as a single-copy replicated case, but this implies burdening the non-replicated applications with the overhead of the replication mechanism. Another way is to use another layer of indirection that allows the system to differentiate between replicated and non-replicated objects and branch to code specialized for each case, while keeping the interface to the client uniform. This, too, introduces extra overhead. We can see ways to minimize this extra cost, but not to eliminate it; hence there remains some trade-off between transparency and performance.

We found this to be a more difficult trade-off at the object level than at the kernel level and while we would not yet claim that this is a general effect, we would suggest that the designer be wary. For example, in our object implementation, the client invokes the root of the R2D2 addressing system to translate a string name into a set of capabilities and to set up a transaction manager which deals with the required number of copies. In Eden, these extra invocations can cost around 0.2 seconds. In Eden without replication, a client accesses a directory with a string name to get a single capability for the desired object, thus avoiding the extra overhead of dealing with the transaction manager. Transaction control is not needed, since exclusive access to a single object can be handled by monitors within the object.

Using Replects, which give transparency because the client deals with a single capability for the resource, there still is a price to be paid: The kernel must keep track of the set of passive representations, and select one from which to create the active form that represents the replicated passive representations. One might assume that the user of a non-replicated Eden object could totally bypass this Replect overhead, but subtleties are involved: One would have to forbid conversion of a Replect to a single Eden object while the system was being used. Suppose access to a Replect was attempted just as it was being converted back to a non-replicated object. Suppose further that the conversion failed, in which case the object reverts to being a Replect, but in the meanwhile the object had been accessed and some action was taken on the assumption it was a non-replicated object. To avoid this one obviously would make the conversion under control of a transaction, but that implies that all activations of single objects would in some way have to be aware of this transaction mechanism, hence incurring some extra overhead. One could keep a flag on each passive representation, denoting it either a single object or a member of a Replect. The cost to a non-user of replication to check this flag could be kept small but it is still some finite amount of overhead.

More generally, replication requires binding multiple objects to a single name known to the clients and this implies a cost. Whether replication is defined at the *parent* level, as in R2D2, or the *sibling* level, as in Replects, a price must be paid for maintaining single-name transparency and preventing loss of the resource due to a single crash.

5.3. Performance

We believe that replication can be implemented at either kernel or object level without creating performance penalties for non-users of replication much beyond the normal competition for memory, transmission time, and CPU cycles incurred in any application or system facility. However as noted in the section above, this may call for giving up some aspects of transparency to the client, in the object-level case.

We restrict our conclusions drawn from the relative performance of the two implementations since for many reasons, as pointed out in preceding sections, they are not strictly comparable. Our object-level replicated directory was slower than the non-replicated directory during query operations by around 80%, but we believe this difference could be reduced to less than 10% by restructuring the R2D2 design. We do not see this difference as inherent between object-level and kernel-level implementations.

Some differences were observed in the updating case, with the object-level response time for two copies being longer by 50% than the non-replicated directory. More work is being done to understand the reasons for this delay despite concurrent updating of the two copies. Some of the difference may decrease for updates larger than the 1K bytes that were used, since the existence of multiple active forms in R2D2 provides an advantage over the Replect single active form during checkpoint. (For 1K bytes the contention for access to the single Replect active form by the multiple PODs writing to disk is negligible). The R2D2 performance may be affected not only by the extra invocations to the root and the transaction manager; their mere *presence* may be causing more paging since Eden objects are large. If that is the case there could be an inherent disadvantage in implementing at object-level in our existing Eden environment, but we would not suggest that this would be the case in object-oriented systems in general.

5.4. Implementation Experience

One of the important motivations for our studies in replication was to assess the strengths and weaknesses of the Eden system, a motivation shared with most of the applications programmed in the Eden environment. For replication at the object-level it was useful to have invocation be location independent, but to be able to bind objects to locations so that nodes with independent failure modes could be used. This is a feature that was not included in the original Eden implementation but was added shortly after its birth. In connection with this, the uniform use of capabilities for naming PODS, Hosts, and objects was useful in binding objects to particular parts of the distributed system. We had positive experience with other features of Eden such as object mobility, which was useful when the system configuration changed. The Eden Programming Language[18], and the fact that object structuring made it easy to define modules that seemed natural in the design of R2D2, both proved to be useful. Black made a good assessment of Eden features[17], and our experience generally supports his conclusions. Similarly, the slowness of the checkpoint operation, the lack of incremental logging to stable storage and the isolation from the disk that was imposed by building Eden on top of Unix caused high overhead in the replicated resources.

If the underlying system cannot handle partitions, it is difficult but not impossible to do so with replication systems built on top, by using voting to restrict updates to one portion, or using a version comparison mechanism during combination of partitions. The Eden system was not designed to handle partitions. If for a given Eden object the active form and the passive representation end up on separate sub-networks, the active form cannot transfer data to the passive representation to checkpoint. However, the passive representation (after suitable timeouts) can create a second active form. When the partitioned sub-nets are merged, the two active forms have to be dealt with. This is not a serious problem within a single Ethernet, due to the small probability of partitions, but can become a problem with gateways or with other types of communication hardware.

The two approaches to replication that we implemented vary in their ability to compensate for this inability to deal with partitions. This is due to interacting factors that are best discussed separately. First consider the general effect of the recovery method on a replicated system's response to partitioning. Using Regeneration, all partitions that contain a copy will be able to continue to operate if the partition contains an adequate number of operable machines. Then one would need methods to adjudicate among competing versions when partitions merge. If one uses voting methods, the partition that contained the required quorum would be able to operate, whereas the others would not. This would allow simple recombination of partitions. A second factor has to do with caching, represented in Eden by the object's active form. This does not affect the above description of partitioning using Regeneration. With Replects, using voting, it is possible that a partition might contain a quorum of passive representations but if the active form is gone it is not possible to tell if it is dead or in another partition. To be safe, one would not go ahead and create another active form without also having merging techniques to handle competing versions, since the isolated active form might be updated with no attempt to checkpoint. With R2D2, on the other hand, if one were to use voting methods the system would allow a partition containing a quorum to go ahead and operate since each of the replicas can have its individual active form as well as its passive representation.

Implementation at kernel level was more difficult than at object level. Again, no quantitative statement can be made, because two different implementations by two different persons working in different programming environments give no statistically significant data. However, the modularity, uniformity of invocation, and object location independence made the object-level implementation distinctly easier. *Design* efforts were comparable.

One would like not to implement transaction management twice in the system, once for replication and once as a general tool, and indeed our experience has shown that it is not necessary. Both of our replication implementations used concurrency control methods that were specializations of a more general approach, although only in the R2D2 case was this more general facility exported to the user level. This specialization allowed bypassing of the features necessary for user interface. In the Replect case further simplifications came about because of dealing only with replicas of a single object rather than keeping track of a list of objects.

We recommend provision of a general transaction manager tool with components of it specialized for use for replication control. However, the generally available tool should have user override so that application programmers have the option of either using the general tool or experimenting with special transaction techniques, making use of the semantics of the application.

In summary, kernel level implementation of replication appears preferable to object level if the following conditions hold:

1. Transparency to the client is important.
2. Upward compatibility is important. Applications might be written without replication followed by a later decision to replicate some or all of the objects involved, in which case one would like to do it with no change to the application's objects.
3. Kernel size is not an overriding issue.
4. System programming staff is available to deal with the additional complexity in writing the kernel.

Object level implementation of replication is preferable to kernel level if:

1. Kernel programmers are scarce compared to object programmers.
2. Faster implementation and easier maintenance are important. (All points related to programming are heavily influenced by the availability of a high level programming language for object programming. In the Eden case EPL provided a considerable advantage at object level compared to the use of C at the kernel level.)
3. Increased kernel size cannot be tolerated.

6. Observations on Related Work

Relative to the "new-wave" of object-oriented systems being designed and developed that incorporate various features of replication, crash recovery and transaction control, our experience leads us to comment on several of the efforts, to the extent we understand the work, much of which is just emerging in reports and papers.

The ISIS project[10] is designing replication and crash-recovery into the layers that provide and maintain the object level - i.e., what we have called the kernel-level approach. Apparently the client is not offered the option of *not* using these built-in fault tolerant features in order to improve performance. Similarly, the client is restricted to the built-in nested transaction mechanisms, although concurrent updating of replicated data (when using pessimistic concurrency control, gathering locks in advance) is provided to improve transaction performance. As noted above, we would vote for providing options in these areas, to encourage experimentation with transaction control mechanisms that take advantage of application semantics.

For replication of procedures, rather than of objects containing both data and code, Cooper's *replicated procedure call* mechanism[23] calls several replicated servers when one replicated procedure call is made; the call succeeds if just one of the servers remains functional. In the Eden case, objects include data and code. Both Replects and R2D2 provide a special case of replicated procedure call, the one-to-many call. For example, if a copy of R2D2 directory is inaccessible, the R2D2 transaction manager simply invokes the next copy in line. Cooper also supports many-to-many calls, in which several client processes make the same replicated procedure call.

The Clouds project[13] is building atomic transaction control into the kernel and developing a high level language (Aeolus) to make them available to users. The users will have the option of selecting the kernel's synchronization and recovery techniques or to develop their own, to take advantage of the semantics of the application. The emphasis in Clouds appears to be on *reliability* of results, in the go/no-go sense of atomic transactions, rather than *availability* in spite of crashes. The reference implies that studies of replication at a level above the kernel are being made.

There are several projects building support for transactions, such as Argus[24], TABS[11], and PROFEMO[12]. Argus implements nested transactions (as defined by Moss[25]) on top of Unix. TABS is built on Accent Kernel and supports distributed transactions. PROFEMO will build nested transactions into the operating system level, which will have hardware support for object management, crash recovery, and concurrency control. These systems do not support replication in the kernel, but provide powerful primitives to build replicated resources. For example, a replicated directory object is being built as an application server in TABS.

In summary, the results of our experiments in the Eden environment lead us to favor many of the features we see being included in this new wave of object-oriented designs and implementations, although as far as we have seen, no one of them encompasses all the features we feel are desirable. Our "wish list" for such features is the following:

1. Kernel support of replication, or support of primitives that make replication easy at object level. These include location-independent invocation, ability to pin an object to a location, mapping of a resource name into a set of object names, and a concurrency control method.
2. User choice of replication, including none at all.
3. Uniform interface to the system for the use of single or replicated objects.
4. Minimum performance penalty for users of replication.
5. Very little performance degradation for non-users.
6. Provision of high-level language for applications programming, including easy control of what is replicated and where copies are placed.
7. Export of a general-purpose transaction mechanism for use at the applications level.
8. Export of the underlying atomic actions that allow users to build their own transaction mechanisms to take advantage of application semantics.

7. Summary and Conclusions

The Eden distributed system has provided the means to implement two different approaches to object replication, one at the object level, and one in the kernel that supports the object environment. Both implementations succeeded in making resources available in spite of network node crashes.

The implementation at the kernel level was more difficult than at the object level, which also appears to be easier to maintain. Modularity and the programming environment seem responsible for this. A number of Eden features were very helpful at the object level; the nature of the objects, including the fact that they contain active processes; the high level language for programming objects; location-independent invocation; object mobility, and uniform naming of objects, Hosts and disk repositories.

Eden also had some shortcomings for our purposes. The slowness of checkpointing, and the lack of incremental logging to stable storage were handicaps at the object level, although logging was used at kernel level. The Eden system was not designed to handle partitions, and neither of the implementations overcame this limitation, although the experience shows us ways to do so. An R2D2 implementation using voting would allow a majority partition to operate; means for detecting and resolving updates on separate partitions would allow more general solutions.

There are trade-offs between performance and the transparency of access obtained by using a single name for a replicated resource. Problems arise in retaining single-name access, forcing either the use of a layer of indirection, or the system to broadcast to locate a copy. To hide such problems from the client and provide uniformity of access for either replicated or non-replicated resources implies a performance cost.

The choice of object or kernel level implementation depends on the relative importance of transparency, performance, implementation time, kernel complexity and size, as well as the relative ease of programming and the languages used at the two levels. We believe that replication can be implemented at either level with small and comparable performance penalties for the client not wishing to use replication. Our experiments lead us to favor inclusion of more functions in the kernel than Eden provides; either complete kernel support of replication or provision of a number of primitives that aid implementation of replication at the object-level.

8. Acknowledgments

The Eden system is the result of the work of many faculty, staff members, and graduate students whose efforts and insights have made the work possible. Andrew Black and Ed Lazowska made very helpful suggestions on drafts of this paper, although the authors are to be held responsible for the final result. In addition to our acknowledgment of the contributions of our colleagues, we appreciate the support of the Coordinated Experimental Research program of the National Science Foundation.

REFERENCES

1. Bartlett, J.F., "A NonStop Operating System", *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, December 1981, pp. 22-29.

2. Bernstein, P.A., Rothnie, J.B., Goodman, N., and Papadimitriou, C.A., "The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case)", *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 3, May 1978, pp. 154-168.

3. Lampson, B. and Sturgis, H., "Atomic Transactions", in *Distributed Systems -- Architecture and Implementation*, Springer-Verlag, Lecture Notes in Computer Science, Vol. 105, 1981.

4. Sturgis, H., Mitchell, J., and Israel, J., "Issues in the Design and Use of a Distributed File System", *ACM SIGOPS Operating Systems Review*, Vol. 14, No. 3, July 1980, pp. 55-69.

5. Dion, J., "The Cambridge File Server", *ACM SIGOPS Operating Systems Review*, Vol. 14, No. 4, October 1980, pp. 26-35.

6. Mitchell, J.G. and Dion, J., "A Comparison of Two Network-Based File Servers", *Communications of ACM*, Vol. 24, No. 4, April 1982, pp. 233-245.

7. Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, May 1979, pp. 188-194.

8. Walker, B., Popek, G., English, R., Kline, C., and Thiel, G., "The LOCUS Distributed Operating System", *Proceedings of Ninth Symposium on Operating Systems Principles*, ACM/SIGOPS, October 1983, pp. 49-70.

9. Haerder, T. and Reuter, A., "Principles of Transaction-Oriented Database Recovery", *ACM Computing Surveys*, Vol. 15, No. 4, December 1983, pp. 287-317.

10. Birman, Ken, "Replication and Fault-Tolerance in the ISIS System", *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985, pp. 79-86.

11. Spector, A.Z., Butcher, J., Daniels, D.S., Duchamp, D.J., Eppinger, J.L., Fineman, C.E., Heddaya, A., and Schwartz, P.M., "Support for Distributed Transactions in the TABS Prototype", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, June 1985, pp. 520-530.

12. Nett, E., "PROFEMO: Design and Implementation of a Fault-Tolerant Distributed System Architecture", Tech. report Draft, GMD-Studien, Nr. 100, GMD, Schloss Birlinghoven, 5205 St. Augustin 1, West Germany, June 1985.

13. LeBlanc, R.J. and Wilkes, C.T., "Systems Programming with Objects and Actions", Tech. report GIT-ICS-85/03, Department of Computer Science, Georgia Institute of Technology, March 1985.

14. Pu, C. and Noe, J.D., " Design and Implementation of Nested Transactions in Eden", Tech. report 85-12-03, Department of Computer Science, University of Washington,

February 1986.

15. Proudfoot, A., "Replects: data replication in the Eden System", Tech. report 85-12-04, Department of Computer Science, University of Washington, December 1985.

16. Almes, G.T., Black, A.P., Lazowska, E.D. Noe, J.D., "The Eden System: A Technical Review", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, January 1985, pp. 43-58.

17. Black, A.P., "Supporting Distributed Applications: Experience with Eden", *Proceedings of the Tenth Symposium on Operating Systems Principles*, ACM/SIGOPS, December 1985, pp. 181-193.

18. Black, A.P., "The Eden Programming Language", Tech. report 85-09-01, Department of Computer Science, University of Washington, September 1985.

19. Holt, R.C., *Concurrent Euclid, The Unix System, and Tunis*, Addison-Wesley Publishing Company, 1983.

20. Gifford, D.K., "Weighted Voting for Replicated Data", *Proceedings of the Seventh Symposium on Operating Systems Principles*, SIGOPS, ACM, December 1979.

21. Pu, Calton, Noe, Jerre D., and Proudfoot, Andy, "Regeneration of Replicated Objects: A Technique and Its Eden Implementation", *Proceedings of the Second International Conference on Data Engineering*, February 1986, pp. 175-187.

22. Noe, J.D. and Andreassian A., "Effectiveness of Replication in Distributed Computer Networks", Tech. report 86-06-05, Department of Computer Science, University of Washington, September 1986.

23. Cooper, E.C., "Replicated Distributed Programs", *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985, pp. 63-78.

24. Liskov, B.H. and Scheifler, R.W., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *Proceedings of 9th Annual ACM Symposium on Principles of Programming Languages*, January 1982, pp. 7-19.

25. Moss, J. Elliot B., *Nested Transactions: An Approach to Reliable Distributed Computing*, The MIT Press, Information Systems, Vol. 1, 1985.

MAYBE ALGEBRA Operators IN DATABASE MACHINE ARCHITECTURE

L.L. Miller

Department of Computer Science
Iowa State University
Ames, Iowa

A.R. Hurson

Dept. of Electrical and Comp. Engr.
The Pennsylvania State University
University Park, PA

Abstract

Changes in the user population continues the trend to more simplicity for the user and increased sophistication of the computer systems. In addition, advances in data storage technology have enforced the growth and creation of incomplete data files. A growing number of users require information from the database which cannot be obtained through the operators found in traditional query languages. Such needs have increased the expectation that a system should be able to assist the user in gathering and interpreting the appropriate data. The recent discussion of null values and maybe algebra is an attempt in this direction. However, the current literature has not addressed the practical issues of such theoretical concepts. This presentation examines the inclusion of the maybe algebra operators in a database machine. In addition, modifications designed to increase the time and space efficiency of the maybe algebra operators as well as the quality of the results are discussed.

1. Introduction

Incomplete information continues to be a problem for systems involved in information utilization. The problem of how to handle missing information has been studied by a number of researchers [5,6,8,9,16,17,18,26,27]. The current discussion of the use of the universal relation assumption (URA) [7,13,14,20,22,25], has increased the interest of dealing with null values.

A number of different uses for null values have been given in [1,22], but the problem of unknown data value represents the most common usage of null values. Codd [6] introduced a comprehensive extension to relational algebra that allows the user to examine potentially interesting data relationships based in part on unknown (incomplete) data. The theoretical foundations of Codd's new maybe algebra have been examined by Biskup [2].

While these works have developed a strong theoretical basis for maybe algebra, they have not taken into consideration the practical impact of such operations. In addition, recent developments in technology have led researchers to seek a hardware solution to the design of database systems, namely, the database machines [3]. Such machines have been developed with the purpose of improving the performance of the database operations. However, designers of such systems have not addressed many of the current issues that have surfaced recently in relational database design theory.

This presentation is an attempt to incorporate the concepts of null values and maybe algebra into the design of a relational database machine to extend its usefulness to the user population. Some practical restrictions of such an approach are also discussed. The database machine used as the basis of our discussion is the Associative search Language Machine (ASLM) [12]. The choice of ASLM stems from the fact that ASLM is highly parallel, and in many cases, the tuple search time is almost the same as relation search time (Associative Search).

The architecture of ASLM is briefly overviewed in Section 2. The basic concepts of maybe algebra are investigated in Section 3. The issues and practical considerations of implementing maybe algebra on ASLM are examined in Sections 4 and 5.

2. Database Machines

Several different classifications of database machines have been proposed. Rosenthal [23] has classified these machines as large backend, distributed network data node, and smart peripheral systems. Champine [4] has a similar classification using four classes: backend system, storage hierarchy, intelligent controllers and database computers. Su et. al. [24] have classified database machines as cellular logic systems, backend computers, integrated database machines and high speed associative memory systems. These classifications are similar in that for the most part they are based on how the database machine will be used. Bray and Freeman [3] have proposed a classification based on the concept of parallelism and the location where the data is searched. Based on this criteria, we have the following five groups: single processor indirect search, single processor direct search, multiple processor direct search, multiple processor indirect search and multiple processor combined search.

CH2345-7/86/0000/1210$01.00 © 1986 IEEE

These classifications suffer from the fact that there is some overlap between the categories. For example, there is no clear way to determine when a smart peripheral system becomes a backend computer as functions are moved from the host to the peripheral system.

However, one common feature of such classifications is the existence of a special purpose hardware system designed to manipulate the database. This approach is based on the need to remove the database functions from the host machine. The hardware specification of the database operations in the backend computer, the ability to overlap operations between the host and the backend, and elimination of the problems associated with the 90-10% rule [11] have made the backend approach more promising than the other categories. In the remainder of this section, the design of a backend database machine, ASLM, is discussed.

