Palo Alto Research Centers

An Interactive High-Level Debugger for Control-Flow Optimized Programs

Polle T. Zellweger

XEROX

An Interactive High-Level Debugger for Control-Flow Optimized Programs

Polle T. Zellweger

CSL-83-1 January 1983 [P83-00001]

© Copyright Xerox Corporation 1983. All rights reserved.

Abstract: The transformations performed by an optimizing compiler have traditionally impeded interactive debugging in source language terms. A prototype system, called Navigator, has been developed for debugging optimized programs written in Cedar, an Algol-like language. Navigator can be used to monitor program execution flow in the presence of two optimizations: inline procedure expansion and cross-jumping (merging identical tails of code paths that join). This paper describes the problems that these two optimizations create for debugging and Navigator's solutions to these problems. The approach taken is to collect extra information during the optimization phases of compilation. At runtime, Navigator uses the additional information to hide the effects of the optimizations from the programmer.

Categories and Subject Descriptors: D.2.5 **[Software Engineering]:** Testing and Debugging—*debugging aids*; D.3.4 **[Programming Languages]:** Processors—*code generation; optimization*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Interactive debugging, high-level debugging, program optimization

A version of this paper will appear in the Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, March 1983.

This work was partially supported by Stanford University under a contract from the Lawrence Livermore National Laboratory (LLNL Contract 9628303).

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

In order for an interactive high-level debugger to function in a compiler-based programming environment, the compiler must provide mappings between source lines or statements and object code locations, and among variable names, types, and data locations. Program optimization can move or delete statements and eliminate or overlay variables. As a result, variables can be given values at different relative locations in the compiled code than in the source program, or the program's flow of control can be altered. These transformations affect the debugging mappings in ways that have not previously been understood; hence, current optimizing compilers do not support interactive high-level program debugging.

Nevertheless, the ability to apply an interactive high-level debugger to an optimized program is important. Interactive high-level debuggers have long been recognized as useful program development tools [3], and compilers that perform some level of optimization are becoming increasingly common. Some reasons for this trend are the emphasis on portability and modularity in current compiler construction, the increased speed and reliability of optimizing transformations, and the continuing need for efficient use of a computer's time and space resources [5].

This paper describes efficient ways to provide high-level debugging capabilities in the presence of two simple but nontrivial optimizations: cross-jumping and inline procedure expansion. These optimizations are more frequently performed than many global optimizations, largely because they do not require global flow analysis and are therefore more widely implemented in current compilers.

A prototype implementation of these methods, in a system called Navigator, has been developed in the Cedar programming environment at the Xerox Palo Alto Research Center. The Cedar language is an Algol-like language that is very closely related to Mesa [9]. One of the major differences between Cedar and Mesa is that Cedar provides safe automatic deallocation (garbage collection). The compiler modifications needed by Navigator apply to Cedar and Mesa programs, since the Cedar compiler is derived from the Mesa compiler. The approaches described here can assist with interactive high-level debugging of more heavily optimized programs.

2. Interactive high-level debuggers

An interactive high-level debugger helps a programmer examine and control the state of a program during its execution [7]. The user can specify points in the program, called *breakpoints*, at which execution of the program is to be suspended. If the program halts for any reason, whether at a breakpoint or because a runtime error is encountered (e.g., division by zero or illegal memory reference), control is given to the debugger. Typically, the user can then enter commands to discover the current execution point, examine or modify values of variables, or execute new statements in the current context of the program. A *procedure traceback* command is usually

available if the debugger operates on programs written in an Algol-like language. A procedure traceback is a list, in reverse order of invocation, of the currently active procedures and their points of suspension. The debugger may also allow the user to examine or modify the local variables (including parameters) of any active procedure. In a high-level debugger, the programmer communicates with the debugger in terms of the source text. Breakpoint and runtime error locations are specified as positions in the source text. Variables are referred to by their names; their values are displayed in the correct formats for their declared types.

3. Control-flow optimizations

The class of program optimizations considered in this paper are *control-flow* optimizations. These optimizations rearrange the object code for a program either by merging identical code sequences, making the program smaller, or by copying code sequences, making the program faster.