2.1 ASLM - a backend database machine

ASLM (Associative Search Language Machine) is a backend database machine [12]. The system is composed of a general purpose frontend machine supported by ASLM (Figure 1). The frontend system acts as the interface between the user and the backend machine. Security validation of the user and the user's query, translation of the user's query into ASL primitives and transmission of the final results to the user are the major functions of the frontend system.

Reducing the semantic gap and alleviating the transportation problem have been the general motivations behind the design of ASLM. Semantic gap reduction has been achieved through: i) the one to one relationship between the set operations and associative operations. ii) hardware implementation of the basic relational operations and iii) elimination of the address accessability of the data. The transportation problem has been resolved by screening the data close to the secondary storage. Solving these problems has determined the organization of ASLM.

ASLM is composed of four modules: i) controller, ii) secondary storage interface, iii) an array of preprocessors, and iv) a database processor.

Controller: The controller is a microprogrammable control unit. The contents of the writable control memory will be determined by the user's program. More specifically the controller: i) stores the ASL microinstructions generated by the ASL compiler, ii) decodes the ASL microinstructions, and iii) propagates the control sequences to the appropriate modules in the backend. The simplicity of the controller is primarily due to the use of associative hardware for the execution of the ASL primitives, since there is no need for address generation and the majority of the operations are strictly sequential.

Secondary Storage Interface: The secondary storage interface is a collection of random access memory modules, augmented by some hardware facilities. This module is an interface between secondary storage and ASLM. It accesses blocks of data from secondary storage and distributes them in a tuplewise fashion among the preprocessors.

Preprocessors: Because of the practical limitation on the size of the associative memory, it may not be possible to store all available data in the associative memory in order to perform an operation on the data items. Due to this restriction, all the systems proposed earlier based on fully associative memory, are not capable of handling large databases. In contrast, the design of ASLM overcomes this problem through consideration of two points. First, queries almost always refer to a subset of attributes in the tuples. Second, in each query, a small subset of tuples will satisfy the search criteria [11].

The preprocessors act as a filter which screens the data, selecting the valid data. The valid data is then placed in an associative stack in the database processor. In addition, the preprocessors perform a projection over the relevant attributes. In a database operation the attributes of a relation can be classified into three groups: i) members of the search argument (SA), ii) members of the output set (OS), and iii) the remaining attributes. It is the task of the preprocessors to validate tuples, on the basis of the SA attributes and to project them over the OS attributes.

The module is composed of an array of identical and independent processors which communicate with the database processor and the secondary storage interface. The preprocessors are initialized by the controller according to the query. After the preprocessors have been initialized, they perform operations on the tuples independent of any direct supervision from the controller.

Database Processor: The database processor is composed of a set of associative stacks enhanced by some hardware capabilities for direct implementation of the relational operators. The result of the operations of the array of preprocessors on a relation r with relation scheme R is a relation r_s with relation scheme R_s, which is stored in an associative stack.

$$R_s \leq R \text{ and } \forall t_s \; \varepsilon \; r_s \; \exists \; t \; \varepsilon \; r \text{ such that}$$

$$t_s = t[OS] \text{ and } t[SA] \equiv \text{search criteria.}$$

The associative stacks act as intermediate storage which holds part of the relation in order to perform the required relational operations on it. The database processor receives the database operations from the controller and performs them on the data stored in the appropriate associative stacks. At the end of the operations defined by

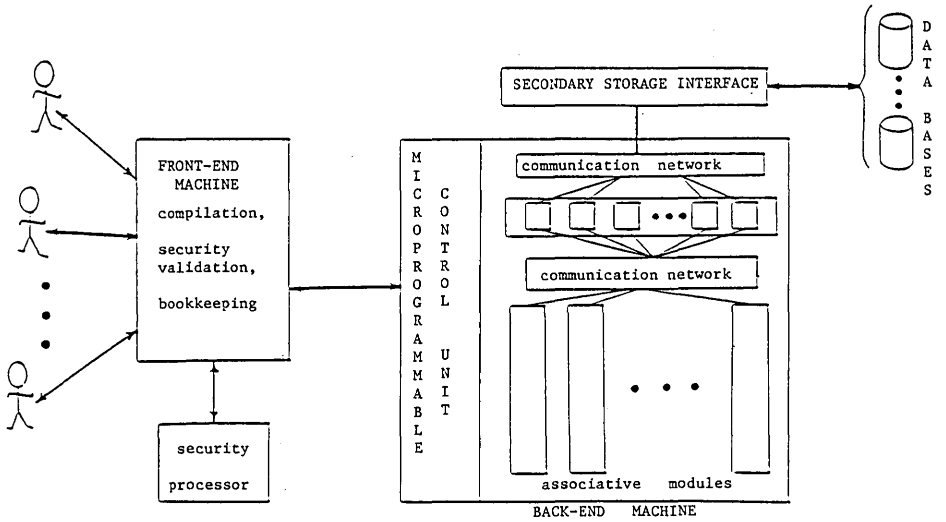


Figure 1: Overall Architecture of ASLM.

the query, the result is transferred to the general purpose computer.

The module is a set of identical and independent associative stacks of size w*d (where w is the width and d is the depth) augmented by some hardware circuits. The associative stacks can be linked together to make a memory size (K*w)*d ($1 \leq K \leq n$, where n is the number of associative modules) capable of holding d tuples of size K*w bits, or they could be linked to form a memory of size w*(K*d) capable of holding K*d tuples of size w bits. Because of the generality of the tuple sizes in a database, this facility enables the system to adjust the available memory modules based on the length of the tuples and the cardinality of the relation. In addition, this increases the space efficiency of the database processor. The independence of the associative modules enhances the modularity and as such the fault tolerance of the system.

Database processor is enhanced by a group of small special purpose modules to facilitate the direct implementation of the relational operators. One of these modules is used to reformat and realign tuples in the associative stacks. Such an action is necessary during some operations such as join, where the join attributes in the source tuples should be organized according to the position of join attribute(s) in the target tuples. The reformat unit is capable of performing basic logic operations on a collection of shift registers (4 units). The function of this unit will be discussed in more detail in Section 5.

ASLM is based on the concept of variable length tuples, where tuples and attribute fields are separated from each other by tuple separator markers and attribute separator markers, respectively. The null value is represented in this format as two adjacent attribute separator markers. The fields of a tuple t defined by t[OS] are expanded by the preprocessors to their predefined size in the normal manner. Fields containing null values are expanded to their appropriate size with don't care symbols.

The ASLM architecture is used in Section 4 to examine the implementation issues of Codd's maybe algebra [6]. The basic concepts of maybe algebra are examined in the next section.

3. Maybe Algebra

A great deal has been written about the relational model in recent years. We will assume that the reader has examined the basic concepts, in particular, relational algebra at the level of [26]. We will adopt Maier's notation [19] for partial information in the relational model. A partial tuple is one that contains zero or more null values ($\perp$). A tuple t is said to be total if it contains no null values. A tuple t whose scheme includes attribute A is definite on A (written $t(A){\downarrow}$) if t has a non-null value for A. We write $t(X){\downarrow}$ when $t(A){\downarrow}$ $\forall A \varepsilon X$ and $t{\downarrow}$ when t is total. If $t \varepsilon r$, where r is a relation defined by the relation scheme R, then $DEF(t) = \{A \mid A \varepsilon R \text{ and } t(A){\downarrow}\}$. A tuple t subsumes a tuple u ($t \geq u$) if $u(A){\downarrow}$ implies $u(A)=t(A)$. If $t \geq u$ and $t{\downarrow}$, then t is an extension of u ($t{\downarrow}\geq u$).

These concepts can be directly extended to relations as well. A relation r is total ($r{\downarrow}$) if all of its tuples are total. A relation r subsumes a relation s ($r \geq s$) if for every $t_s \varepsilon s$ there is a tuple $t_r \varepsilon r$ such that $t_r \geq t_s$. A relation r is an extension of s if $r \geq s$ and r is total. If r can be obtained from s by changing some null values in s to data values, then r augments s ($r \succeq s$). A relation r is a completion of s if $r \succeq s$ and $r{\downarrow}$.

Missing information presents a problem for any data model and as the relational model has matured, researchers have examined the question of how to handle missing data. Codd [6] developed an extension to relational algebra that allows a user to retrieve information based on partial results. Such a capability allows the user to probe the database for potential data relationships that might be useful to the user, but cannot be retrieved directly by true relational algebra. Recently, Biskup [2] has made maybe algebra more appealing by establishing its theoretical foundations.

A number of different interpretations of null values are possible [1,22], but maybe algebra is only concerned with missing or unknown data. Using this interpretation of null values, Codd [6] defines a 3-valued logic where

T - condition is true,
F - condition is false, and
ω - maybe - data values are unknown.

The truth tables for the boolean operations are given as:

AND	T	ω	F
T	T	ω	F
ω	ω	ω	F
F	F	F	F

OR	T	ω	F
T	T	T	T
ω	T	ω	ω
F	T	ω	F

NOT	
T	F
ω	ω
F	T

The null substitution principle [6] then can be stated as:

> A truth-valued expression has the value ω if and only if both of the following conditions hold:
>
> 1) Each occurrence of a null value in the expression can be replaced by a non-null value so as to yield the truth value T for the expression.
>
> 2) Each occurrence of a null value in the expression can be replaced by a non-null value so as to yield the truth value F for the expression.

Relational operators using a search condition to select tuples will base the search on a set of attributes, say X, that fall within a single relation scheme. Clearly, the selection of a tuple (t) based on a given condition will depend on the state of the values for t(X). If there exists A ε X such that $t(A)\downarrow$ and t(A) disagrees with the search condition, then the truth value associated with t is false (F). A truth value of true (T) would be generated for t if $t(X)\downarrow$ and t(X) matches the search argument. The truth value maybe (ω) is assigned if ∀A ε X such that $t(A)\downarrow$, the values t(A) match the search condition and there exists at least one element of X, say B, such that t(B) = ⊥.

Two operators that make use of a search condition to manipulate a relation are the select and join. The following examples illustrate the maybe and true relational algebra select, join and division operations.

r(R)

A	B	C
a_1	b_1	1
a_2	b_2	⊥
a_3	b_3	2

s(S)

C	D
1	d_1
⊥	d_2

a) True select r
where C=1

A	B	C
a_1	b_1	1

b) maybe select r
where C=1

A	B	C
a_2	b_2	⊥

c) True join of
r and s over C

A	B	C	C	D
a_1	b_1	1	1	d_1

d) maybe join of
r and s over C

A	B	C	C	D
a_1	b_1	1	⊥	d_2
a_2	b_2	⊥	⊥	d_2
a_3	b_3	2	⊥	d_2
a_2	b_2	⊥	1	d_1

Maybe division is treated in a similar manner as shown by the following example.

r(R)

A	B	C
a_1	b_1	c_1
a_2	b_1	c_1
a_1	b_2	c_2
a_2	b_2	⊥

s(S)

B	C
b_1	c_1
b_2	c_2

a) True division
($r \div s$)

A
a_1

Maybe division
($r \div_{\omega} s$)

A
a_2

A somewhat different view of the null value is taken for the remaining relational algebra operators. For these operations, null values in different tuples are interpreted as being equivalent. As such, we have no change in these operations over their use in traditional relational algebra. The following example illustrates this interpretation for the project operator.

r(R)

A	B	C
a_1	⊥	c_1
a_1	⊥	c_2
a_2	b_2	⊥
a_3	b_3	c_3

Project r over A and B

A	B
a_1	⊥
a_2	b_2
a_3	b_3

Codd [6] has also addressed the issue of using null values to bring the so called dangling tuples into the join process and to make relations union compatable. The outer theta join [6] will be used here to motivate the concept. Similar join operations have been proposed by Merrett [21], Zanilo [29] and LaCroix and Pirotte [15].

The outer theta join (denoted by $\textcircled{\theta}$) for r(A,B) and s(C,D) over BθC (i.e., B and C are defined over the same domain and θ is the test condition) is defined by Codd as:

$$r \circledcirc s = T \cup (R_1 X(C{:}\bot, D{:}\bot)) \cup ((A{:}\bot, B{:}\bot) X S_1)$$

T = the true join of R and S over $B\theta C$
$R_1 = r - T(AB)$ $S_1 = s - T(CD)$.
(where θ is equality)

r	A	B
	a_1	b_1
	a_2	b_2
	a_3	b_3

s	C	D
	b_1	d_1
	b_2	d_2
	b_4	d_3

T	A	B	C	D
	a_1	b_1	b_1	d_1
	a_2	b_2	b_2	d_2

R_1	A	B
	a_3	b_3

S_1	C	D
	b_4	d_3

$r \circledcirc s$	A	B	C	D
	a_1	b_1	b_1	d_1
	a_2	b_2	b_2	d_2
	a_3	b_3	$\bot$	$\bot$
	$\bot$	$\bot$	b_4	d_3

If we think of the traditional relational algebra as true (T) algebra, then we can view Codd's maybe algebra as retrieving tuples with unknown critical values. The search conditions for such tuples evaluate to maybe (ω). Since the subset of the search condition attributes that is definite matches the search criteria and the remainder are null, it is unclear whether or not such tuples are of use to the user. Maybe algebra gives the user the opportunity to investigate the set of tuples and draw his/her own conclusions. The impact of implementing the maybe algebra operators in the ASLM environment is examined in Section 4.

4. Implementing Maybe Algebra in ASLM

In this section, the ASLM implementation of maybe algebra is examined. We ignore the obvious changes required to the query language and concentrate on the compilation process and the architectural changes required to add the maybe algebra operators to ASLM.

i) The null value is simply taken as a data value by relational operators such as union, intersection, difference and projection. Such an interpretation of null values means there is no change in the implementation of the operators over traditional relational algebra. The details of these operators can be found in [12].

ii) The result of a maybe select operation is the set of tuples that yield the maybe truth value for the select condition under Codd's null value substitution principle [6]. A query of the form:

maybe select from r where
A = "a" AND B = "b" AND C = "c"
can be written as

select from r where
(A = "a" OR A = $\bot$) AND (B = "b" OR B = $\bot$)
AND (C = "c" OR C = $\bot$) AND (A = $\bot$ OR B = $\bot$
OR C = $\bot$).

In the event the condition is defined over a single attribute, the result in ASLM can be determined by an equality search against $\bot$. For more general search conditions, the maybe select can be implemented in the database processor as a sequence of 2^d-1 select operations, where d is the number of definite attributes used in the search condition. The sequence of operations will be determined by the compiler and hence will be passed to the database processor prior to execution. The results of the selects are placed together on the resultant associative stack. The operation

Maybe select from r where a<1 AND b>2
would require

select from r where a<1 AND b=$\bot$;
select from r where a=$\bot$ AND b>2;
select from r where a=$\bot$ AND b=$\bot$.

Such an approach means that the maybe select operator can be implemented directly in the current design of ASLM. No architectural changes are required to implement the operator. Since the search condition is introduced into the process by the user's query, the 2^d-1 select conditions required can be generated directly by the compiler. Naturally, an increase on the number of searches reduces the performance of the maybe select operator. However, since no changes have been introduced into the design of ASLM, no degradation of performance of the true algebra operators will be experienced.

iii) The maybe join·operation places a tuple in the result relation when the test for a join between two tuples results in a maybe truth value. For each tuple in r_1 (source relation), the relation r_2 (target relation) is search 2^d-1 times, where d is the number of definite attributes in the join set. As in the maybe select, the searches are looking for tuples in r_2 with different combinations of the definite attributes of the join set that contain one or more null values.

In the ASLM design, the join of relations r_1 and r_2 is initiated by loading the relevant data from each relation into two associative stacks. The tuples from the source relation are popped one at a time and passed to the reformat module. The reformat module realigns the join set attributes to conform to the positions they have in the target relation. Then, for each reformatted set of attribute values the target relation is searched in associative fashion. In our new environment, compiler distinguishes the true join and the maybe join during the compilation. For true join, additional select operations will be initiated to exclude all the tuples in the source and target relations which have at least one null value in their join set attributes. For the maybe join, the reformat module is triggered to enumerate all possible permutations of the join set attributes in the source relation tuple (i.e. $2^{d}-1$ cases as above). This enhancement to the reformat module is the only hardware overhead of the maybe algebra over the original design of the system.

maybe join r_1 and r_2 ($A_1 = A_2$ and $B_1 = B_2$)

r_1	$(A_1$	B_1	$C)$
	a_1	$\perp$	c_1
	a_2	b_2	c_2

r_2	$(A_2$	B_2	$D)$
	a_1	b_1	d_1
	a_2	$\perp$	d_2
	a_2	b_2	d_3
	$\perp$	b_2	d_4

maybe join r_1 and r_2

A_1	A_2	B_1	B_2	C	D
a_1	a_1	$\perp$	b_1	c_1	d_1
a_1	$\perp$	$\perp$	b_2	c_1	d_4
a_2	a_2	b_2	$\perp$	c_2	d_2
a_2	$\perp$	b_2	b_2	c_2	d_4

Maybe join has the potential to have drastic effects on the resource utilization and the performance of a database system. For example, if we wish to use the maybe join to join r_1 and r_2 over A = B, where R_1 and R_2 are the respective schemes and $A \varepsilon R_1$ and $B \varepsilon R_2$, then we get one copy of r_2 for each tuple t_1 in r_1 where $t_1(A)$ is null. Similarly, we get a copy of r_1 for each tuple t_2 in r_2 where $t_2(B)$ is null. We can find the size of the maybe join result for this example by:

$m*n_1 + n*m_1 - m_1*n_1$
where
$m = |r_1|$, $n = |r_2|$ and
n_1 = the number of nulls in r_2 for B
m_1 = the number of nulls in r_1 for A.

The potential for drastic loss of performance is apparent from the following example.

Example: Suppose m=n=1000
with 1% nulls for A and B
1000*10 + 1000*10 - 10*10 = 19,900
tuples in the maybe join result.

with 5% nulls for A and B
1000*50 + 100*50 - 50*50 = 97,500 tuples
in the maybe join result.

with 10% nulls for A and B
1000*100 + 100*100 - 100*100 = 190,000
tuples in the maybe join result.

Interestingly, the loss in physical performance is paralleled by a loss in logical performance. The value of such an operator must be rated by its ability to supply a "useable" set of tuples which the user can examine to see the potential relationships between data values. The relation sizes suggested by the previous example fall beyond the useable category. Zaniolo's generalized join removes the join over two nulls, but still has the potential to produce extremely large relations [29].

If we consider the probability of a relationship appearing in the maybe join result, given that one or more of the join set attributes are unknown, we can consider the information content of the operation by using Shannon's formula [10]

$$H = - \sum_{i=1}^{n} p_i \log_2 p_i$$

where P_i is the probability of a configuration's occurence.

Consider the case where the join set is a singleton attribute. When a tuple from one of the relations has a null value in the join attribute, we have a probability of one that the tuple will be joined to all of the tuples in the second relation. Using the interpretation from information theory, we can say that it is possible to guess the set of relationships produced by the maybe join and as such the information value of the operation is zero. From the database point of view, this means that we can use other operators that require significantly less resources to produce the same result.

Based on this discussion, we present a practical restriction on the maybe join operator. In its current form, the operator will likely cause stack overflow in ASLM when either the number of attributes in the join set is small or the two relations have a high percentage of null values for the join attributes. To deal with this problem, two solutions are being incorporated into the ASLM design.

a) Maybe join over a single attribute is detected by the ASL compiler and ignored. The user is notified that such an operation is expected to result in a low information result.

b) During the maybe join operation on two or more attributes, the system will be able to detect that a large percentage of nulls are resulting in a large number of maybe joined tuples. When the stack containing the maybe join result reaches a predefined load density, the operation is terminated and the user is notified of the low information quality of the data. By using the relative sizes of the two relations involved in the join, the system can set the predefined load density for the stack in question.

The use of these two restrictions will result in higher information value results. The first decision can be made during the compilation while the second decision is determined during the execution time. This can be done by the reformat module which controls the sequence of the maybe join operations.

iv) The maybe division operator can be implemented using the operators previously defined. The following algorithm provides the sequences of operations for r(R) maybe ÷ s(S) in ASLM:

```
Algorithm maybe divide;
begin
(maybe join r(R) and {$t_1$ ε s(S)}) U (join r(R)
and {$t_1$ ε s(S)});
  place result in stack $S_1$;
  project $S_1$ over R-S;
i=2;
while there is an unprocessed tuple in s do
begin
(maybe join r(R) and {$t_i$ ε s(S)}) U (join r(R)
and {$t_i$ ε s(S)});
  place result in stack $S_2$;
  project $S_2$ over R-S;
  intersect $S_1$ and $S_2$ placing the result in $S_1$;
  i:=i+1
end
Place r÷s on stack $S_2$;
Place $S_1$-$S_2$ on stack $S_1$
end.
```

Example

r(A	B	C)
a_1	b_1	⊥
a_2	b_2	c_2
a_1	b_2	c_2
a_2	b_1	c_1
a_3	b_1	c_1

s(B	C)
b_1	c_1
b_2	c_2

Stack S_1	a_1	Stack S_2	a_1	$S_1 \wedge S_2$	a_1
after	a_2	after	a_2		a_2
projection	a_3	projection			
over R-S		over R-S			

Stack S_2 r÷s	a_2	S_1-S_2	a_1

The maybe divide operation is extremely slow, but does not require any additional hardware changes beyond the changes required to implement the maybe join operator. As such, it offers no further degradation of the performance of the true algebra operators.

v. The outer join operation is designed to bring the so called dangling tuples into the join process. When the relations r_1 and r_2 are joined with the outer join, we have each dangling tuple of r_1 extended with null values for the attributes defined for r_2 that do not appear in r_1. Similarly, the dangling tuples from r_2 are extended with null values to make them union compatible with $r_1 ⟗ r_2$.

In ASLM, the outer join can be performed by testing both relations. In the traditional join operation, dangling tuples in r_1 are encountered by the process when the tuples are tested for a match in r_2. The outer join simply requires extending the tuples and placing them in the appropriate stack. On the other hand, the dangling tuples in r_2 are not located by the present join operation. To do this, it requires using the tuples of r_2 to test whether or not a tuple is a dangling tuple. When a dangling tuple is encountered, it is extended with null values and placed in the stack.

r_1 (A	B	C)
a_1	b_1	c_1
a_2	⊥	c_2
a_3	b_3	c_3

r_2(A	D	E)
a_2	d_2	e_2
a_4	d_4	e_4

using the outer join of r_1 and r_2 in ASLM, we have

A	B	C	D	E
a_1	b_1	c_1	⊥	⊥
a_2	⊥	c_2	d_2	e_2
a_3	b_3	c_3	⊥	⊥

after the first step.

Searching the result for a_2 @ @ d_2 e_2 (where @ means the fields are masked out on the search), a match tells us that the tuple participated in the join. On the other hand, searching for a_4 @ @ d_4 e_4, we find that the tuple needs to be extended and added to the stack. The final result of the outer join would be

A	B	C	D	E
a_1	b_1	c_1	⊥	⊥
a_2	⊥	c_2	d_2	e_2
a_3	b_3	c_3	⊥	⊥
a_4	⊥	⊥	d_4	e_4

5. Maintaining the database

The use of null values in a database system increases the complexity of the insertion policy (i.e., concepts such as tuple subsumption and null replacement as well as traditional concerns like duplicates must be considered). Recent literature has addressed a wide variety of such techniques [19]. Creating an insertion policy from the available techniques requires establishing the assumptions on how the system will be used. Our approach is to use somewhat of a compromise of the available techniques based on the following assumptions:

i) A new tuple can be inserted into an ASLM relation if it is not already subsumed by a current tuple.

ii) There are no hard violations of the database semanties in the current copy of the database.

The first assumption means that if a tuple such as $t_1 = < a_1 \perp \perp \perp e_1 >$ currently exists in an ASLM relation with scheme $\{A, B, C, D, E\}$ then the insertion of a new tuple $t_2 = < a_1 \perp c_1 \perp e_1 >$ means that the tuple $t_1 = < a_1 \perp \perp \perp e_1 >$ is no longer needed since $t_2 \geq t_1$. Let $s = \{t \mid t \varepsilon r$ and $t_2 \geq t\}$, the relation s can be found by applying a variation of the maybe select over the relation scheme for t_2. For example, the query for t_2 would be:

Select from r where
  (A = a_1 or A = $\perp$) AND (B = $\perp$) AND (C = c_1 OR
  C = $\perp$) AND (D = $\perp$) AND (E = e_1 OR E = $\perp$).

The resulting relation, $r' = (r-s) \cup \{t_2\}$, subsumes the relation r. The relation r does have a larger possibility set, but the completions of r' will all appear in the possibility set of r.

The second assumption allows the tuples in the current relation to be used to determine the true value of null values in the insert tuple, based on the existing set of functional dependencies. For example, given the existence of the functional dependency A→B, a query such as:

Select from r where A = a_1

would generate the set of tuples with the same A value. The system can then test the resulting set of tuples for a non-null value of B. Based on assumption ii) if such a value exists, there should be only one such value (ignoring null values which are considered soft violations). If such a non-null value exists, the backend system can then use the value to replace the null value in t_2 for the attribute B.

Null value replacement as discussed can be extended beyond just changing insertion tuple. The definite values of the insertion tuple can be used to replace null values in the tuples of the current relation. Such a policy is clearly more time consuming, since, first it requires that the user interface generates test conditions for all functional dependencies for which the left side attributes of the insertion tuple are definite and then the definite right side attributes in the insertion tuple are used to replace nulls in the tuples of the current relation. The policy rests on the assumption that the insertion tuple is valid. Notice that such an assumption is consistent with assumption (ii). Inserting a tuple that has an incorrect data value will result in an incorrect relation.

The null replacement policy used in ASLM will not provide replacement of all of the nulls that might be replaced under a more comprehensive replacement policy such as the one described by Maier [19]. The more comprehensive replacement algorithms, such as the one described by Maier, make use of marked (indexed) null values to increase the situations where nulls can be replaced. In considering the overall performance of ASLM, the use of marked nulls does not seem to offer sufficient advantages to offset the increased cost.

6. Conclusion

The inclusion of the maybe algebra operators into the design of the Associative Search Language Machine (ASLM) has been examined. It is shown that the system can be upgraded to include the operators with only modest changes. Practical restrictions for the implementation of the maybe join have been given.

The primary concern in adapting ASLM to include the maybe algebra operators, is that any required changes should not degrade the performance of the traditional relational algebra operators. To this end, the implementation of maybe algebra is slow. Current work centers around attempts to solve this problem.

Reference List

1. ANSI/X3/SPARC Study Group on Database Management Systems, Interim Report, ANSI, Feb., 1975.

2. Biskup, J., "A Foundation of Codd's Relational Maybe-Operations," ACM TODS, Vol. 8, No. 4, 1983, pp. 608-636.

3. Bray, O.H. and H.A. Freeman, Database Computers, Lexington Books, 1979.

4. Champine, G.A., "Four Approaches to a Database Computer", Datamation, December 1978, pp. 101-106.

5. Codd, E.F., "Understanding Relations", FDT, 7:3-4, Dec. 1975, pp. 23-28.

6. Codd, E.F., "Extending the Database Relational Model to Capture More Meaning", ACM TODS, Vol. 4, No. 4, 1979, pp. 397-434.

7. Fagin, R., A.O. Mendelzon, and J. Ullman, "A Simplified Universal Relation Assumption and its Properties", ACM TODS, vol. 7, No. 3, 1982, pp. 343-360.

8. Grant, J., "Null Values in a Relational Database", Information Processing Letters, Oct. 1977, pp. 156-157.

9. Grant, J. "Partial Values in a Tabular Database Model", Information Processing Letters, August 1979, pp. 97-99.

10. Heaps, H.S., Information Retrieval: Computational and Theoretical Aspects, Academic Press, New York, 1978.

11. Hsiao, D.K., "Database Computers," Advances in Computers, 1980, pp. 1-64.

12. Hurson, A.R., An Associative Backend Machine for Database Management, Ph.D. Dissertation, University of Central Florida, 1980.

13. Kent, W., "Consequences of Assuming a Universal Relation," ACM TODS, vol. 6, No. 4, 1981, pp. 539-556.

14. Korth, H., G.M. Kuper, JU. Feigenbaum, A. Van Gelder and J. Ullman, "System/U:A Database System Based on the Universal Relation Assumption, ACM TODS, vol. 9, No. 3, 1984, pp. 331-347.

15. LaCroix, M. and A. Pirotte, "Generalized Joins", ACM SIGMOD Record, Sept. 1976, pp. 14-15.

16. Lien, Y.E., "Multivalued Dependencies with Null Values in Relational Databases," VLDB, Rio the Janeiro, Brazil, 1979, pp. 61-66.

17. Lipski, W., "On Semantic Issues Connected with Incomplete Information Databases", ACM TODS, vol. 4, No. 3, 1979, pp. 262-296.

18. Lipski, W., "On Databases with Incomplete Information", JACM, vol. 28, No. 1, 1981, pp. 41-47.

19. Maier, D., The Theory of Relational Databases, Computer Science Press, Rockville, Maryland, 1983.

20. Maier, D., J. Ullman, and M. Vardi, "On the Foundations of the Universal Relation Model", ACM TODS, vol, 9, No. 2, pp. 283-308.

21. Merrett, T.H., "Relations as Programming Language Elements", Information Processing Letters, vol. 6, No. 1, 1977, pp. 29-33.

22. Parker, D.S. and P. Atzeni, "Assumptions in Relational Database Theory," Proceedings of the ACM Symposium on Principles of Database Systems, Mar. 1982, Los Angeles, pp. 1-9.

23. Rosenthal, R.S., "The Data Management Machine, A Classification", Third Workshop on Computer Architecture for Non-numeric Processing, 1977.

24. Su, S.Y.W., H. Chang, G. Copeland, P. Fisher, E. Lowenthal, and S. Schuster, "Database Machines and Some Issues on DBMS Standards", National Computer Conference, 1980, pp. 191-208.

25. Ullman, J., "The U.R. Strikes Back", Proceedings of the ACM Symprosium on Principles of Database Systems, Mar. 1982, Los Angeles, pp. 10-22.

26. Ullman, J., Principles of Database Systems, (Second Edition), Computer Science Press, Rockville, Maryland, 1982.

27. Vassilion, Y., "Null Values in Database Management Denotational Semantics Approach," ACM SIGMOD Conference, 1979, pp. 162-169.

28. Vassilion, Y., "Functional Dependencies and Incomplete Information," VLDB, Montreal, 1980, pp. 260-269.

29. Zaniolo, C., "Relational Views in a Database System: Support for Queries", IEEE COMPSAC, 1977, pp. 267-275.

PROS AND CONS OF OPERATING SYSTEM TRANSACTIONS FOR DATA BASE SYSTEMS

Gerhard Weikum

Computer Science Department
Technical University of Darmstadt
D-6100 Darmstadt, West Germany

Abstract

Recently, several proposals have been made to integrate the well-known notion of transaction into operating systems. A general transaction service could serve as a common basis for various client types, including data base systems. In this paper, benefits as well as drawbacks of such an operating system facility are discussed. A major contribution is to investigate the suitability of three basic kinds of DBS server concepts with regard to the utilization of OS transactions. To overcome some of the problems of building DBS transactions on top of OS transactions, we propose a multi-level transaction methodology. Single DBS requests are handled as transactions by the OS layer. Management of entire DBS transactions is done in the DBS supported by basic OS services. This multi-level transaction management method seems to have certain performance advantages over pure OS transactions.

1. The Notion of Transaction in Operating Systems

The notion of transaction has proven to be one of the key concepts of modern data base systems (DBS). It enables application programmers to overlook all effects of multi-user mode as well as the possibility of certain classes of failures including a system crash. The DBS transaction manager guarantees that each transaction appears isolated from concurrent executions and ensures atomicity and persistence of transactions [21]. Application programs simply have to indicate begin and end of a transaction, usually denoted BOT, standing for 'Begin of Transaction', and EOT, standing for 'End of Transaction', or RBT, standing for 'Rollback Transaction', in the case of a deliberate abort. Thus, transactions are an easy to use concept for fault tolerance in a multi-user environment [15].

At the moment, many people believe that transactions need not necessarily be tied to the framework of data base systems, but could be useful for other types of applications too [36]. This idea has given rise to various approaches aiming at a general transaction management service within the operating system (OS) as a common basis for all kinds of applications. A number of projects on this issue, both in the context of commercial systems (e.g. [38], [41], [46], [51]) and focussing on research prototypes (e.g. [8], [27], [28], [32], [37]), are in an advanced stage or have already been finished. Basically, all designs suggest an architecture similar to that of figure 1. Besides a DBS, possible clients of the OS transaction manager could be a DC-System managing a network of terminals (cf. [19], [45]), file servers in a distributed environment [43] or even mail service as a component of advanced office information systems [47].

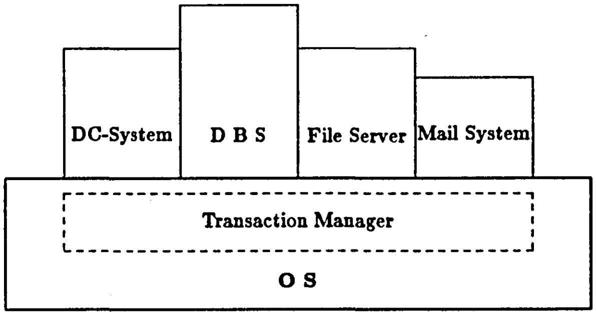


Fig.1: System architecture based on OS transactions

The above architecture seems to be a very promising direction for future OS development. On the other side, today's DBSs often ignore OS services or even circumvent the OS interface because of performance reasons, thus reimplementing functions like buffer management and scheduling [39]. Sometimes DBSs even use special channel programs to optimize critical I/O requests [29]. Usually, the argument is that general-purpose OS facilities cannot meet the requirements of a DBS, which for example has to schedule thousands of transactions in a very short time without the overhead of OS processes. The crucial question arising here is whether an OS transaction management service is suitable for DBSs, also w.r.t. performance. To be more precise, we actually should ask whether there is a good chance to implement OS transactions efficiently enough without semantic knowledge about data and DBS operations. In the case of a negative answer, DBSs are likely to ignore OS functions once more, keeping their own transaction mechanisms (cf. [41]).

This paper discusses benefits of as well as objections to OS transactions from a DBS point of view. One major issue is the correlation between transactions and the DBS process structure. Since, in most proposals, a process can run OS transactions only one after the other, difficulties to utilize this new OS feature arise when a single DBS process or a small group of DBS server processes performs the scheduling of transactions. On the other side, having done all the scheduling by the OS would require one DBS process per transaction which seems not to be feasible in common OSs due to high process management costs. We point out an efficient solution based on a multi-level approach.

The paper is structured as follows. Section 2 considers basic OS services supporting DBS transactions. The emphasis of section 3 is on drawbacks of a direct mapping of DBS transactions to OS transactions. A different kind of mapping is presented in section 4 taking into consideration the process structure of a DBS. Design considerations for the proposed multi-level transaction approach as well as preliminary performance estimations are given in section 5. We conclude with the recommendation of a revised system architecture on the basis of figure 1.

CH2345-7/86/0000/1219$01.00 © 1986 IEEE

2. Operating System Services for DBS Transactions

A first step towards an OS transaction management service would be to provide at the OS level basic functions for both compoments of a DBS transaction manager, concurrency control and recovery. As concurrency control is mostly implemented by locking and recovery is usually based on logging or shadow storage, this means that two kinds of services should be offered: lock management and a facility to backout and redo actions. Implications of such OS functions for a DBS transaction manager are discussed in this section. We make no attempt to give an exhaustive survey on various systems or proposals, which usually are very hard to compare with each other, but restrict ourselves to a discussion of typical issues.

2.1 Locking Services

State-of-the-art OSs provide functions that allow processes to operate on global resources in a synchronized way (e.g. [9]). These functions are:

LOCK < *resource name* > < *mode* >
UNLOCK < *resource name* > < *mode* >

At least two lock modes, shared access and exclusive access, should be supported. Additionally, special modes may be useful for hierarchical locking protocols [13]. When a process wants to acquire a lock on a resource that is already locked by a different process in an incompatible mode, the process is enqueued on a FIFO waiting list and deactivated. It is made dispatchable again as soon as the lock can be granted.

All this is essentially the very same as in DBS concurrency control. At a second glance, however, some subtle differences become apparent. OS lockable resources must have a unique name upon which all processes agree. In a DBS, most resources have unique names such as page numbers or tuple identifiers, but also proposals have been made to lock predicates or key values and key intervals in an index (e.g. [5]). If, for example, a reader scans the key interval [10,300], then, in order to achieve serializability, a writer inserting key 120 must be prevented from running concurrently. To recognize such a situation, both the interval and the single key have to be mapped to the same resource name. Thus, besides the OS locking service such a DBS would need some table management of its own. However, as most DBSs use page locking, this seems not to be really a serious disadvantage of OS lock management. Even the widely discussed phantom problem can be handled properly by page locking, since predicate oriented operations are inevitably transformed into either an index scan or some kind of relation scan on an implementation level (cf. [2]).

More serious is that building upon OS locking solely requires an OS kernel call, i.e. an SVC in the common /370 architecture, for each LOCK operation, even in the nowaiting case. If instead locks were managed in a common memory pool by the DBS itself, this overhead can be avoided whenever a lock request is granted immediately. As far as we know, no figures about this specific SVC overhead have ever been published. We are convinced, however, that, by microcoding frequent instruction sequences or with a better design of hardware, the costs of OS locking service functions can be reduced by an order of magnitude compared with today's figures (cf. [23], [31]). In the common case of (virtual memory) page locking, special hardware support has been suggested [40]. The proposed hardware automatically marks a virtual memory page as locked on the first machine instruction referencing this page, either in exclusive or shared mode depending on the kind of reference. This method seems to be practicable provided that locked pages remain in virtual memory, i.e. the OS has a 'one-level storage' file system [44], this is what the method originally was proposed for, or locked pages always could be pinned to the DBS buffer pool. Please note that this approach would require an explicit OS kernel call only for UNLOCK operations.

OS locking services can be used by DBSs in two ways. Most DBSs need a small number of semaphores (sometimes called 'latches') to synchronize access to common adminstration pools such as the lock table or the buffer pool LRU chain. These semaphores are typically acquired very often and released again after some hundred machine instructions. When a DBS process is deactivated by the OS scheduler, e.g. due to time-slice runout, while holding such a semaphore, all other processes soon enqueue on this semaphore, thus forming a kind of convoy. It has been observed that convois are a lasting phenomenon and hence have a very negative influence on the overall performance [3]. If DBS semaphores were known to the OS, the OS scheduler could easily avoid deactivating a process that would block other processes. Semaphores should be implemented by the OS locking service therefore.

The second way to use OS locking is for data items. This seems to be the method of the relational system INGRES to implement page locking [30]. We have already discussed inherent limitations of this approach above. On the other side, two advantages over DBS locking can be imagined. As the OS has knowledge about CPU utilization, it could try to run a deadlock detection 'demon', i.e. to check for cycles in a waiting-graph, whenever the CPU would be idle otherwise. Such an approach, of course, is not feasible for a DBS, that would have to start deadlock detection in regular intervals, or, even worse, at each lock wait. In addition to this, we observe that global deadlocks, e.g. involving data base locks and file locks, can be easily detected if resource locking is performed entirely in one subsystem, viz. the OS.

Another benefit of OS locking is simplification and potential performance improvement of resource unlocking. When the DBS executes an EOT operation, it has to release all locks held by the committed transaction, and has to notify all transactions that have been waiting for one of these locks and are now ready to run. The second step usually is a costly event, since all DBS processes acting on behalf of these transactions must be signaled via OS kernel calls. This requires as many interprocess messages as there are waiting processes. Probably, it would be much more efficient, if a process committing a transaction calls the OS locking service once, leaving it to the OS to make waiting processes dispatchable again. We would simply have to extend the UNLOCK operation to release a set of resources in one call.

It seems that OS locking would not only be far from being a bottleneck for DBS transactions, but could really improve the overall performance.

2.2 Recovery Services

The purpose of an OS recovery service is twofold. First, it provides clients with the facility to go back deliberately to a previous system state, either by reconstructing the state of all objects that have been changed since then or by compensating all update operations through their inverses. The second task is to secure object states, i.e. to protect them against crashes by forcing all necessary data to stable storage.

These two facilities are called UNDO and REDO functions in DBS terms. As in DBSs, rollback need not be expected to any previous state nor need resiliency be guaranteed for every single action. For most clients it seems to be sufficient, if states to be secured and backup states must be indicated explicitly. A special form of this facility is the notion of a 'recovery sphere' which is a unit of work with the properties of atomicity and persistence. Recovery spheres enable a client to go back to a single previous state, marked by a 'BRU' call denoting 'Begin of Recovery Unit', until it reaches a commit point, marked by an 'ERU' call denoting 'End of Recovery Unit' [46]. Only committed states are secured.

Recovery spheres closely resemble the notion of transaction. The difference is that the former need not be synchronized with each other. Thus, recovery spheres plus strict two-phase locking provide an OS transaction service. In the following, we exclude nested recovery spheres [26, 27] from our discussion.

Obviously, an OS service providing recovery spheres can adopt DBS recovery techniques which mostly are based on shadow storage or logging. The basic idea of shadow storage is to map updated pages, i.e. current but not yet committed page versions, to disk blocks different from those of the original, i.e. old page versions which become so-called 'shadow pages' [44]. This mapping usually is done through two versions of a translation table. The necessity of additional I/Os for these page tables has been considered as a main bottleneck of shadow storage mechanisms [14].