Examples of merging optimizations include procedure discovery [4] and cross-jumping [15]. Procedure discovery locates identical sequences of instructions and forms a single new procedure that is called from each original location. Cross-jumping is a special case of procedure discovery that examines code paths that join. If the tail portions of any two of the paths are the same, cross-jumping moves the join point for those two paths from its original location backward to the earliest identical point and deletes one copy of the identical code. Cross-jumping is often performed as an object code optimization, but it can also be performed on intermediate representations of the program (e.g., flowgraphs or quads [2]). The cross-jumping optimization is illustrated in Figure 1.

Examples of copying optimizations include loop-unrolling and inline procedure expansion [1]. Loop-unrolling makes multiple copies of the statements inside a short loop in order to reduce the effects of loop overhead. Inline procedure expansion, also known as procedure integration, replaces a call to a procedure by an instance of the actual code of the procedure in order to save the execution time associated with procedure linkage (moving parameters, saving and restoring registers, etc.). Inline procedure expansion may also provide opportunities for further optimizations. Inline procedure expansion is illustrated in Figure 2.

These optimizations create problems for interactive high-level debugging, even though they do not alter the order of execution of instructions that are meaningful to the programmer. The problems arise because 1) merging optimizations cause a many-to-one mapping from the source program to the object code, and 2) copying optimizations cause a one-to-many source-to-object mapping. In contrast, a nonoptimizing compiler produces only a one-to-one source-to-object mapping.

A) Source Program Fragment

```
1       IF cond THEN
2        {a ← 1;
3         c ← 5}
        ELSE
4        {a ← 2;

5         c ← 5};
```

Effective Fragment After Cross-jumping

```
1       IF cond THEN
2        {<load 1>;
         GO TO L}
        ELSE
4        {<load 2>;
2,4 L:    <store a>;
3,5       c ← 5};
```

B) Unoptimized Object Code

```
0       LOD cond
2       BEZ L1
4       LOD 1   ◄——— entrance to sequence 1

6       STO a
8       LOD 5        toDelete sequence
10      STO c         (path determiner identifier = 1)

12      BR  L2
14 L1:  LOD 2   ◄——— entrance to sequence 2

16      STO a
18      LOD 5        toRemain sequence
20      STO c         (path determiner identifier = 2)
22 L2:
```

Optimized Object Code

```
0       LOD cond
2       BEZ L1
4       LOD 1   ◄——— path determiner 1

6       BR  L3

8 L1:   LOD 2   ◄——— path determiner 2

10 L3:  STO a
12      LOD 5
14      STO c
16
```

C) Unoptimized Debugger Table

source <->	object	
1	0	
2	4	*statement map*
3	8	
4	14	
5	18	

Debugger actions to set a breakpoint at statement 5:

source 5 -> object 12
  => set primary breakpoint at object 12

object 12 -> source 3_1, 5_2
  => activate determiner 1
    => set determining breakpoint at object 4
  activate determiner 2
    => set determining breakpoint at object 8

Optimized Debugger Tables

source ->	object	
1	0	
2	4	*source table*
3	12	
4	8	
5	12	

object ->	source	
0	1	
4	2	*object table*
8	4	
10	2_1, 4_2	
12	3_1, 5_2	

determiner ->	object	
1	4	
2	8	*determiner table*

Figure 1. Simple cross-jumping example. Throughout this figure and other similar figures, unoptimized items appear on the left, while their optimized counterparts appear on the right.

A) shows the effect of the cross-jumping transformation at the source level. Since Navigator performs cross-jumping on the object code, the resulting program is not wholly expressible in the source language. Abstract stack machine instructions are shown in italics. Some "statements" in the optimized fragment have more than one statement number. Each time that such a "statement" executes, it executes on behalf of exactly *one* of the *unoptimized* statements whose number is listed.

B) presents object code generated from the unoptimized source fragment and illustrates the application of the cross-jumping transformation. When label L2 (at object location 22) is seen, the code preceding the label and preceding the branch to it (at object location 12) are examined for identical sequences by the method of Section 8.3. The resulting object code is shown on the left. The italicized annotations refer to the compile-time path determination bookkeeping described in Section 8.5.

C) shows debugger tables corresponding to the object code above. The unoptimized version is a one-to-one statement map; the optimized tables are more complex. In the source table, statements 3 and 5 both map to the same object location. In the object table, object locations 10 and 12 each have two possible source counterparts, distinguished by subscripted path determiners. The determiner table shows where determining breakpoints must be placed for each path determiner.

The box at the lower left describes the steps the debugger takes to set a breakpoint at statement 5 if cross-jumping has been applied (see Section 7.2 for further details).

A) Source Program Fragment

```
1       PROCEDURE P1 [i: INTEGER] = INLINE
2        {a ← i};

3       PROCEDURE P2 =
4        {P1[1];
5         b ← 2;
6         P1[5 - b]};
```

Effective Fragment After Inline Expansion

```
  3       PROCEDURE P2 =
2,4        {a ← 1;
  5         b ← 2;
2,6         i ← 5 - b;
2,6         a ← i};
```

B) Unoptimized Object Code

```
 0      STO i
 2      LOD i
 4      STO a
 6      RET
 8      LOD 1
10      CAL P1
12      LOD 2
14      STO b
16      LOD 5
18      LOD b
20      SUB
22      CAL P1
```

Optimized Object Code

```
 0      LOD 1
 2      STO a
 4      LOD 2
 6      STO b
 8      LOD 5
10      LOD b
12      SUB
14      STO i
16      LOD i
18      STO a
```

C) Unoptimized Debugger Table

source	<->	object	
1		0	
2		2	*statement map*
3		8	
4		8	
5		12	
6		16	

Debugger actions to set a breakpoint at statement 2:

source 2 -> object 0, 16
=> set breakpoint 1 at object 0
set breakpoint 2 at object 16

Debugger actions to field breakpoint 1:

object 0 -> source 2^1
=> inline 1 -> call source 4, inline name P1

Output to user: **Breakpoint 1 at statement 2 inside P1 called from statement 4 inside P2**

Optimized Debugger Tables

source	->	object	
1		0, 8	
2		0, 16	*source table*
3		0	
4		0	
5		4	
6		8	

object	->	source	
0		2^1	
4		5	*object table*
8		2^2	

inline	->	call source, inline name		
1		4	P1	
2		6	P1	*inline table*

Figure 2. Inline procedure expansion example.

A) shows the effect of the inline procedure expansion transformation at the source level. In contrast with cross-jumping, the resulting program is always expressible in the source language. Statement 2 has two separate copies in the optimized fragment, corresponding to the two calls of P1. The expanded statements can be considered to execute on behalf of a statement from the inline procedure's definition as well as on behalf of the inline procedure's call. In the first call to P1, the parameter (1) is sufficiently simple that the compiler substitutes it directly for uses of i (subsumption). The debugger requires additional mechanisms to display the value of i in that region.

B) presents the object code generated from the two source fragments.

C) shows the corresponding debugger tables. In the source table, statements 1 and 2 each map to two object locations. The superscripts in the object table are references to inline table entries. Each call on an inline procedure generates a separate inline table entry. The inline table records inline call nesting information for displaying a procedure traceback.

The box at the lower left describes the steps the debugger takes to set a breakpoint at statement 2, as well as the steps required to field one of the resulting breakpoints.

4. Objective

Navigator's primary objective is to provide a programming environment in which debugger responses to user requests concerning the execution of an optimized program are the same as the responses would be for an unoptimized version of the program. I call this property *transparent behavior* with respect to a given optimization. A less desirable but still acceptable alternative is to provide *correct behavior* with respect to an optimization. This means that the debugger can display, in source program terms, the relevant changes caused by the optimization at an execution point.

A debugger that has neither transparent nor correct behavior is likely to give confusing responses to queries about an optimized program. For example, if a merging optimization has been applied, the debugger might report an incorrect source location when a breakpoint or error is encountered. If a copying optimization has been applied, the debugger might not place a copy of a breakpoint in each copy of the code. The unmodified Cedar debugger exhibits these problems.

A practical implementation of a system for transparently debugging optimized programs must have two additional properties. First, an optimized program that is capable of being debugged should not be larger or slower than an unoptimized version of the same program. Ideally, adding debugging capabilities for optimized programs will not cost any extra execution time or space unless the debugger is actively responding to a user request. Second, the modified compiler and debugger should still perform reasonably efficiently.

5. Difficult situations for debugging control-flow optimized programs

This section describes the information that a debugger must have to perform the following debugging actions in the presence of control-flow optimizations: setting and fielding breakpoints, reporting the current execution point, displaying values of variables, and providing a procedure traceback.

Setting a breakpoint at a given source statement requires an accurate mapping from the beginning of each source statement to all object locations that represent the start of execution of an instance of the statement. If a single object location represents the beginning of multiple source statements (due to code merging), it must be possible to place multiple logical breakpoints at that location (possibly with different activation conditions).