Integration of shadow storage into a virtual memory OS could help to reduce the table management overhead drastically. As virtual address spaces typically are based on a page-to-disk block mapping too, it seems very advantageous to combine both kinds of tables (e.g. [8]). Please note, however, that this would require a complete file to be bound in virtual memory, which is the basic idea of 'one-level storage' systems (cf. [44]). No DBS buffer pool is necessary in such an approach, as all I/Os are done via virtual memory paging. Thus, the problem of double-page faults [10] is resolved incidentally. One point of criticism of this technique has been that virtual memory replacement algorithms, usually LRU-like, are not suitable for DBS page reference patterns [39]. We leave it to the reader's judgement whether this issue could be remedied or rather is a persistent objection.

Even with a more conventional file system in mind, the OS could decrease shadow storage table overhead combining it with the mapping of extents, i.e. collections of consecutive disk blocks, to disk addresses. This would at least help to shorten page table entries, and consequently saves I/Os (cf. [24], [32]).

The other main approach to recovery - logging - requires an UNDO log record to be written each time before updating a disk page in place and REDO log records to be written before committing a transaction [13]. The first kind of I/Os due to the write-ahead log principle often can be avoided when buffer pools are large enough to fix updated pages until they are committed. This is the idea of the 'Cache/Safe' strategy [11] and is termed a NO-STEAL buffer policy in [21]. So, as UNDO log records can be kept in main memory until they are no longer needed, forcing REDO log records to disk during the commit phase of transactions remains the major bottleneck of log-based recovery techniques.

The I/O time to write REDO log records of a transaction can be decreased considerably through the use of special channel facilities, e.g. 'chained I/O' (cf. [11]). In fact, some DBSs have implemented highly sophisticated channel programs (e.g. [29]). It is our strong opinion that low-level I/O driver routines are the responsibility of the OS kernel and actually should not be executed directly by a non-trusted process like a DBS normally is. So the natural place for optimizing REDO loggging I/Os is in the OS.

An even higher performance enhancement is achievable through a method called 'group commit' [7]. When a client of the OS recovery service requests to commit, it could be deferred until a certain number of clients is ready to commit. Then, REDO log records of these clients are altogether written to disk in one chained I/O operation, thus drastically decreasing I/O costs. DBSs have already utilized this technique too [12,22]. The overall performance gain could be still higher yet in an OS recovery service, as not only DBS processes but also clients such as DC system processes and file servers (cf. fig.1) participate in group commit.

Recovery services in the OS kernel seem to be very attractive because of possible optimization due to its closeness to hardware. In addition to special I/O drivers, non-conventional disk architectures such as associative search disks (cf. [32]) or non-volatile semi-conductor disks (cf. [34]) will have an impact on recovery performance and maybe even recovery algorithms. It has been a traditional task of OSs to provide device independence for user processes. We think that this should hold for DBSs too, also in view of novel device types.

3. Drawbacks of OS Transactions

To summarize the results of the last section, we consider OS locking service and OS recovery service as a basis that is well-suited for DBS transaction management. Since both services are complementary to each other, they can be integrated to provide a complete OS transaction facility. Clients of such a service, particularly DBS processes, only have to issue BOT and EOT or RBT operations. The OS transaction service acquires all necessary locks automatically, and is fully responsible for recovery measures.

However, any OS transaction concept can only comprise objects and operations that are known to it. As a consequence of this, each client operation on an OS object such as a file must have associated an explicit OS kernel call in order to inform the OS transaction manager about it. DBSs usually refer to pages, i.e. fixed-size (typically 2K or 4K) blocks of a primitive file, as OS objects. This means that each request to fix or unfix a page has to be treated as an OS kernel call, whether it causes an I/O or not. Under these circumstances it seems reasonable to delegate buffer management to the OS too. As we mentioned before, it is not clear whether OS buffer replacement algorithms can achieve results nearly as good as a DBS would. Certainly, the OS has no semantic knowledge about DBS page reference patterns, but it seems that most DBSs essentially use a rather simple LRU policy [10].

Two further consequences are derived from the page orientation of the DBS-to-OS interface. If transaction management is in the OS, then [40]:

- page level locking is performed for all types of data, and
- page level recovery is performed for all types of data.

The first item does not take into consideration 'hot spot' data, i.e. data with a high rate of congestion. Whole pages are even locked when catalog information such as tuple counters are accessed, although the DBS only uses this data for query optimization and thus could tolerate inaccuracies [41]. Similar considerations appear to be valid for recovery. Page logging usually means that full before images and after images are recorded, even if only a few bytes have been changed [40]. We will revise this point in the next section.

[41] proposes a solution to these drawbacks. It recommends that the OS transaction service could be informed to relinquish locking and recovery measures for certain objects. Moreover, it should allow clients to record application-specific events of their own in the log. We suggest a different approach called 'multi-level transactions' that remedies the above mentioned disadvantages and yet tries to profit from the OS transaction service as much as possible. Basically, its idea is to deal with OS transactions as they are, building a higher-level notion of DBS transaction upon them.

As we will show in the next section, multi-level transactions also provide a clean concept to cope with another difficulty of pure OS transactions. The OS has to identify a client for each active transaction by some unique criterion. From an OS point of view, it seems very natural to use process IDs for this purpose (e.g. [27], [46]), as a transaction is always executed in the context of a particular process, or possibly a tree of processes in a distributed system. In order to avoid confusion about transaction clients, this process cannot issue a BOT request for a second transaction unless the one it is executing is terminated first, either by EOT or by RBT. Note that in the nested transaction case, this condition still holds for transaction roots. Thus, in many proposals, client processes are tied to at most one OS transaction for each period of time. Conversely, when a process issues a BOT operation, all subsequent actions of this transaction must be invoked directly, or indirectly in the case of a process tree, from the original process. Implications of this binding scheme are discussed in detail in the following section 4.

4. Mapping of DBS Transactions to OS Transactions

As we indicated at the end of the last chapter, the realization of DBS transactions by OS transactions depends on the DBS process structure. This correlation has its origin in a one-to-one binding scheme between OS transactions and processes. The following three basic types of DBS process structures are used in practice [18,20].

In a **symmetric server process** approach, each DBS transaction is run as its own DBS process. Either the DBS code is executed directly in the application process, or because of code protection a separate DBS process is spawned dynamically and destroyed after the transaction is finished. The first method seems to be implemented in DB2 [17], the second one is followed by INGRES [30] for example. In either case, scheduling is fully delegated to the OS.

At the opposite end of the range of possibilities, a **single server process** method has been successfully implemented e.g. in SQL/DS [4]. One DBS process, permanently running in the background, performs the requests of all transactions. To achieve satisfactory throughput results, such a server must do asynchronous I/Os. It cannot afford being blocked by a request that runs into a lock wait, as the blocking transaction can make progress and eventually release the lock only if this single DBS process is executed further. Hence the DBS has to do some fairly complicated scheduler tasks, e.g. saving context information of interrupted request-processing 'agents'. On the other side, this approach has the virtue of low costs due to process management.

To simplify scheduling and to allow for a multi-processor environment, **multiple server processes** (e.g. [35]) could be installed at system boot time. These processes need not necessarily perform asynchronous I/Os. They could do without internal multitasking, simply deactivating a process in case of a lock wait. Typically, the number of concurrent transactions is higher than the number of DBS processes by a factor of 2 to 5 (cf. [20]). So a good and yet simple strategy for such a server type could be to process any DBS operation request to its end in one of the server processes before this process accepts another request. At each point of time, every DBS process acts on behalf of at most one transaction.

We are now going to investigate the suitability of these various server types with regard to OS transactions. In a symmetric server process system, each DBS transaction can be mapped one-to-one onto an OS transaction running in the DBS process assigned to it. The following example demonstrates a possible process structure.

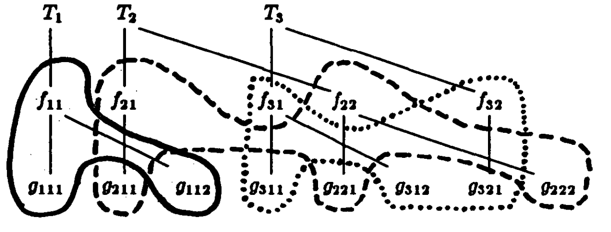


Fig.2: Transactions in a symmetric server process system

The figure shows transactions T_i issuing several DBS tuple operations f_{ij} which in turn result in a number of OS page operations g_{ijk}. Transactions T_1, T_2 and T_3 are processed in three different DBS servers created dynamically. The execution order of actions has to be read from left to right in the above figure. Due to index maintenance, one DBS tuple operation usually corresponds to more than one page operation. Separation of application programs and DBS code in distinct processes is indicated by excluding transaction roots from servers.

Each DBS server can initiate a (page level) OS transaction in this approach. For the scenario of figure 2 the transactions would be:

$T'_1 = < g_{111}, g_{112} >$ in the solid process
$T'_2 = < g_{211}, g_{221}, g_{222} >$ in the dashed process
$T'_3 = < g_{311}, g_{312}, g_{321} >$ in the dotted process

Clearly, the notion of OS transactions fits nicely with a symmetric server process scheme.

In a single server approach there is no possibility to assign a unique process ID to DBS actions, since all requests are handled in the same process. Therefore, OS transactions are not attractive at all for such an environment, as one process would have to act as a client for more than one OS transaction at a time. Actually this is not even surprising, as a single server DBS nearly is an operating system in itself, ignoring most OS functions as e.g. scheduling. For this type of server it would probably be better to run on a small special-purpose OS kernel that only provides most rudimentary facilities and therefore incurs little overhead (cf. [42]). However, this is in sharp contrast to our overall philosophy as indicated in section 1.

In the case of a multiple server process structure we could have a scenario like that of figure 3.

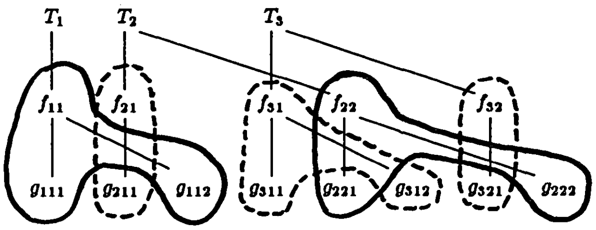


Fig.3: Transactions in a multiple server process system

Obviously, the one-to-one mapping of DBS transactions onto OS transactions, that has worked for the symmetric server type, is not feasible here. Requests of three different transactions are executed in the context of two DBS processes, which are drawn in the above figure with solid lines and dashes respectively. Assignment of requests to servers could be according to some load balancing scheme. Since transaction T_2 migrates from the dashed process to the solid one, treating its page action sequence $< g_{211}, g_{221}, g_{222} >$ as an OS transaction would violate the principle that OS transactions are always bound to one single process.

This binding rule is yet fulfilled by page level actions of each DBS operation request. Therefore, we could treat these requests as OS transactions obtaining:

$T'_{11} = < g_{111}, g_{112} >$ in the solid process
$T'_{21} = < g_{211} >$ in the dashed process
$T'_{22} = < g_{221}, g_{222} >$ in the solid process
$T'_{31} = < g_{311}, g_{312} >$ in the dashed process
$T'_{32} = < g_{321} >$ in the dashed process

But is this actually of any value to a DBS? Fortunately, the answer is yes, as we are going to show now. Our idea is to apply a multi-level transaction methodology [48,49] with two layers here. The OS transaction manager guarantees that single DBS operations appear isolated from concurrent requests and are resilient w.r.t. certain failure categories. From the view of an additional DBS transaction manager this means that single DBS operations can be treated as atomic actions. Hence it is valid for a DBS to use tuple locking without the need to care about interferences on the page level. DBS transactions thus are:

$T_1 = < f_{11} >$
$T_2 = < f_{21}, f_{22} >$
$T_3 = < f_{31}, f_{32} >$

It has been shown elsewhere [1,50] that level-by-level serializability is sufficient for a multi-level concurrency control method to be sound. The DBS transaction manager yields serializability of DBS transactions, and the OS transaction manager ensures serializability of single DBS operations, i.e. OS transactions. Transaction T_3, for example, then actually looks like:

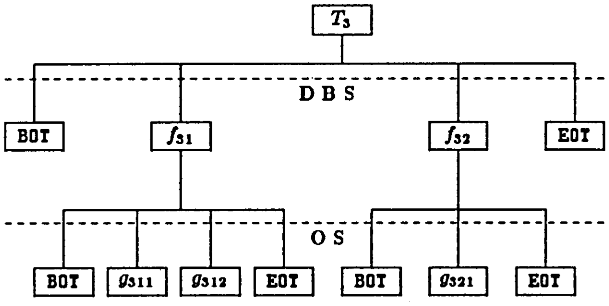


Fig.4: Two-level transactions based on OS transactions

Details of this kind of transaction management are discussed in the next section 5. The multi-level transaction approach is also applicable in the symmetric server process case. We are getting into difficulties, however, with the multiple server type, when DBS processes perform internal multitasking, e.g. to avoid becoming deactivated in case of a lock wait (e.g. UDS [35]). The problem is illustrated in the following figure.

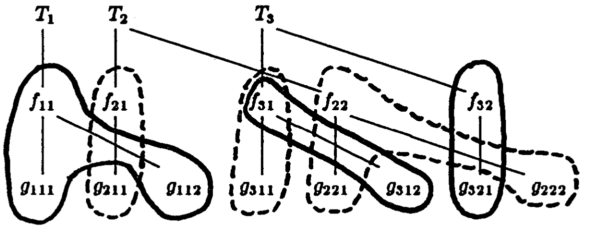


Fig.5: Transactions in a multiple server process system with internal multitasking

In this scenario even a single DBS operation, viz. f_{31}, migrates from the dashed process to the solid one. Therefore, the multi-level transaction methodology is rendered impracticable in this case.

5. Multi-Level Transaction Management

Multi-level transactions not only overcome the problem of binding OS transactions to processes, but have another major virtue. Page locks are released at the end of each OS transaction, i.e. at the end of each DBS tuple operation. Only tuple locks must be held until EOT of the DBS transaction. In most cases this results in increased parallelism. One of the major objections to OS transactions (see chapter 3), viz. pure page locking, is remedied this way. However, it is still not possible to use special locking techniques for special DBS-like data structures within the OS, e.g. a B-tree locking protocol, unless the OS has semantic knowledge about these structures. On the other hand, such protocols are hardly used in real DBSs today, since for recovery reasons one has to keep locks on updated tree nodes until EOT anyway (cf. [33]). We claim that also our multi-level locking protocol prevents B-tree roots from becoming 'hot spots', just by releasing page locks at the end of each DBS tuple operation.

Log space may be saved by recording tuple changes only instead of whole pages (cf. [16]). However, as tuple actions are in turn made resilient by page logging, effectively reducing the log size requires some more refined technique than straightforward multi-level recovery. We will not discuss this issue further here.

To be fair, disadvantages of our multi-level transaction approach should not be withhold. As page locks are released prematurely, UNDO of an unfinished DBS transaction through OS page level recovery, i.e. restoring page before images for example, could result in anomalies such as lost updates. Instead DBS operations have to be compensated by their inverses. Undoing an 'INSERT' operation means to execute a 'DELETE' operation on the inserted tuple (cf. [14]). Admittedly, this is a costly method, but as transaction UNDOs, e.g. due to deadlocks, normally are infrequent events, we do not consider this as a serious drawback and are willing to accept this price in return for enhanced concurrency.

Another objection could be that the DBS transaction manager needs its own locking and recovery services [41]. This is true for locking, but, apart from certain restrictions mentioned in section 2.1, the OS lock manager could be used for this purpose if it is made available as an independent OS service. DBS operation logging could be performed within OS transactions like any other page level operation. A DBS operation inserting a tuple t into a page p and causing index maintenance operations on pages q and r would look like:

$$< \mathtt{BOT}, read(p), write(p), read(q), read(r), write(r), write(s), \mathtt{EOT} >$$

where s is a log page containing an entry for the inverse DBS operation, viz. 'DELETE t'. Atomicity and persistence of OS transactions ensure that after a crash this UNDO entry will be contained in the DBS operation log if and only if tuple t has really been inserted and all indices have successfully been updated. Thus, log management functions of the OS transaction service need not be duplicated in the DBS.

However, building DBS logging into OS transactions also has disadvantages. As the OS does not distinguish between data pages and DBS log pages, it enforces page locking on the DBS log too. Fortunately, potential lock waits on a log page are very short, because DBS log records are always written as the very last action of an OS transaction. By splitting the DBS log into n separate logs, where n is the maximum number of concurrent OS transactions, contention on this ressource could be totally eliminated, but, on the other hand, this would not allow for packing log records of several processes into a single page. In either case, in a rigorous realization of our multi-level transaction approach, two DBS processes cannot write their DBS log records together using only one single I/O.

An essential drawback of our multi-level transaction scheme is that it will pretty surely increase I/O rates due to REDO logging. Our approach splits one DBS transaction into a number of short OS transactions. Each time such an OS transaction commits, page level REDO log records must be forced to disk. These I/O costs are inevitable and can only be reduced to a certain degree. A clever technique based on the 'Cache/Safe' strategy of [11] could be to record only changed byte strings (cf. [25]) instead of whole page after images in a log buffer. Such a log entry can be computed without knowing anything about the DBS update operation that has caused the page modification, simply by comparing the page with its before image at commit time. This comparison causes no additional I/O since in the Cache/Safe approach before images are kept in the buffer pool throughout the duration of a transaction. Please note that such a NO-STEAL strategy [21] is more likely to be practicable when page level transactions are rather short as is the case in our multi-level scheme. Then, using group commit and sequential chained I/O for the REDO log buffer might reduce EOT costs to a tolerable rate.

In order to get quantitative results about the trade-offs of the proposed multi-level transaction approach, a number of simulation experiments have been performed. Since we have no OS offering transactions available, we misused a DBS for page level locking and recovery as well as for mapping tuple operations into sequences of page I/Os. On top of this layer, tuple level locking and logging has been implemented as sketched above. Measurements were performed with a mix of DBS transactions from a real-world application against a 130 MB data base of real-world data. As underlying process structure we used a variant of the symmetric server concept, combining it with a simple load generating mechanism. More technical details about this performance study are contained in [6].

In the experiments, two basic techniques were applied and compared with each other: using page level transactions directly for application transactions vs. our multi-level transaction approach. A series of measurements has given rise to the following key observations:

- On the average, multi-level transaction management increased the throughput of application transactions by 12 percent.

- Pure page level locking resulted in an unexpectedly high number of deadlocks, whereas the multi-level technique hardly suffered from deadlocks at all. Obviously, for short-lived subtransactions the danger of a page level deadlock is extremely low, and usually only few deadlocks occur due to tuple level waiting situations.

- With pure page locking, the lock wait probability, i.e. the frequency of lock requests resulting in a lock wait, was fairly low, viz. about 1 percent. This result, however, was clearly outperformed by the multi-level locking protocol which achieved less than 0.2 percent on pages, just by releasing locks early. Of course, in the latter case one has to consider the lock wait probability on the tuple level too, which ranged from 1 to 2 percent. However, as the absolute number of lock waits on tuples was extremely low, compared with lock waits on pages, the crucial performance impact seems to arise from page conflicts.

- The inherent increase of I/Os for the multi-level technique was even worse than expected. Our measurements indicated a factor of 3 per application transaction. In particular, the number of I/Os caused indirectly by tuple level logging was tremendous. Basically, this had two reasons: The underlying page level recovery system forced before-images to disk before modifying pages in the buffer, and flushed committed pages to the permanent data base immediately at EOT. Therefore, each tuple level log record caused two additional I/Os.

Our simulation results have been very encouraging so far. Although the underlying page level recovery method actually has been totally unsuitable for short (sub-) transactions, the multi-level technique was the winner w.r.t. throughput. The next step will be to investigate potential performance gains of implementation techniques such as the Cache/Safe approach and the group commit mechanism. These concepts are supposed to reduce the I/O bottleneck substantially.

6.Conclusion

Several consequences might be concluded from our discussion of benefits and drawbacks of OS transactions in the various DBS server process structures.

- A first conclusion could be that process management costs have to become much lower in next generation OSs. Under this precondition, the symmetric server architecture would certainly appear most favourable, as it is easy to implement, delegates all scheduling tasks to the OS, and seems to fit best with the concept of OS transactions.

- If a very high number of concurrent processes still causes intolerable overhead, a multiple server solution would probably yield better performance results, especially in combination with a DC system (cf. [19]). This approach, however, should be implemented without internal multitasking, in order to allow for the utilization of OS transactions according to our multi-level transaction methodology.

- As pointed out in chapter 4, the trouble with OS transactions and DBS servers primarily comes from binding OS transactions to processes. If instead each BOT operation returns a system-wide unique transaction ID, this ID could be used as a client identification in all subsequent service calls that refer to the same transaction. Thus, clients could migrate from one process to another without interfering with OS transaction management, and, on the other hand, a DBS server process would be allowed to act on behalf of more than one OS transaction in an arbitrarily interleaved way.

Of course, we vote for the first two conclusions, whereas the third one is debatable. Our personal opinion is, that from an OS point of view, binding service functions, like OS transactions are, to processes seems to be the most natural and manageable approach as long as processes themselves are considered a suitable concept to structure program executions in a multi-user OS. Therefore, it is likely that future OSs will indeed expect transactions to be bound to processes. DBSs, like any other client, will have to make the best of it.

Altogether, OS transactions seem to be a really useful service for various kinds of clients. Whether DBSs in particular will accept this facility, will probably depend on performance issues. As we have argued mostly qualitatively in this paper, more efforts should be taken to gain quantitative results.

Our own work will concentrate on the promising direction of multi-level transactions. For this method, the architecture of figure 1 has to be expanded slightly. In addition to the transaction service, the OS should offer locking as a common service to be used by higher layers. Avoiding reimplementation of functions in different subsystems certainly outweighs minor performance issues in this case. Furthermore, to reduce the costs of DBS logging in the multi-level transaction framework, it could be advantageous to allow direct use of the OS logging service from the DBS level too, though this actually violates the spirit of the multi-level transaction approach to some extent. The main conclusion is that to improve the overall performance of multi-level transactions, an OS transaction kernel should be oriented towards a high number of short-lived (sub-) transactions rather than handling DBS transactions entirely.

References

[1] C.Beeri, P.A.Bernstein, N.Goodman, A Model for Nested Transaction Systems, Technical Report, The Hebrew Univerity, Jerusalem, 1986

[2] P.A.Bernstein, N.Goodman, M.-Y.Lai, Laying Phantoms to Rest, IEEE COMPSAC Conference, 1981

[3] M.Blasgen, J.Gray, M.Mitoma, T.Price, The Convoy Phenomenon, ACM Operating Systems Review Vol.13 No.2, 1979

[4] D.D.Chamberlin, A.M.Gilbert, R.A.Yost, A History of System R and SQL/Data System, Proc. 7th VLDB Conference, Cannes, 1981

[5] P.Dadam, P.Pistor, H.-J.Schek, A Predicate Oriented Locking Approach for Integrated Information Systems, Proc. Information Processing 83, Paris, 1983

[6] T.Dey, H.Georg, Performance Evaluation of Multi-Level Transaction Management Strategies, Master Thesis, Technical University of Darmstadt, 1986, in German

[7] D.J.DeWitt, R.H.Katz, F.Olken, L.D.Shapiro, M.R.Stonebraker, D.Wood, Implementation Techniques for Main Memory Database Systems, Proc. ACM SIGMOD Conference 1984

[8] H.Diel, G.Kreissig, N.Lenz, M.Scheible, B.Schoener, Data Management Facilities of an Operating System Kernel, Proc. ACM SIGMOD Conference 1984

[9] VAX/VMS System Services Reference Manual, Digital Equipment Corp., Maynard, Massachusetts

[10] W.Effelsberg, T.Haerder, Principles of Database Buffer Management, ACM Transactions on Database Systems Vol.9 No.4, 1984

[11] K.Elhardt, R.Bayer, A Database Cache for High Performance and Fast Restart in Database Systems, ACM Transactions on Database Systems Vol.9 No.4, 1984

[12] D.Gawlick, D.Kinkade, Varieties of Concurrency Control in IMS/VS Fast Path, IEEE Data Base Engineering Vol.8 No.2, 1985

[13] J.Gray, Notes on Data Base Operating Systems, in: Operating Systems - An Advanced Course, LNCS 60, Springer-Verlag, Berlin-Heidelberg-New York 1978

[14] J.Gray et al., The Recovery Manager of the System R Database Manager, ACM Computing Surveys Vol.13 No.2, 1981

[15] J.Gray, Why Do Computers Stop and What Can Be Done About It?, Proc. German Chapter of the ACM Conference on Office Automation, 1985

[16] N.D.Griffeth, Reducing the Cost of Recovery from Transaction Failure, IEEE Database Engineering Vol.8 No.2, 1985

[17] D.J.Haderle, R.D.Jackson, IBM Database 2 Overview, IBM Systems Journal Vol.23 No.2, 1984

[18] T.Haerder, Embedding a Database System in an Operating System Environment, Proc. German Chapter of the ACM Conference on Data Base Technology, 1979, in German

[19] T.Haerder, K.Meyer-Wegener, Transaction Processing Systems, TP Monitors, DB/DC Systems - Functional Requirements and Implementation -, Technical Report, University of Kaiserslautern, 1985, in German

[20] T.Haerder, P.Peinl, Evaluating Multiple Server DBMS in General Purpose Operating System Environments, Proc. 10th VLDB Conference, Singapore, 1984

[21] T.Haerder, A.Reuter, Principles of Transaction Oriented Database Recovery, ACM Computing Surveys Vol.15 No.4, 1983

[22] P.Helland, Transaction Monitoring Facility (TMF), IEEE Database Engineering Vol.8 No.2, 1985

[23] J.L.Keedy, J.Rosenberg, K.Ramamohanarao, On Synchronizing Readers and Writers with Semaphores, Computer Journal Vol.25 No.1, 1982

[24] J.Kent, H.Garcia-Molina, Optimizing Shadow Recovery Algorithms, Technical Report, Princeton University, 1985

[25] B.G.Lindsay et al., Notes on Distributed Databases, IBM Research Report RJ2571, San Jose, 1979

[26] B.Liskov, R.Scheifler, Guardians and Actions: Linguistic Support for Robust, Distributed Programs, ACM Transactions on Programming Languages and Systems Vol.5 No.3, 1983

[27] E.T.Mueller, J.D.Moore, G.J.Popek, A Nested Transaction Mechanism for LOCUS, ACM Operating Systems Review Vol.17 No.5, 1983

[28] T.W.Page, M.J.Weinstein, G.J.Popek, Genesis: A Distributed Database Operating System, Proc. ACM SIGMOD Conference, 1985

[29] R.J.Peterson, J.P.Strickland, Log Write-Ahead Protocols and IMS/VS Logging, Proc. ACM Symp. on Principles of Database Systems, 1983

[30] INGRES Reference Manuals, Relational Technology Inc., Berkeley, California

[31] J.T.Robinson, A Fast General-Purpose Hardware Synchronization Mechanism, Proc. ACM SIGMOD Conference, 1985

[32] W.D.Roome, The Intelligent Store: A Content-Addressable Page Manager, Bell System Technical Journal Vol.61 No.9, 1982

[33] D.Shasha, What Good are Concurrent Search Structure Algorithms for Databases Anyway ?, IEEE Database Engineering Vol.8 No.2, 1985

[34] S.Shibayama, T.Kakuta, N.Miyazaki, H.Yokota, K.Murakami, A Relational Database Machine with Large Semiconductor Disk and Hardware Relational Algebra Processor, New Generation Computing Vol.2, 1984

[35] UDS: Universal Database Management System - System Reference Guide, Siemens AG, Munich

[36] A.Z.Spector, P.M.Schwarz, Transactions: A Construct for Reliable Distributed Computing, ACM Operating Systems Review Vol.17 No.2, 1983

[37] A.Z.Spector et al., Support for Distributed Transactions in the TABS Prototype, IEEE Transactions on Software Engineering Vol.SE-11 No.6, 1985

[38] L.L.Spratt, The Transaction Resolution Journal: Extending the Before Journal, ACM Operating Systems Review Vol.19 No.3, 1985

[39] M.Stonebraker, Operating System Support for Database Management, Communications of the ACM Vol.24 No.7, 1981

[40] M.Stonebraker, Virtual Memory Transaction Management, ACM Operating Systems Review Vol.18 No.2, 1984

[41] M.Stonebraker, D.DuBourdieux, W.Edwards, Problems in Supporting Data Base Transactions in an Operating System Transaction Manager, ACM Operating Systems Review Vol.19 No.1, 1985

[42] M.Stonebraker, J.Woodfill, J.Ranstrom, M.Murphy, M.Meyer, E.Allman, Performance Enhancements to a Relational Database System, ACM Transactions on Database Systems Vol.8 No.2, 1983

[43] L.Svobodova, File Servers for Network-Based Distributed Systems, ACM Computing Surveys Vol.16 No.4, 1984

[44] I.L.Traiger, Virtual Memory Management for Database Systems, ACM Operating Systems Review Vol.16 No.4, 1982

[45] I.L.Traiger, Trends in Systems Aspects of Database Management, Proc. 2nd Int. Conf. on Databases, Cambridge (UK), 1983

[46] J.Verhofstad, Common Journaling and Recovery Units, Field Test 1 Seminar for VMS 4.0, Digital Equipment Corp., 1983

[47] B.Walter, Nested Transactions with Multiple Commit Points: An Approach to the Structuring of Advanced Database Applications, Proc. 10th VLDB Conference, Singapore, 1984

[48] B.Walter, Multi-Level Synchronization and Nested Transactions in Advanced Information Systems, Proc. GI Conference on Database Systems for Office Automation, Engineering and Scientific Applications, 1985

[49] G.Weikum, H.-J.Schek, Architectural Issues of Transaction Management in Multi-Layered Systems, Proc. 10th VLDB Conference, Singapore, 1984

[50] G.Weikum, A Theoretical Foundation of Multi-Level Concurrency Control, Proc. ACM Symposium on Principles of Database Systems, 1986

[51] N.Whyte, ELXSI Data Management Services, Exhibition Program, 10th VLDB Conference, Singapore, 1984

MAIN MEMORY DATABASE RECOVERY

Margaret H. Eich

Department of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275

ABSTRACT

The idea of having entire databases reside in main memory, or *main memory databases (MMDB)*, has recently been an active research topic. It is recognized that in this framework the issues concerned with efficient database recovery are more complex than in traditional DBMS systems. While several authors have looked at different methods for MMDB recovery, an examination of MMDB recovery functions and how they differ from traditional DBMS recovery has not been performed. This paper examines MMDB recovery, identifies differences from traditional DBMS recovery, composes a "wish list" of MMDB recovery requirements, describes why previously proposed techniques do not satisfy these requirements, and proposes a new MMDB recovery technique which does.

Introduction

The declining cost of main memory and need for high performance database systems have recently inspired research into systems with massive amounts of memory and the ability to store complete databases in main memory [1,3,8]. The use of memory resident databases, or *main memory databases (MMDB)*, can achieve significant performance improvements over conventional database systems by eliminating the need for I/O to perform database applications.

Database recovery techniques are used to ensure that any erroneous database state due to transaction, system, or media failure can be repaired and restored into a usable state from which normal processing can resume [2,9,11,18]. Due to the volatility of main memory, MMDBs complicate database recovery issues. This problem has been recognized and several new techniques for MMDB recovery have been proposed [3,5,6,15]. The objectives of this paper are to examine recovery requirements in a MMDB environment, provide a survey of proposed techniques, and then propose a new MMDB recovery technique which better meets these requirements than previous methods.

The approach used in this paper is different than that in previous papers on MMDB recovery. Prior to defining a technique, we investigate what is meant by MMDB recovery. With MMDBs, the primary database copy is memory resident. This change in perspective from secondary storage to main memory not only causes problems with the recovery operations, but also changes some of the accepted methods for recovering from various failures. In the next section we identify the MMDB model to be used throughout the paper, defining the various database components used. Section 3 then uses this model to define MMDB recovery and compare it's requirements for recovery to those in traditional database systems. From this evaluation, we construct a "wish list" of MMDB recovery requirements which we feel a MMDB recovery technique should satisfy. Section 4 reviews existing MMDB recovery proposals and shows that none meet all requirements on the wish list. Finally, in section 5 we propose a new recovery technique which does satisfy the wish list items.

MMDB Model

The DBMS model under evaluation is one where entire databases being accessed reside in main memory. Although only recently receiving much research interest, the concept is certainly not new. IMS has supported main memory databases, Main Storage Data Bases, for quite some time [12]. Unlike the IBM approach, however, we assume no practical limitation on the size or structure of a main memory database.

Figure 1 shows the MMDB model used throughout this paper. Sometime prior to access, a database must be loaded into main memory. To achieve this purpose, an *Archive Database* exists on secondary storage. The archive database is a complete image of some prior database state and is used solely for loading main memory. Since no access of the archive is made during database processing, the organization of this file should be for efficient loading of memory. Due to the volatility of main memory, all updates to main memory must be recorded on a *Log* located in some stable memory. Figure 1 shows the log on a secondary storage device, but it may actually exist in a nonvolatile main memory supported by a backup power supply, or in a combination of the two. During recovery processing, the log can be used to achieve UNDO and REDO processing and thus may contain *before Images (BFIM)* and/or *after images (AFIM)* of modified data [11]. Together, the archive and log provide the ability to recover from a

CH2345-7/86/0000/1226$01.00 © 1986 IEEE

main memory failure. In actual use the archive database may be updated directly from data in main memory or from data in the log.

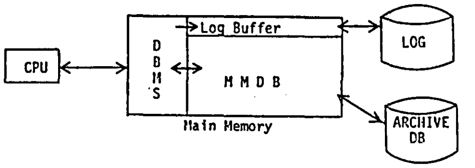


Fig. 1. - MMDB Model

This view of a MMDB may seem almost identical to that of a conventional database. However, there exist several major differences:

1. Main memory is assumed to be large enough to hold all databases currently being accessed.
2. Any recovery schemes must deal with restoring the MMDB not data on secondary storage.
3. No database access can be performed against the archive database. It's use is strictly as a backup to the main memory database.

There are several advantages to the use of MMDBs. Obviously, processing time and throughput rates should improve due to the elimination of I/O overhead. It has been suggested that the improved performance can eliminate the need for concurrency control by allowing the serial execution of MMDB transactions [7]. While we don't agree with this observation, it is certainly conceivable that concurrency control mechanisms specifically designed for the MMDB environment can reduce the overhead and complexity usually associated with concurrency control techniques [3].

Some problems not existing in conventional DBMS systems are introduced in the MMDB environment. The major problem deals with the volatility of main memory. To reduce the impact of this problem, it has been proposed that a small stable main memory be used to support recovery processing [3,6]. If a stable memory is assumed, its size must be large enough to contain all updates of active transactions. Another major problem is the excessive overhead needed to initially load a database into memory for processing. This procedure requires merging data on the archive database and the log to obtain data needed to recover to the most recent consistent state. To reduce the time, archives should be created very frequently and could be distributed across several secondary storage devices. Additional concerns center around the increase in the number of main memory components and the resulting reliability problems and increase in access time. A unique architecture is currently under investigation at Princeton to address these issues [7]. We are currently only addressing what appears to be the major problem: MMDB recovery.

MMDB Recovery Defined

The issues concerning traditional database recovery are well known and understood [2,9,11,18]. The purpose of this section is to examine aspects concerning database recovery under the MMDB assumption. We investigate the various types of failures effecting database processing, describe recovery operations necessary for MMDBs, and conclude by listing the desired features a MMDB recovery technique should possess.

When discussing MMDB recovery, it is important to realize that the objective is that of recovering data in main memory. With conventional databases, the current database state exists partly in main memory and partly in the secondary storage. With MMDBs, the current state completely exists in main memory. Secondary storage is used solely for recovery purposes. Since MMDB processing incurs no I/O, any I/O required to ensure recoverability can have a significant impact on system performance and become the major bottleneck during processing.

As with traditional database processing, a *transaction* is assumed to be the unit of recovery and consistency, and the failures which must be anticipated are transaction, system, and media failures. The differences between conventional DBMS recovery and MMDB recovery are introduced when the specific operations required to accomplish recovery are examined. TABLE 1 shows the operations needed to recover from the three failure types in both the tradtional and MMDB environments.

TABLE 1
DATABASE RECOVERY OPERATIONS

Failure Type	Recovery Operations Required	
	Traditional DBMS	MMDB
Transaction Failure	Transaction UNDO	Transaction UNDO
System Failure	Global UNDO Partial REDO	Global REDO
Media Failure	Global REDO	Global REDO Partial REDO

Transaction failure occurs when a transaction does not successfully commit. This type of failure occurs more often than the other two, and thus efficient recovery from it is essential. A rule of thumb is that recovery should occur in a similar time frame to that of successful completion of the transaction [10,11]. The normal procedure for recovery after a transaction failure is a *Transaction UNDO.* This implies that all effects the aborted transaction has had on the primary database copy must be removed. The major concern existing with transaction UNDO in a MMDB environment is that it be done with as little I/O as possible. If the rule of thumb is to be achieved, no I/O should be required to accomplish transaction UNDO. Additional processing during transaction UNDO involves the removal of any dirty data from the log or archive database. These would also like to be performed with no I/O.

Recovery from a *system failure* is quite different with MMDBs than with traditional DBMSs. Traditionally, the effects of any interrupted transactions must be undone, *Global UNDO*, and any completed transactions which have not had updates reflected in the database need to be redone, *Partial REDO*. When a system failure occurs, the entire MMDB contents are lost. MMDB recovery must therefore perform a *Global REDO* by completely reloading all databases in main memory. The archive database is used to reload the databases to some prior state and then any committed transactions reflected in the log are redone. The Global REDO operation is required in conventional systems only after a media failure causes the loss of the primary database copy on secondary storage.

System failures occur less often than transaction failures and more often media failures [11]. A goal for system failure recovery is that it be accomplished in a time comparable to that required for successful completion of all active transactions. With MMDBs, a Global REDO requires I/O from the archive database, and thus it seems impossible to achieve this goal. One way to reload the MMDBs as quickly as possible is to ensure as much of recent database updates as possible are reflected in the archive database. This implies that log data must be frequently flushed to the archive database.

In traditional database systems a checkpoint is often used to reduce the work needed to recover from system failures [2,9,11,18]. We view a *MMDB Checkpoint* as recording all data concerning a prior database state into the archive database and writing a corresponding checkpoint record on the log. These checkpoints should be done with as little impact on transaction processing as possible. Frequent flushing to the archive database thus requires frequent checkpointing.

Global REDO loads MMDBs into main memory. However, not all transactions require all data, therefore transactions can begin processing as soon as some of the data they need is available. This problem is similar to the fetching strategies associated with virtual memory management. At least four possibilities exist when loading databases into main memory:

1. Database Prefetching - Loading an entire database into main memory prior to scheduling any transactions accessing it.
2. Page Prefetching - Prefetch some subset of database pages and allow transactions to begin access of them as the remainder of the database is loaded.
3. Demand Loading - Only load a database when some transaction first accesses it.
4. Demand Paging - Load database pages after first access to them.

More research is needed to determine which of these provides the best performance for Global REDO. The method to be used depends on such factors as database storage structure, access methods used, storage location on disk, and whether any transactions have immediate need of the data.

Although *media failure* may only occur once or twice a year, the impact on recovery of traditional databases can be severe [11]. A memory failure with MMDBs can be treated as a system failure and a Global REDO performed. However, if the specific location of the failure can be identified, a Partial REDO of only the effected area would be warranted. This indicates that the archive database should be physically structured to correspond with memory addresses. Perhaps partitioning of memory and archive databases and the ability to recover by these partitions is needed. Future research will examine this idea of partitioning for partial redos. Media failures can effect the archive database or log. Restoring these files creates similar problems for conventional and MMDB systems. Differences do exist in that the archive database may be needed more frequently than in conventional DBMSs and thus it's quick recovery is more important with MMDBs. With the existence of prior archive databases and corresponding log data, recreation of archives simply requires redoing checkpoint processing.

Authors have ignored the problems associated with failure of stable memories. The use of stable memories does imply that overhead for recovery from system failures is greatly reduced, however, there is really no such thing as a "forever stable" memory. Thus any reliable recovery technique must prepare for the failure of stable memory. This implies that any MMDB recovery technique needs to provide the facilities for Global and/or Partial REDO of all of memory - stable and non-stable.

Another issue concerning MMDB recovery is when log I/O operations occur. It is important that any I/O needed be performed asynchronously to normal database processing. This implies that log I/O not occur only at commit time, but that it be performed throughout transaction processing. Transaction processing should not be dependent on or held up by I/O to the log.

We close this section by summarizing the major requirements for MMDB recovery in the following wish list:

1. No I/O required to accomplish transaction UNDO.
2. Frequent checkpoints performed with minimum impact on transaction processing.
3. Asynchronous processing of log I/O and transaction processing.

Certainly an additional requirement be that a minimum amount of redundant data be used. For example, the use of before images on the log should be avoided if possible.

Previous MMDB Recovery Techniques

Prior to introducing the new MMDB recovery method, we examine previously proposed techniques and compare their processing to the items on the wish list.

As stated earlier, IBM has as implemented MMDBs in the IMS/VS Fast Path Feature [12,13]. At initialization of IMS, the MMDBs are loaded into main memory. Updates are performed in special database buffers and MMDB pages are not modified until commit time. Commit processing ensures that all after images are written to the log prior to updating the MMDB. System wide transaction-consistent checkpoints (TCC) [11] are accomplished by an asynchronous IMS task running in parallel with MMDB processing. The major disadvantages of this scheme are that log I/O is only performed at commit time and entire MMDBs must be read to perform checkpoints.

The Massive Memory Machine (MMM) project at Princeton University has described an architecture specifically designed to support massive amounts of primary storage [7,8,17]. Associated with this project is the design of a MMDB recovery scheme based upon a hardware logging device, HALO [6]. HALO intercepts all MMDB operations and creates BFIM and AFIM log data initially written to a nonvolatile main memory, and as time permits written out to the log on disk. The use of the stable main memory implies that commit processing need not wait until the log buffers have been flushed. The BFIM log data is needed to accomplish transaction UNDO. Continuous action-consistent checkpoints (ACC) [11] of log data to the archive database is made. The asynchronous updating of the log and parallel, continuous updating of the archive database are certainly advantages of this scheme. However the requirements for specialized hardware and stable main memory, the interception of all database calls, and the requirement for BFIM log data to accomplish transaction UNDOs are disadvantages.

Researchers at the University of California at Berkeley, have investigated some of the implementation concerns for a MMDB [3]. Their recovery scheme assumes the use of a log with BFIM and AFIM plus frequent checkpointing. The notion of a *pre-committed transaction* is used to achieve asynchronous logging and database processing. When a transaction commits, the commit record is placed in the log buffer and other conflicting transactions are allowed to progress even though the log buffers have not been flushed to disk. The transaction completes commit processing only when this is done. ACC checkpointing occurs continously and in parallel with transaction processing by reading the entire MMDB and identifying modified pages. Although not specifically discussed, it appears that an entire database must be loaded into main memory prior to access.

The concept of a *Database Cache* has been proposed for database systems with large amounts of main memory [5]. It is assumed that there is sufficient memory space to store all dirty pages plus some other pages which have been fixed for reading. The database cache and disk database together represent the current database just as with traditional DBMS systems. Even though this approach is not strictly a MMDB, an unusual recovery scheme appropriate to MMDBs is proposed. This approach recognizes that "shadow" main memory pages [16] can be used to eliminate the need for transaction UNDO. To avoid the overhead of loading entire databases, a demand paging technique is used to bring pages into main memory. No log is used, rather a *safe* located in nonvalatile memory containing data needed to reconstruct part of the cache after failure is maintained. As a minimum, the safe contains all pages not currently residing on the disk database. When memory pages are to be modified, a main memory shadow page is used for updating if the disk database does not contain a copy of the page to be modified. In the event of transaction failure, these shadow pages are simply deleted in the cache. Subsequent transactions will either access the other page in the cache or incur a page fault to bring in a new copy of the page. To commit a transaction all modified pages are written to the safe. Novel procedures are used to limit the number of records on the safe and to determine the exact state of each page in main memory. Only when a page is targeted for replacement is it written back to the disk database.

Design for a MMDB including data structure representation and recovery technique has been propsed at IBM [1]. It is assumed that a MMDB relation is loaded into main memory at the first reference and, if modified, written to disk at commit time. All recovery overhead is restricted to the commit operation achieving a type of transaction-oriented checkpoint (TOC) [1]. No separate log is suggested, rather the use of shadow pages on the archive database. At commit time, all modified relations are written to shadow areas on the archive database. Once this has been accomplished the new directory structure is updated and old database areas released.

Design for a MMDB system, MM-DBMS, is currently under way at the University of Wisconsin-Madison [14,15]. This study includes the design of an architecture, query processing, data structures, and recovery technique for a MMDB. Recovery processing uses a stable log buffer as well as a special log processor to perform checkpointing. Further details concerning the recovery strategy used are not yet available.

TABLE 2 summarizes the different MMDB recovery techniques described. Although some unique recovery ideas are introduced, none of the techniques satisfies all items on our wish list. The Fast Path Feature, DB Cache, and Ammann techniques don't meet the asynchronous log I/O and transaction processing requirement because all log I/O overhead occurs at transaction commit. The MMM and Berkeley methods require BFIM and AFIM log data and must have log I/O operations to accomplish a transaction UNDO. The information shown in the last row of TABLE 2 describes the type of checkpointing performed by the various techniques. Part of this data indicates whether checkpointing is accomplished by reading the MMDB or log data. Checkpoints for the Fast Path and Berkeley

TABLE 2
COMPARISON OF PREVIOUS MMDB RECOVERY TECHNIQUES

Recovery	Recovery Technique				
Operation	Fast Path	HALO	Berkeley	DB Cache	Ammann
Loading (Global REDO)	DB Prefetch	DB Prefetch	DB Prefetch	Demand Paging	Demand Loading
Shadow	Yes	No	No	Yes	No
Log I/O	Commit	Asynchronous	Asynchronous Pre-Commit	Commit	Commit
Checkpoint	TCC Parallel MMDB	ACC Parallel Continuous Log	ACC Parallel Continuous MMDB	Page Replacement MMDB	TOC MMDB

techniques require examining all MMDB pages and therefore must impact transaction processing when checkpointing occurs.

A Better Technique

In this section we describe preliminary results concerning a new MMDB recovery technique which better satisfies the requirements identified in section 3 than previously proposed methods. The highlights of this new method are:

1. Main memory shadow pages
2. Pre-committed transactions
3. Automatic checkpointing
4. Recovery processor

Each of these is discussed in the following paragraphs. A followup paper will define this technique in more detail as well as provide simulation results examining its performance. We assume that no stable main memory exists, but note any changes which would be needed in the event nonvolatile main memory were available. The impact of concurrency control is not included in this discussion. It may be assumed that transactions are either run serially or that two-phase locking at the page level is used.

Main memory shadow pages (similar to that proposed for the DB Cache [5]) are used to achieve the goal of no I/O for transaction UNDO. Duplicate copies are made of any pages updated by a transaction and all modifications occur on these pages. As pages are modified, AFIM records are written into the log buffer for output to disk. At commit time, a commit record is also written in the buffer for output. Subsequent conflicting transactions can begin processing as soon as this occurs. They will use the data in the dirty pages and, if needed, create new copies for their updating. If a transaction commits (commit record written to disk), the previous clean pages are released and the dirty pages become the new clean ones. When a transaction abnormally terminates, the dirty pages are released.

To accomplish asynchronous log I/O and transaction processing, the pre-commitment technique is used. As explained above, transactions need not wait until a conflicting transaction has completely commited prior to beginning execution. Also, log I/O is performed throughout a transaction execution rather than just at commit time. Indeed, a transaction can not commit until all log buffers are written to disk, but without a stable main memory for the buffer this can not be avoided. If stable main memory were available, the pre-commitment technique would not be necessary and completely independent log I/O and transaction processing would be possible.

The log contains Begin_Transaction (BT), Commit_Transaction (CT), Abort_Transaction (AT), Checkpoint, and AFIM records. All log records except the checkpoint contain the ID of the corresponding transaction. The BT record contains a flag that indicates the state of the corresponding transaction. It is initially written with an indication that the transaction is active, when commited or aborted it is appropriately modified. This random access to the log implies that a random access device must be used. As explained below, this flag is used during checkpointing to avoid flushing dirty data to the archive database. It also eliminates the need for removing this dirty data during transaction UNDO.

To accomplish *automatic checkpointing*, the logger keeps track of the state of the log. Assuming an initial state of 0, state transitions occur when BT, CT, or AT records are written to the log. The BT record increments the state value by 1, while the CT and AT decrement it by 1. A state value of 0 indicates that a TCC state exists on the disk. When the logger detects a state value of 0, a checkpoint record is written out to the disk before any other records already in the buffer. To find the most recent checkpoint record on the log, the logger also notes the address of this checkpoint record and records it along with its unique checkpoint ID in a predefined fixed disk address. Performing this automatic checkpointing only requires two additional I/O operations and is performed independently of transaction processing. To ensure that TCC checkpoints occur, the logger may need to periodically force checkpoints even though the checkpoint state has not occurred. To accomplish this, the logger must write all log records for actively executing transactions to be written to the log, before allowing any new transactions to begin writing to the log. Unlike the techniques used in current database systems where processing is quiesced to reach a TCC, this technique has no impact on executing transactions.

Actual checkpointing to the archive database is accomplished by a separate *Recovery Processor (RP).* Checkpointing is a two step process: creating a new copy of the archive database and then applying log AFIM records for committed transactions to bring the new archive database state up to that indicated by the latest checkpoint on the log. When an archive database is first created, the checkpoint ID and associated address of the latest log checkpoint record are written into a predefined location on the archive database. All AFIM records for successfully committed transactions (as identified by the BT log record) between the checkpoint address on the previous archive database and the address of the latest checkpoint are applied to the archive. If a checkpoint is attempted, but the checkpoint ID on log is the same as that on the prior archive, then the RP must wait until a new checkpoint is taken. To avoid the overhead of copying the entire archive, many checkpoints could be applied to the same copy and at periodic time intervals request the creation of a new archive database.

Figure 2 shows the model of the MMDB recovery system proposed. The use of main memory shadow pages ensures that only AFIMs are needed on the log and that no I/O is required for transaction UNDO. Checkpoint records are automatically written to the log, and continous checkpointing to the archive database is performed by the RP. Both of these operations are performed in parallel with and asynchronously to MMDB transaction processing. Without stable main memory, pre-commitment of transactions and continous writing to the log buffer provide asynchronous log I/O and transaction processing. With the use of stable memory, a nonvolatile log buffer removes the need for precommitment but achieves a true asynchronous operation of log I/O and transaction execution.

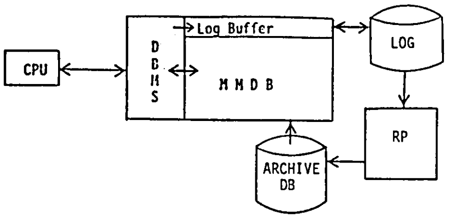


Fig. 2. - Model of Proposed MMDB Recovery System

Summary and Future Research

After defining MMDB recovery, identifying its requirements, and surveying the literature, we have proposed a new MMDB recovery technique which better meets these requirements than previous methods. Our technique requires no transaction UNDO processing after a transaction failure, uses asynchronous I/O and transaction processing to reduce recovery overhead during transaction processing, and provides continuous system checkpoints in parallel with yet no impact on transaction processing. Recent MMDB recovery performance studies have shown that the major factor concerning efficient recovery is the use of stable memory [4,17]. Next to stable memory, the use of additional logging and checkpointing processors can also impact performance [4]. This is the only known technique proposing the use of a special checkpoint processor.

Many areas for future research remain. A major area of study will address the issue of efficient loading for MMDBs including the idea of partitioning. This will examine methods for distributing log and archive database information across multiple secondary storage devices. Along this line we also intend to evaluate various storage techniques to be used for the archive databases.

Currently, a simulation study is being performed to more accurately compare the proposed technique to previous ones. A future paper will more precisely define our proposed technique and present the results of the simulation experiments.

References

[1] Arthur C. Ammann, Maria Butrico Hanrahan, and Ravi Krishnamurthy, "Design of a Memory Resident DBMS," *Proceedings of the IEEE Spring Computer Conference,* 1985, pp. 54-57.

[2] C. J. Date, *An Introduction to Database Systems Volume II,* Addison-Wesley Publishing Company, July 1984, pp. 1-33.

[3] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 1984, pp. 1-8.

[4] Margaret H. Eich, "A Classification and Comparison of Main Memory Database Recovery Techniques," Southern Methodist University Department of Computer Science Technical Report 86-CSE-15, June 1986.

[5] Klaus Elhardt and Rudolf Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM Transactions on Database Systems,* Vol. 9, No. 4, December 1984, pp. 503-525.

[6] Hector Garcia-Molina, Richard J. Lipton, and Peter Honeyman, "A Massive Memory Database System," Princeton University Department of Electrical Engineering and Computer Science Technical Report, September 1983.

[7] Hector Garcia-Molina, Richard J. Lipton, and Jacobo Valdes, "A Massive Memory Machine," *IEEE Transactions on Computers,* Vol. C-33, No. 5, May 1984, pp. 391-399.

[8] Hector Garcia-Molina, Richard Cullingford, Peter Honeyman, and Richard Lipton, "The Case for Massive Memory," Princeton University Department of Electrical Engineering and Computer Science Technical Report 326, May 1984.

[9] J. N. Gray, "Notes on Data Base Operating Systems," *Lecture Notes in Computer Science No. 60,* Springer-Verlag, 1978, pp. 394-481.

[10] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger, "The Recovery Manager of the System R Database Manager," *Computing Surveys,* Vol. 13, No. 2, June 1981, pp. 223-242.

[11] Theo Haerder and Andreas Reuter, "Principles of Transaction-Oriented Database Recovery," *Computing Surveys,* Vol. 15, No. 4, December 1983, pp. 287-317.

[12] IBM, *IMS/VS Version 1 Fast Path Feature General Information Manual,* GH20-9069-2, April 1978.

[13] IBM World Trade Systems Centers, *IMS Version 1 Release 1.5 Fast Path Feature Description and Design Guide,* G320-5775, 1979.

[14] Tobin J. Lehman and Michael J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," University of Wisconsin-Madison Computer Sciences Department Technical Report #605, July 1985.

[15] Tobin J. Lehman and Michael J. Carey, "Query Processing in Main Memory Database Management Systems," *Proceedings of the ACM-SIGMOD International Conference on Management of Data,* May 1986.

[16] Raymond A. Lorie, "Physical Integrity in a Large Segmented Database," *ACM Transactions on Database Systems,* Vol. 2, No. 1, March 1977, pp. 91-104.

[17] Kenneth Salem and Hector Garcia-Molina, "Crash Recovery Mechanisms for Main Storage Database Systems," Princeton University Department of Computer Science Technical Report CS-TR-034086, April 1986.

[18] Joost S. M. Verhofstad, "Recovery Techniques For Database Systems," *Computing Surveys,* Vol. 10, No. 2, June 1978, pp. 168-195.

A Relational Database Machine Organization for Parallel Pipelined Query Execution

Masahito Hirakawa, Kazuyuki Tsuda, Minoru Tanaka, and
Tadao Ichikawa

Information Systems, Faculty of Engineering, Hiroshima University
Shitami, Saijo-cho, Higashi-Hiroshima 724, Japan

Abstract

When we try to implement a relational database system on a conventional von-Neumann computer, we face the gap between the relational model and the computer architecture.

This paper presents a relational database machine organization for the execution of relational algebra operations. The proposed database machine is composed of several processing modules and an interconnection network. The processing module is a special unit and can efficiently execute any relational algebra operation except *selection* on a uniform architecture. For given queries, several processing modules are connected to each other through the interconnection network. Queries are then executed in a parallel pipelined fashion by going through these processing modules.

The effectiveness of the system has been proved through a simulation. The evaluation results are also given.

This proposed system shows sufficient flexibility to be able to cope with the enormous increase in demand for the development of large-scale database machines anticipated in the future. It can also be said that the system organization is relevant from the viewpoint of cost-effectiveness when the recent progress of hardware technologies is considered.

1. Introduction

As the social demand for database utilization expands, implementation of database systems becomes more and more important. When we try to implement a relational database system on a conventional von-Neumann computer, however, we face the gap between the relational model and the computer architecture. This has been accelerating development of special hardware for database manipulations [1], [2].

Database machines developed at an earlier stage, RAP [3] and CASSM [4] for example, execute relational operations directly on the disk by attaching a logic to each disk head. Although these machines contributed to an improvement in execution performance of simple relational operations such as *selection*, still the performance of complex operations such as *join* and *projection* was not satisfactory. Recent approaches toward finding a breakthrough are DIRECT in which relational operations are processed on the internal memory by means of multiple general purpose processors [5], and some special modules for the execution of relational operations [6]-[8].

Meanwhile, the necessity of improving improve the execution performance of relational operations increases as database applications expand. The extension of a relational database system so that it can handle a semantic aspect of data, for example, requires that it is able to execute relational operations efficiently [9], [10].

The authors have proposed a special module which executes *join* efficiently in a parallel fashion [11], and have investigated a processing module for the execution of all other relational algebra operations, and an organization of a relational database machine as the next step.

This paper presents a relational database machine organization for the execution of relational algebra operations. The database machine we propose is composed of processing modules and an interconnection network. Each of the processing modules can efficiently execute any relational algebra operation except *selection* on a uniform architecture. The reason that *selection* is excepted is that each tuple can be processed independently of the others, so an improvement in the execution performance of *selection* should therefore be achieved in connection

CH2345-7/86/0000/1233$01.00 © 1986 IEEE

with secondary storage devices.

For a given query, the system translates it in a form of the relational algebra tree, and after that several processing modules are connected to each other through the interconnection network according to the generated tree. Queries are then executed in a parallel pipelined fashion by going through these processing modules.

In Chapter 2, the organization of the database machine is presented. The architecture of the processing module and algorithms embodied in it for performing relational algebra operations are described in Chapter 3. In Chapter 4, the execution management of the system is described. Finally, in Chapter 5, the evaluation of the system performance is given.

2. System Organization

The organization of the database machine we propose is shown in Fig. 1. The system is composed of multiple processing modules, multiple I/O and overflow buffers, a set of staging buffers, and an interconnection network. Each of the two input lines of the processing module for the input of the operand relations can be connected to any buffer via the interconnection network, and each of the two output lines are directly connected to the I/O and overflow buffers.

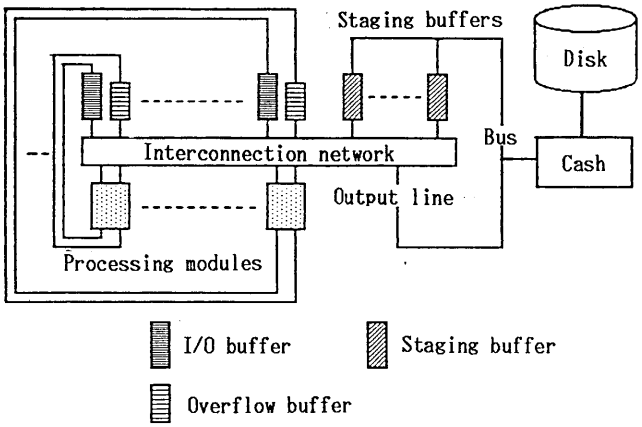


Figure 1 System Organization

The processing module is a special unit and executes relational algebra operations except *selection*, that is, *join*, *projection*, *division*, *union*, *intersection*, *difference*, and *Cartesian produci* on a uniform architecture. The reason that the execution of *selection* has been left out of our investigation is that *selection* can be carried out directly on secondary storage devices. The attachment of a logic to disk heads or the use of indices effectively improves the performance of *selection* [1], [12]. This also contributes to a reduction in the volume of data to be transferred into the staging buffer for processing.

The interconnection network is constructed by 2x2 switching elements in a multistage organization. Each switching elements enables the connection of either one-to-one or one-to-two correspondency between input and output ports. Therefore, it takes one of the four states [13] shown in Fig. 2 .

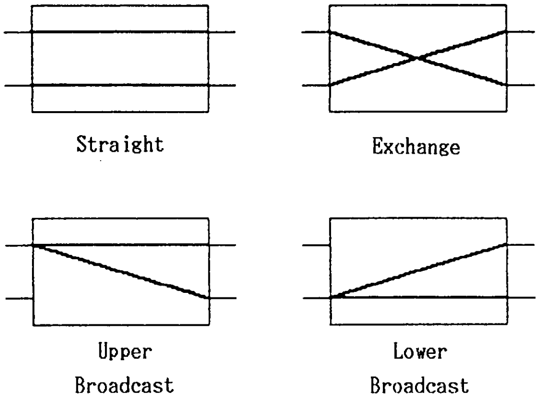


Figure 2 States of Switching Element

Relations to be manipulated are stored in the staging buffer through the cash memory after *selection* has been fully evaluated. In parallel to the evaluation of *selection*, the reduction of attributes which will not be used any more in the following database manipulation is also carried out to reduce the amount of disk I/O. And the resultant relations are fed into the processing modules through the interconnection network. The I/O buffer is used to route the resultant relation of an operation to other processing modules. The overflow buffer is used when memory overflow has occurred. A memory overflow handling scheme will be given in 4.2.

The advantages of the proposed database machine are as follows:

(1) The system is easily constructed, and can easily be extended. There is no need to worry about the correspondency between the processing modules and the specific operations required for the execution of a query in advance. What we have to take into consideration is the number of processing modules

needed in order to get a satisfactory processing performance. Furthermore, the system can easily be extended by attaching additional processing modules.
(2) The unified processing module is functionally flexible and is efficient. Thus, the system shows fairly good adaptability in the execution of arbitrary queries, and can process any type of query efficiently.

3. Dedicated Hardware Module

The processing module is a special unit which is designed to execute seven relational algebra operations, that is, *join*, *projection*, *division*, *union*, *intersection*, *difference*, and *Cartesian product*. Parallel comparison of the condition enables the processing module to execute relational algebra operations efficiently.

3.1 Architecture

Figure 3 shows an organization of the processing module. The processing module consists of S-memory, T-memory and Comparator. Both S-memory and T-memory are divided into *n* segments. Here, S-memory is random-access memory and T-memory is shift-register memory. Two operand relations are input independently to S-memory and T-memory, which will hereafter be referred source relation and target relation, respectively. Each tuple in a relation is stored in a segment, and *n* segment pairs of S-memory and T-memory are compared by means of *n* comparators in parallel.

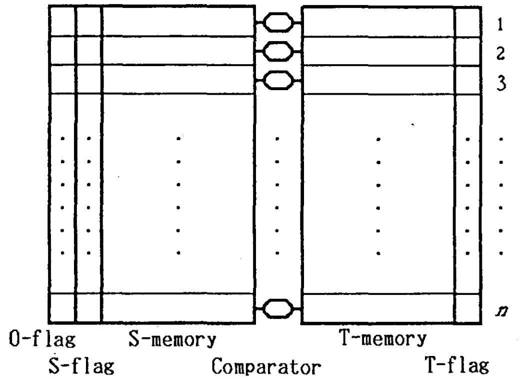


Figure 3 Organization of Processing Module

Furthermore, three flags, that is, S-flag, T-flag, and O-flag are provided for each memory segment pair. S-flag and T-flag are used for the duplication removal of tuples during the execution of *projection*, *union*, *difference*, and *division*. If the flag is on, the same tuple as the tuple indicated must have already existed. O-flag is used to indicate the result of comparison. If the condition of the comparison is satisfied, the corresponding O-flag is set on and the resultant tuple is output.

3.2 Algorithms

In this section, we describe a hardware algorithm provided for the processing module. Before giving details of the algorithm of each relational algebra operation, we will explain a basic algorithm applicable to all relational algebra operations. Let S denote source relation which has an attribute A, and T denote target relation which has an attribute B.

The basic algorithm is listed below:

STEP 0: *INITIALIZATION.*
Initialize the processing module.

STEP 1: *INPUT.*
Input tuples of S and T to the segments of S-memory and T-memory in the following way: For any *i*-th input cycle, the *i*-th tuple of S is input to the *i*-th segment of S-memory, and the *i*-th tuple of T is always input to the 1st segment of T-memory.

STEP 2: *COMPARISON.*
Compare *n* memory segment pairs by means of *n* comparators in parallel. According to the results of the comparison, S-flag and T-flag are changed depending on their preceding states. If the result of comparison meets the output condition, the corresponding O-flag is set on.

STEP 3: *OUTPUT.*
Output the resultant tuples from the segments of S-memory and/or T-memory whose corresponding O-flag are set on. O-flags are set off immediately after the tuples are output.

STEP 4: *SHIFTING.*
Shift the whole contents of both T-memory and T-flag.

STEP 5: *CONDITION for ITERATION.*
Return to STEP 1 when there are still tuples to be input.

STEP 6: *CONDITION for TERMINATION.*
Return to STEP 2 when there are still tuples to be compared. Figure 4 shows the final state when termination will occur.

STEP 7: *TERMINATION.*
End algorithm.

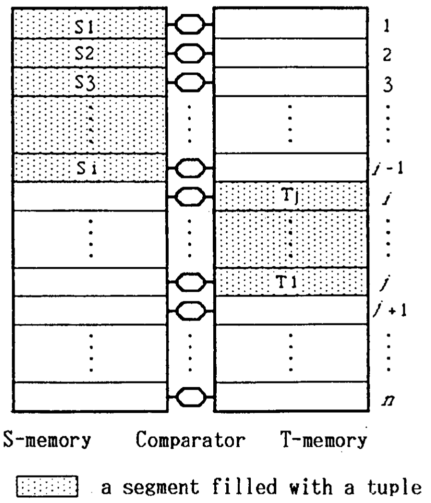


Figure 4 Final state of Processing Module

In the algorithm given above, initialization at STEP 0, operand relations to be served to S-memory and T-memory at STEP 1, and state transition of flags in STEP 2 differ depending upon each operation. In STEP 3, when several O-flags are set on simultaneously, tuples corresponding to the O-flags are output in sequence. However, if the output buffer is large enough to store tuples temporarily, there is no degradation of execution performance. The details of each relational algebra operation are as follows:

(1) ***Projection***: S[A]

In *projection*, S is served to both S-memory and T-memory. Each tuple of S is input to both S-memory and T-memory in STEP 1. In STEP 2, the comparison is made for every pair of segments of S-memory and T-memory, partially on the contents specified by the attribute A under the condition "=". If the condition is satisfied, S-flag, T-flag and O-flag change their states as shown in Table 1 depending on their preceding states. The tuple is output only when O-flag is set on. The state transition specified above prevents duplication of the same tuple output.

Table 1 Transition of Flag States in Projection

S-flag	T-flag	O-flag		S-flag	T-flag	O-flag
0	0	0	⟶	1	1	1
1	0	0	⟶	1	1	0
0	1	0	⟶	0	1	0
1	1	0	⟶	1	1	0

(2) ***Join***: S[A θ B] T

In *join*, S and T are served to S-memory and T-memory, respectively. Each tuple of S and T is input to S-memory and T-memory respectively in STEP 1. In STEP 2, the comparison is made for every pair of segments of S-memory and T-memory, partially on the contents specified by the attributes A and B under the condition θ ("=", "≠", ">", "≥", "<", or "≤"). If the condition is satisfied, O-flag is set on. In STEP 3, tuple pairs whose O-flags are set on are output.

(3) ***Union***: S∪T

Union is regarded as *projection*, where S and T are combined into a single relation and served to both S-memory and T-memory.

(4) ***Difference***: S - T

Difference is regarded as *projection*, where all tuples of T are input to S-memory by setting the corresponding S-flag on in STEP 0. After that, in STEP 1, S is served to both S-memory and T-memory.

(5) ***Intersection*** : S∩T

Intersection is regarded as *join* under the condition that the values of all corresponding attributes are equal to each other.

(6) ***Cartesian product*** : S x T

Cartesian product is considered as *join* with no condition for comparison.

(7) ***Division***: S[A ÷ B] T

Division is carried out in two phases, requiring two modules for the processing. In Phase-1, dividend relation S is grouped according to the value of attribute B of divisor relation T. In Phase-2, intersection among groups of the relation derived from Phase-1 is performed. This process is illustrated in Fig. 5.

The details of the process can be explained as follows:

Phase-1 (grouping)

All tuples of dividend relation S are input to S-memory in STEP 0. In STEP 1, one tuple of relation T is input to the 1st segment of T-memory, and copied for all segments of T-memory. The comparison is made for every pair of segments of S-memory and T-memory under the condition "=", in STEP 2. In STEP 3, the tuples which satisfy the condition are output. STEP 1 through STEP 3 are repeated *m* times by inputing an individual tuple of T at each cycle, where *m* is the cardinality of T. Furthermore, at each end of STEP 3, a special symbol is output as a separator.

Phase-2 (intersection)

In STEP 1, a tuple of the first group of the relation derived from Phase-1 is input to S-memory by setting the corresponding O-flag on, and a tuple of the

S	A1	A2
	A	4
	B	2
	C	1
	A	2
	C	2
	A	3
	B	3
	C	3
	D	1
	E	2

T	B1
	A
	B
	C

phase-1 ⇓ grouping

S'	A1	A2
group A	A	4
	A	2
	A	3
group B	B	2
	B	3
group C	C	1
	C	2
	C	3

phase-2 ⇓ intersection

S÷T	A2
	2
	3

Figure 5 Processing Scheme of *Division*

remains (from second to the last groups) of the relation derived from Phase-1 is input to T-memory. At the comparison in STEP 2, change of the flag states differs depending on whether or not the content of the segments is the separator derived from Phase-1. If it is the separator, the state of the flag changes as shown in Table 2 (a). If not, for the segment pair which satisfies the condition "=", the state changes as shown in Table 2 (b). Meanwhile, STEP 3 is skipped while there are still tuples to be compared. The output is generated only once at the end of the repetition.

Table 2 Transition of Flag States in *Division*

S-flag	O-flag		S-flag	O-flag
0	0	⟶	0	0
1	0	⟶	0	0
0	1	⟶	0	0
1	1	⟶	0	1

(a) For separator

S-flag	T-flag		S-flag	T-flag
0	0	⟶	1	1
1	0	⟶	1	1
0	1	⟶	0	1
1	1	⟶	1	1

(b) For tuple of intermediate relation

4. Query Execution Management

4.1 Parallel pipelined execution of queries

In the system, a given query is translated into a form of the relational algebra tree and put into an execution waiting list. If there are enough processing modules not in use, some of them are assigned for the execution of the query. Here it is assumed that at least one processing module is available for the execution of each relational algebra operation of the relational algebra tree representing the query. Moreover, *selection* is not taken into account in the process of module assignment because the execution of *selection* could be achieved directly in connection with secondary storage devices.

After that, read requests are sent to secondary storage devices for obtaining the relations needed. The relations are transferred to staging buffers page by page. If the query includes *selection*, it is assumed that the *selection* is completed while the transference of the relations is carried out.

During the execution of a query, processing modules are connected to each other through the interconnection network according to the relational algebra tree representing the query. Each processing module gets tuples from the I/O buffer or the staging buffer which is connected to it through the interconnection network, and transfers the resultant tuples to the I/O buffer which is directly connected to it.

The following is an example to explain the management of the query execution. Figure 6 shows a sample query Q1 in a form of the relational algebra tree. Figure 7 explains the process of executing the query in Fig. 6. Here it is assumed that processing modules PM1, PM2, and PM3 are assigned for the execution of *joins* J1 and J2, and *projection* P1, respectively. Furthermore, staging buffers SB1, SB2, and SB3 are assigned for relations R1, R2, and R3, respectively.

Relations R1, R2, and R3 get into to the staging buffers SB1, SB2, and SB3, respectively, after *selection* of R1 and R3 has been fully evaluated. The processing module PM1 gets tuples from SB1 and SB2, and outputs resultant tuples of J1 to the I/O buffer B1. The processing module PM2 gets tuples from B1 and SB3, and outputs resultant tuples of J2 to the I/O buffer B2. And the processing module PM3 gets tuples from B2, and outputs resultant tuples of P1 to the I/O buffer B3, which shows the final result for the given query.

In this way, the processing modules execute a query in a pipelined fashion by transferring tuples through the I/O buffers. Furthermore, the system can process several queries in parallel. Accordingly, queries are executed in a parallel pipelined fashion by means of multiple processing modules.

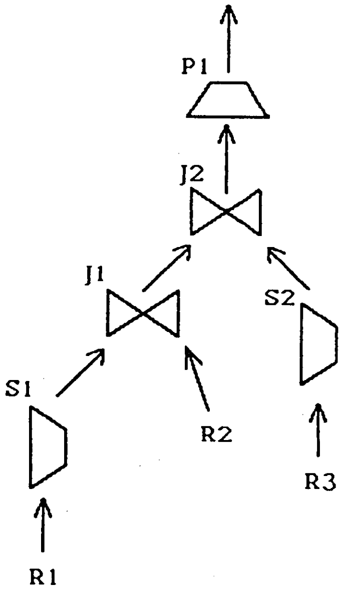


Figure 6 An Example of Query - Q1

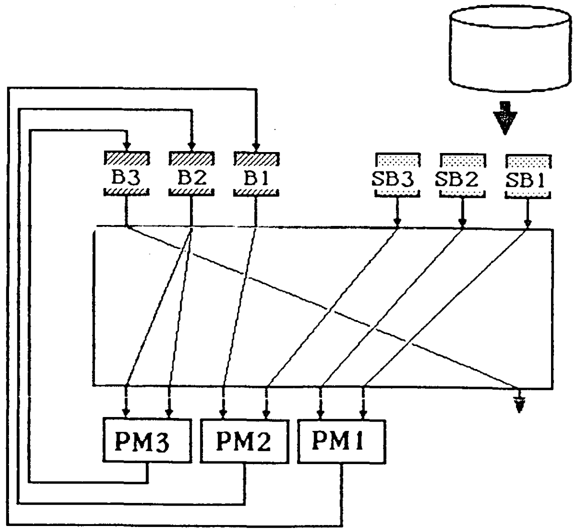


Figure 7 Process of Executing the Query in Fig.6

4.2 Overflow handling

The segment size of a processing module is generally not large enough to contain a whole operand relation, and therefore a countermeasure is required to cope with it. The system offers two types of overflow handling algorithm as follows.

When the cardinality of an operand relation exceeds the number of memory segments, a memory overflow occurs. Here the consideration will be limited to the case of S-memory because the tuples which are shifted out from T-memory must already have been compared with all tuples of S-memory. In the case of a memory overflow, two or more processing modules work together for the execution of a single operation. The tuples in a source relation are sequentially distributed among processing modules with *n* tuples as a unit, and the tuples in an entire target relation are transferred to all processing modules. Here *n* is assumed to be the number of segments in a processing module. The processing is carried out in the internal memory.

Suppose that the number of processing modules necessary for the execution of an operation exceeds the number of available processing modules. We call it a module overflow. In the case of a module overflow, the processing is carried out by limiting the number of tuples in a source relation to the size of S-memories of the processing modules, and the same process is repeated for the remaining source relation. The algorithm described above works externally.

In the following, we explain the memory overflow handling algorithm in detail.

When memory overflow occurs, tuples which have overflowed from T-memory are temporarily stored in the overflow buffer, and connections inside the interconnection network are changed in order to newly utilize an unused processing module. The overflow buffer is used to transfer the overflowed tuples to the additional processing module through the interconnection network. The process of memory overflow handling is described formally as follows (see Fig. 8).

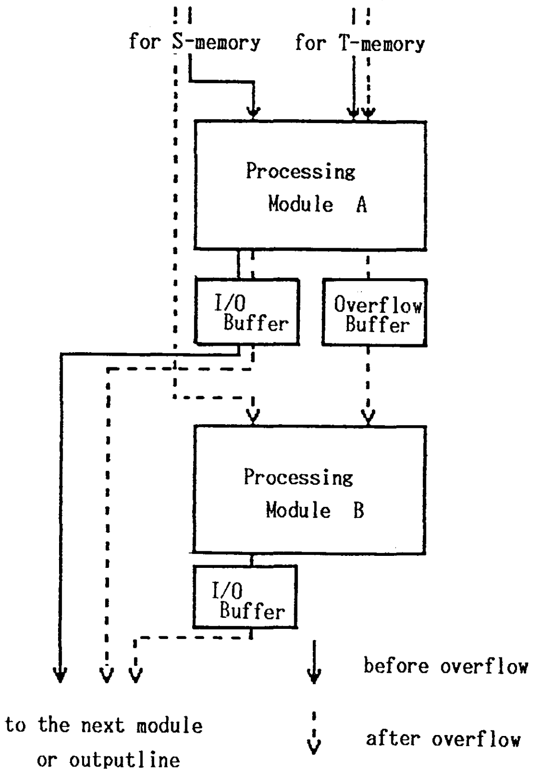


Figure 8 Overflow Handling

The I/O buffer which is connected to the input line for S-memory of the overflowed processing module A will be switched to that of the additional processing module B, and the overflow buffer which is connected directly to the overflowed processing module A will be connected to the input line for T-memory of the additional processing module B. In the overflowed processing module A, only tuples of a target relation are input to T-memory, and the execution continues. Tuples which are shifted out from T-memory are transferred to the overflow buffer. On the other hand, in the additional processing module B, tuples of a source relation and tuples in the overflow buffer are input to S-memory and T-memory, respectively. The execution of the processing is then carried out.

When memory overflow occurs again, the same procedure is invoked. The operation can be executed without any degradation of the execution performance even though the memory overflow occurs, since each processing module executes the operation independently of others. Furthermore, it is possible to manipulate a large relation as long as there are unused processing modules.

The two algorithms described above guarantee the consistency and flexibility of the system in the manipulation of very large databases. Finally, the resultant relation is split into several I/O buffers when the memory overflow occurs, since each processing module transfers the results to its corresponding I/O buffers. This requires the interconnection network to manage recombining of several independent resultant relations into a single relation.

5. Analysis

5.1 Analysis criteria

In this chapter, we will analyze system performance in terms of execution time. The following are the parameters needed for the evaluation.

t_{cb} : page transfer time from cache to buffer
t_{dc} : page transfer time from disk to cache
t_{mc} : message communication time for disk I/O operation
h_r : cache hit ratio (read)
h_w : cache hit ratio (write)
t_m : time to move a tuple between buffer and processing module
t_c : time to compare a pair of tuples
t_s : time to shift tuples
CS : cardinality of source relation
CT : cardinality of target relation
CR : cardinality of resultant relation
n_s : number of segments in a module
n_t : number of tuples in a page
s_p : page size

Then n_s' is defined by n_s multiplied by the number of processing modules used for the execution of a single operation. It is noted that the operation can be executed without any degradation of the execution performance if n_s' is greater than or equal to CS.

The basic tasks used in evaluating the performance of the system are the I/O time, the communication time, and the CPU time.

(1) I/O time

A read request moves a page into a staging buffer from the cache. When the required page is not in the cache, it must be fetched first from a disk. The average cost, T_R, to read a page in the staging buffer is estimated by assuming a cache hit ratio h_r [14].

$$T_R = h_r*t_{cb}+(1-h_r)*(t_{cb}+t_{dc})$$

Similarly T_W, the average cost to write a page, is

$$T_W = h_w*t_{cb}+(1-h_w)*(t_{cb}+t_{dc}),$$

where h_w is a probability for the existence of an available page frame in the cache during a write operation.

(2) Communication time

Page transfer is regarded as I/O operation. Therefore, we take the cost for communicating I/O related (page request and reply) messages into account. This message communication cost, t_{mc}, is added to the I/O cost. Thus, T_R is replaced by T_R+2*t_{mc}, and T_W by T_W+2*t_{mc}.

(3) CPU time

Execution costs of each relational algebra operation are formulated as shown in Appendix I. For a query containing several relational algebra operations, however, it is difficult to formulate the costs for executing it because dynamic aspects of the behavior of the system must be taken into consideration. Therefore, we have implemented a simulation system on a minicomputer. Execution costs of the query are estimated by using it.

5.2 Results

Parameter values which we assume here for the evaluation are as follows: t_{cb}=5[ms], t_{dc}=25[ms], t_{mc}=10[ms], h_r=0.85, h_w=0.35, t_m=64[μs], t_c=128[μs], t_s=0.7[μs], n_s=1024, n_t=256, and s_p=16k[byte].

Table 3 summarizes the execution costs of each relational algebra operation. The sizes of the source and target relations and the size of the resultant relation are changed depending on the operation. In the table, the first row shows the costs for the case that n_s' is large enough to contain a whole relation of S. The second row shows the costs for the case that n_s' is limited to 4096. CS_1 in Phase-2 of *division* denotes the number of tuples of the first group of the relation derived from Phase-1.

Tabel 3 Execution Times (seconds) of Relational Algebra Operations

	Projection	Join	Union	Difference

	CS = 10000
	CR = 7000
		CS = CT = 10000
		CR = 10000
			CS = CT = 10000
			CR = 17000
				CS = CT = 10000
				CR = 7000

$n_s' \geq$ CS	5.7	9.4	12.0	7.5
n_s' = 4096	13.0	14.6	44.5	21.4

	Intersection	Cartesian Product	Division (Phase-1)	Division (Phase-2)

	CS = CT = 10000
	CR = 3000
		CS = CT = 1000
		CR = 1000000
			CS = 10000
			CT = 20
			CR = 5000
				CS = 5000
				CS_1 = 250
				CR = 25

$n_s' \geq$ CS	6.0	392.1	43.3	1.5
n_s' = 4096	11.2	392.1	43.3	2.4

In particular, the execution costs of *join* are shown in Fig. 9, varying the operand relation sizes. The sizes of two operand relations are assumed to be the same i.e., CS=CT, and the size of the resultant relation, CR, is assumed to be the same as CS (and CT). Three curves in the figure show the costs for three different numbers of processing modules. It appears that the costs are proportional to the sizes of the operand relations when the relation sizes are smaller than n_s'.

Next we describe the evaluation result of the performance in executing a query. Each query can be executed independently from others, and hence the evaluation is limited to a single query in the following experiment.

We assume the same query Q1 used in Fig. 6 again for the evaluation, which contains two *joins* and one *projection*. The sizes of the relations served to those two *joins* are assumed to be the same i.e., CS=CT. The sizes of the resultant relations of *joins* are assumed to be the same as CS (and CT), and the size of the resultant relation of *projection* is assumed to be 0.7*CS. The configuration of the system comprises 10 processing modules.

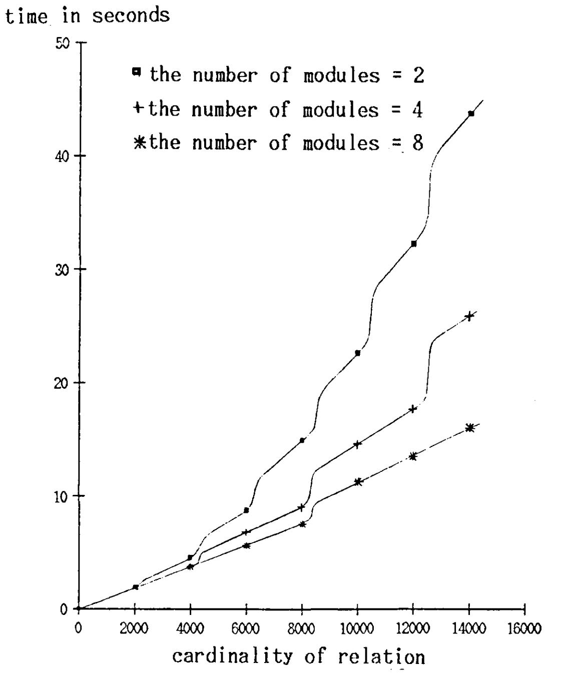


Figure 9 Execution Times (seconds) of Join

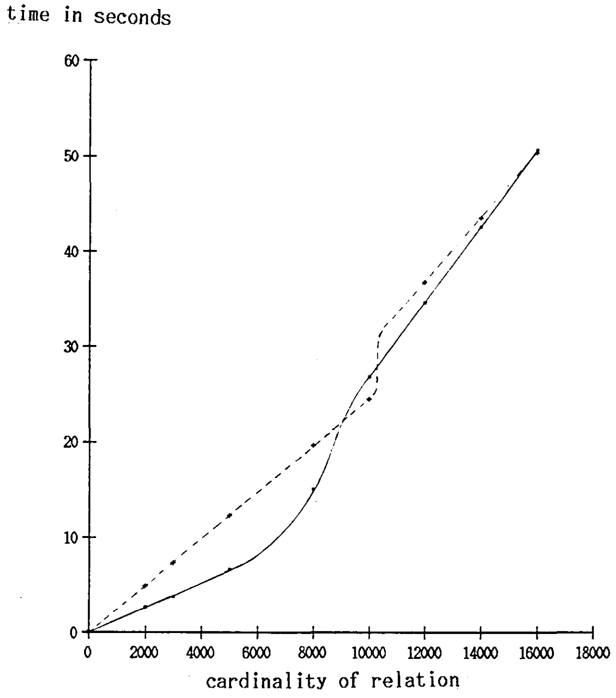


Figure 10 Execution Times (seconds) of the Query in Fig.5

Our result is illustrated in Fig. 10. A solid line curve in the figure shows the cost for the query. A dotted line curve shows the cost when pipelined execution capability is not assumed, and the query Q1 is executed in a sequence of J1, J2, and P1 by using all 10 processing modules for each operation. We would say that pipelined execution is superior when the relation size is smaller than about 10240. This critical value can be estimated in terms of the segment size (1024) multiplied by the number of available processing modules (10). If the relation size is larger than the critical value, the costs based on those two strategies become almost the same because I/O costs significantly influence the total processing costs.

Furthermore, in order to prove the effectiveness of parallel pipelined processing, we compare the execution performance of the query Q1 used in Fig. 6 with that of the query Q2 in Fig. 11 which contains three *joins* and one *projection*. In Q2, *join* J3 is added to Q1. Parameters are assumed to be the same as used in the previous example.

The result of this comparison is shown in Fig. 12. It shows that the performances of the two queries are almost equal to each other, if the cardinality of input relations is smaller than 10000 (almost the total number of segments of the processing modules). Moreover, even though the cardinality of the relations exceeds the total number of segments, it could be said that the system provides relatively good performance by means of parallel pipelined execution.

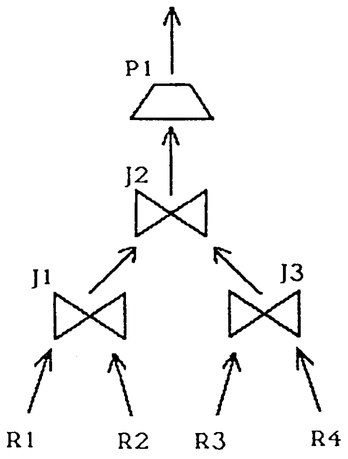


Figure 11 An Example of Query - Q2

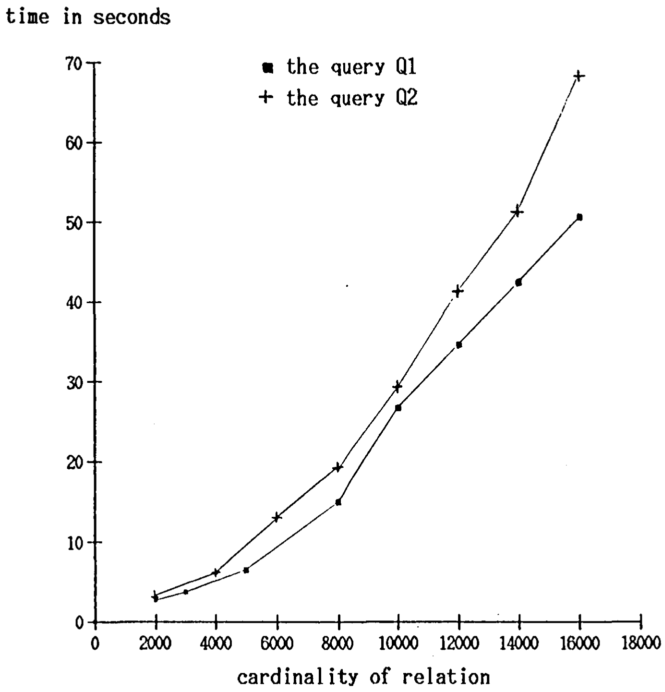


Figure 12 Execution times(seconds) of Q1 and Q2

5.3 Consideration

In this section, we discuss the comparison of the performance for computing equi-join operation in the proposed system and a conventional database management system. Here the conventional system is assumed to use a hashing algorithm. The reason we choose a hashing algorithm is that it is the most suitable when a main memory is large enough to perform the operation [15].

The hashing algorithm used in the conventional system is explained as follows: First, a hash table is constructed by applying a hash function to every tuple of a source relation S independently. S is then partitioned into *m* subsets in such a way that any tuples which have the same hash values lie in the same subset. After the completion of the partition, each tuple of a target relation T is hashed and compared with tuples in the corresponding subset of S by using the hash table. If the comparison is satisfied, pairs of tuples are moved to an output buffer. The cost of this algorithm is shown in Appendix II.

The estimation result is shown in Table 4. It could be said that the proposed system is better than software hash-join method when the number of segments available for the operation is larger than aproximately half the cardinality of S. In the estimation, however, hardware costs are not taken into account. When hardware costs are considered, the software hash-join method may be reasonably acceptable for equi-join operations.

Table 4 Estimation results

CS = CT = CR = 10000
t_{hash} = 40 [μs]

proposed system			hash-jion		
n_S' /CS			*m*		
1	1/2	1/3	10000	8000	5000
9.4 [s]	11.2 [s]	18.3 [s]	12.9 [s]	13.3 [s]	14.2 [s]

However, the application of a hashing technique is limited to equi-join operations. The use of a hashing technique does not effectively reduce the execution cost for non-equi-join operations. In our system, on the other hand, non-equi-join operations can also be processed efficiently.

Furthermore, the system capability could be extended by applying a hashing technique. Both source and target relations are partitioned into *m* disjoint subsets by using a hash function. Each pair of subsets which have tuples with the same hash value is assigned to a processing module, and processed in it. Then an operand can be executed in parallel in cooperation with *m* processing modules. We can improve the performance without preventing the execution in a parallel pipelined fashion.

6. Conclusion

In this paper, the authors have presented a relational database machine which is composed of multiple processing modules and an interconnection network.

The processing module is a special unit for the execution of relational algebra operations. Relational algebra operations except *selection* can be executed efficiently on a uniform architecture. The reason that *selection* has been excepted here is that an improvement in execution efficiency of *selection* should be achieved in connection with secondary storage devices.

For a given query, the system translates it in a form of the relational algebra tree, and after that, several processing modules are connected to each other through the interconnection network according to the tree. The query is executed in a pipelined fashion by going through the processing modules. Furthermore, multiple queries are executed in parallel. Queries can then be executed in a parallel pipelined fashion. When a memory overflow occurs, the system changes the connection of the interconnection network so as to combine two or more processing modules for the execution of a single operation.

Furthermore, the effectiveness of the system is proved through a simulation. Pipelined execution capability greatly contributes to the improvement of the execution performance of the query.

The problems which will remain the future investigation are the scheme of an external storage device management to support efficient data transfer and a mechanism which functions for cases in which the tuple size is greater than the size of the memory segment.

In conclusion, the proposed system shows sufficient flexibility to be able to cope with the enormous increase in demand for the development of large-scale database machines anticipated in the future. It can also be said that the system organization is relevant from the viewpoint of cost-effectiveness when the recent progress of hardware technologies is considered.

Acknowledgement

The authors are grateful to Takashi Nakayama, former graduate student of Hiroshima University, for his contribution to the initial stage of developing this work.

References

[1] G. Z. Qadah, "Database machines: A survey," Proc., AFIPS NCC Conf., Vol. 54, pp.211-223, 1985.

[2] Special issue on database machines, Proc., IEEE Database Engineering, W. Kim, D. Batory, A. Hevner, R. Katz, and D. Reiner, Eds., Vol. 1, pp.5-75, 1981.

[3] E. A. Ozkarahan, S. A. Schuster, and K. C. Smith, "RAP - An associative processor for data base management," Proc., AFIPS NCC Conf., Vol. 44, pp.379-387, 1975.

[4] S. Y. W. Su and G. J. Lipovski, "CASSM: A cellular system for very large data base," Proc., VLDB, pp.456-472, 1975.

[5] D. J. DeWitt, "DIRECT - A multiprocessor organization for supporting relational database management systems," IEEE Trans. on Computers, Vol. C-28, No. 6, pp.395-406, June 1979.

[6] S. Kamiya, K. Iwata, and H. Sakai, "A hardware pipeline algorithm for relational database operation," Proc., Symp. on Computer Architecture, pp.250-257, 1985.

[7] W. Kim, D. Gajski, and D. J. Kuck, "A parallel pipelined relational query processor," ACM Trans. on Database Systems, Vol. 9, No. 2, pp.214-242, June 1984.

[8] M. J. Menon and D. K. Hsiao, "Design and analysis of a relational join operation for VLSI," Proc., VLDB, pp.44-55, 1981.

[9] S. Tsurt and C. Zaniolo, "An implementation of GEM - supporting a semantic data model on a relational back-end," Proc., SIGMOD Conf., pp.286-295, 1984.

[10] T. Ichikawa and M. Hirakawa, "ARES: A relational database with the capability of performing flexible interpretation of queries," IEEE Trans. on Software Engineering, Vol. SE-12, No. 5, pp.624-634, May 1986.

[11] T. Nakayama, M. Hirakawa, and T. Ichikawa, "Architecture and algorithm for parallel execution of a join operation," Proc., Conf. on Data Engineering, pp.160-166, 1984.

[12] D. J. DeWitt and P. B. Hawthorn, "A performance evaluation of database machine architectures", Proc., 7th VLDB, pp.199-213, 1981.

[13] H. J. Siegel, *Interconnection networks for large-scale parallel processing*, Lexington Books, Massachusetts,1985.

[14] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson, "Parallel algorithms for the execution of relational database operations," ACM Trans. on Database Systems, Vol. 8, No. 3, pp.324-353, Sept. 1983.

[15] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, "Implementation techniques for main memory database systems," Proc., SIGMOD Conf., pp.1-8, 1984.

Appendix I

We formulate the execution costs of each relational algebra operation as follows:

PROJECTION

$$T= \frac{CS}{n_t} \times T_R + CS \times t_m + 2 \times CS \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W + 2 \times CS \times t_s$$

$$T'= \left\lceil \frac{CS}{n'_s} \right\rceil \times CS \times t_m + \frac{2 \times CS - n'_s}{n_t} \times T_R + \left\lceil \frac{CS}{n'_s} - 1 \right\rceil \times \frac{CS - n'_s - n_t}{n_t} \times T_R$$
$$+ \left\lceil \frac{CS}{n'_s} \right\rceil \times CS \times t_c + n'_s \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W$$
$$+ \left\lceil \frac{CS}{n'_s} \right\rceil \times CS \times t_s + n'_s \times t_s + \left\lceil \frac{CS}{n'_s} - 1 \right\rceil \times \frac{CS - n'_s - n_t}{n_t} \times T_W$$

JOIN

$$T= \frac{CS+CT}{n_t} \times T_R + \max(CS,CT) \times t_m + (CS+CT) \times t_c + 2 \times CR \times t_m$$
$$+ \frac{2 \times CR}{n_t} \times T_W + (CS+CT) \times t_s$$

$$T'= \left\lceil \frac{CS}{n'_s} \right\rceil \times \max(n'_s, CT) \times t_m + \frac{(n'_s+CT)}{n_t} \times T_R + \left\lceil \frac{CS}{n'_s} - 1 \right\rceil \times \frac{SEG(CT, n'_s+n_t)}{n_t} \times T_R$$
$$+ \left\lceil \frac{CS}{n'_s} \right\rceil \times \max(n'_s, CT) \times t_c + \min(n'_s, CT) \times t_c + 2 \times CR \times t_m + \frac{2 \times CR}{n_t} \times T_W$$
$$+ \left\lceil \frac{CS}{n'_s} \right\rceil \times n'_s \times t_s + \min(n'_s, CT) \times t_s + \left\lceil \frac{CS}{n'_s} - 1 \right\rceil \times \frac{SEG(CT, n'_s+n_t)}{n_t} \times T_W$$

UNION

$$T= \frac{CS+CT}{n_t} \times T_R + (CS+CT) \times t_m + 2 \times (CS+CT) \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W + 2 \times CS \times t_s$$

$$T'= \left\lceil \frac{CS+CT}{n'_s} \right\rceil \times (CS+CT) \times t_m + \frac{2 \times (CS+CT) - n'_s}{n_t} \times T_R + \left\lceil \frac{CS+CT}{n'_s} - 1 \right\rceil \times \frac{(CS+CT) - n'_s - n_t}{n_t} \times T_R$$
$$+ \left\lceil \frac{CS+CT}{n'_s} \right\rceil \times (CS+CT) \times t_c + n'_s \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W$$
$$+ \left\lceil \frac{CS+CT}{n'_s} \right\rceil \times (CS+CT) \times t_s + n'_s \times t_s + \left\lceil \frac{CS+CT}{n'_s} - 1 \right\rceil \times \frac{(CS+CT) - n'_s - n_t}{n_t} \times T_W$$

DIFFERENCE

$$T= \frac{CT}{n_t} \times T_R + CT \times t_m + \frac{CS}{n_t} \times T_R + CS \times t_m + 2 \times CS \times t_c$$
$$+ CR \times t_m + \frac{CR}{n_t} \times T_W + 2 \times CS \times t_s$$

$$T'= \frac{CT}{n_t} \times T_R + CT \times t_m + \left\lceil \frac{CS+CT}{n'_s} \right\rceil \times CS \times t_m + \frac{2 \times CS - n'_s}{n_t} \times T_R + \left\lceil \frac{CS+CT}{n'_s} - 1 \right\rceil \times \frac{CS - n'_s - n_t}{n_t} \times T_R$$
$$+ \left\lceil \frac{CS+CT}{n'_s} \right\rceil \times CS \times t_c + n'_s \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W$$
$$+ \left\lceil \frac{CS+CT}{n'_s} \right\rceil \times CS \times t_s + n'_s \times t_s + \left\lceil \frac{CS+CT}{n'_s} - 1 \right\rceil \times \frac{CS - n'_s - n_t}{n_t} \times T_W$$

INTERSECTION

$$T= \frac{CS+CT}{n_t} \times T_R + \max(CS.CT) \times t_m + (CS+CT) \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W + (CS+CT) \times t_s$$

$$T'= \left\lceil \frac{CS}{n'_s} \right\rceil \times \max(n'_s.CT) \times t_m + \frac{n'_s+CT}{n_t} \times T_R + \left\lceil \frac{CS}{n'_s} - 1 \right\rceil \times \frac{SEG(CT.n'_s+n_t)}{n_t} \times T_R$$
$$+ \left\lceil \frac{CS}{n'_s} \right\rceil \times \max(n'_s.CT) \times t_c + \min(n'_s.CT) \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W$$
$$+ \left\lceil \frac{CS}{n'_s} \right\rceil \times n'_s \times t_s + \min(n'_s.CT) \times t_s + \left\lceil \frac{CS}{n'_s} - 1 \right\rceil \times \frac{SEG(CT.n'_s+n_t)}{n_t} \times T_W$$

CARTESIAN PRODUCT

$$T= \frac{CS+CT}{n_t} \times T_R + \max(CS.CT) \times t_m + (CS+CT) \times t_c + 2 \times CR \times t_m$$
$$+ \frac{2 \times CR}{n_t} \times T_W + (CS+CT) \times t_s$$

$$T'= \left\lceil \frac{CS}{n'_s} \right\rceil \times \max(n'_s.CT) \times t_m + \frac{n'_s+CT}{n_t} \times T_R + \left\lceil \frac{CS}{n'_s} - 1 \right\rceil \times \frac{SEG(CT.n'_s+n_t)}{n_t} \times T_R$$
$$+ \left\lceil \frac{CS}{n'_s} \right\rceil \times \max(n'_s.CT) \times t_c + \min(n'_s.CT) \times t_c + 2 \times CR \times t_m + \frac{2 \times CR}{n_t} \times T_W$$
$$+ \left\lceil \frac{CS}{n'_s} \right\rceil \times n'_s \times t_s + \min(n'_s.CT) \times t_s + \left\lceil \frac{CS}{n'_s} - 1 \right\rceil \times \frac{SEG(CT.n'_s+n_t)}{n_t} \times T_W$$

DIVISION (Phase-1)

$$T= \frac{CS+CT}{n_t} \times T_R + (CS+CT) \times t_m + (CS+1) \times CT \times t_s + CT \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W$$

$$T'= \frac{(CT-1) \times (CS-n_s-n_t) + (CS+CT)}{n_t} \times T_R + (CS+1) \times CT \times t_m + (CS-1) \times CT \times t_s$$
$$+ \left\lceil \frac{CS}{n'_s} \right\rceil \times CT \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W$$
$$+ \frac{(CT-1) \times (CS-n_s-n_t)}{n_t} \times T_W$$

DIVISION (PHASE-2)

$$T= \frac{CS}{n_t} \times T_R + CS \times t_m + 2 \times CS \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W + 2 \times CS \times t_s$$

$$T'= \left\lceil \frac{CS_1}{n'_s} \right\rceil \times CS \times t_m + \frac{2 \times CS - n'_s}{n_t} \times T_R + \left\lceil \frac{CS+CT}{n'_s} - 1 \right\rceil \times \frac{CS - n'_s - n_t}{n_t} \times T_R$$
$$+ \left\lceil \frac{CS_1}{n'_s} \right\rceil \times CS \times t_c + n'_s \times t_c + CR \times t_m + \frac{CR}{n_t} \times T_W$$
$$+ \left\lceil \frac{CS_1}{n'_s} \right\rceil \times CS \times t_s + n'_s \times t_s + \left\lceil \frac{CS+CT}{n'_s} - 1 \right\rceil \times \frac{CS - n'_s - n_t}{n_t} \times T_W$$

Here let T and T' be the execution costs for CS < n_s' and CS > n_s', respectively. SEG(x, y) is the function taking a value either x - y for x > y or 0 for other cases. CS_1 in Phase-2 of *division* denotes the number of tuples of the first group of the relation derived from Phase-1.

Appendix II

We formulate the execution costs of software hash-join algorithm. Here, we take the following parameters for the estimation.

m : the number of subsets of the relation which is partitioned by a hash function

t_{hash} : time to hash a tuple

The cost formulates as follows:

HASH-JOIN

$$T= \frac{CS+CT}{n_t} \times T_R + (CS+CT) \times t_{hash} + 2 \times (CS+CT) \times t_m$$
$$+ \left(\frac{CS}{m} \times \frac{CT}{m} \right) \times m \times t_c + \frac{2 \times CR}{n_t} \times T_W$$

Index to Authors