Reporting the current execution point of a suspended program requires an accurate mapping from each object location to all source statements on whose behalf it executes. This mapping alone provides only enough information for correct debugger behavior, not transparent behavior: when a single object location is a part of the code for multiple source statements, the mapping yields a list of source possibilities rather than just the right one. To achieve transparent behavior, additional work is necessary both during compilation and at runtime. How to accomplish this efficiently is the subject of most of the subsequent discussion. Admittedly, the user can often discover which

statement is really executing by using the debugger to examine values of variables. However, this process can be tedious. If the uninteresting cases occur much more frequently during execution than the desired case does, the user must laboriously check each time to see whether the desired case has finally been reached. (In desperation, the user could recompile with optimization disabled.)

Fielding a breakpoint is my term for deciding whether the current object location corresponds to a source location at which the user has requested a breakpoint. Fielding a breakpoint requires an accurate mapping from each object location at which a breakpoint has been placed to the exact source statement on whose behalf it is executing *this time.*

Finding the local variables of a suspended program can be reduced to discovering the exact source location for a given object location if the source language is lexically scoped. However, if the optimizer applies the subsumption optimization [8] to the formal parameters of an inline procedure, replacing the formal parameters of the expanded procedure by their actual parameter expressions for a given call, the values of the formal parameters may not be available. This problem is not caused by the application of the procedure expansion optimization itself, and hence is outside the scope of this paper.

A procedure traceback should contain exactly the procedural groupings that appear in the source program. Calls to expanded procedures should appear in the traceback as if they had occurred normally; calls to discovered procedures should be hidden. Providing a procedure traceback therefore requires a mapping from each object location to a list of descriptions of the procedures that were expanded to create that object code. Each description must contain the procedure name, the source location of its call, and possibly a symbol table pointer to allow accessing its variables. An inserted call to a discovered procedure must be marked in some way so that the debugger will not include that call in the traceback.

6. Debugger implementation in a conventional setting

In order to support interactive high-level debugging in a compilation environment, a conventional nonoptimizing compiler supplies the debugger with mappings between source statements and object code locations, and among variable names, types, and data locations. The mappings among variable names, types, and data locations are usually encoded in a symbol table.

Two methods of mapping between source statements and object code locations are common. In one method, the generated code for each source statement begins with a call to the debugger, providing the number or some other identification of the source statement as an argument. This method is not very suitable for optimized programs because it can use a significant amount of

execution time and space. In a more efficient method, the compiler creates a separate table that shows the relative program offset of the start of the generated object code for each source statement. This table is called a *statement map.*

A debugger uses a statement map in a straightforward manner. To set a breakpoint at a given source location, the debugger searches the map, finds the source location, and places the breakpoint at the corresponding object location by the usual method of replacing an existing instruction by a trap instruction. To report the current execution point, the debugger finds the nearest preceding object location in the map and reports that the corresponding source statement is executing.

The information for a statement map is typically collected in two stages. First, at the beginning of object code generation for each source statement, the compiler generates a special *source pseudo-instruction* whose operand is the source statement's number. Second, while writing the object code to the output file, each time that the compiler encounters a source pseudo-instruction, it records the current relative program offset and the source pseudo-instruction's operand in the statement map.

7. Overview of Navigator methods for improved debugger behavior

Although the ideas behind the methods described in this paper are general, their implementation depends on the specific source language, compiler, and debugger to which they are applied. This section presents an overview of the compiler and debugger modifications needed to create the Navigator system. The basic idea is quite simple, but many details and complex interactions must be considered in order to provide correct or transparent debugging. The next section will explain many of the complications more fully. The Navigator system always provides correct debugger behavior and usually provides transparent debugger behavior.

The Navigator compiler records and carefully maintains source-to-object correspondence information during its parsing, code generation, and optimization phases. This information is written to debugger tables, allowing the Navigator debugger to perform one-to-many and many-to-one mappings between source locations and object locations. I call the source-to-object map a *source table*, and the object-to-source map an *object table.* These tables replace the conventional statement map.

7.1 Compiler modifications

When the Navigator compiler expands a procedure call in line, it records *inline call information* in an inline procedure table and *inline pointers* in the object table. The inline call information includes the source location of the call and a symbol table pointer for the procedure. This information allows the debugger's procedure traceback command to display an elided call as if it had occurred normally. In the object table, each object location that is the result of an inline

procedure expansion has an inline pointer indicating its appropriate inline call information.

When the Navigator compiler merges program regions via cross-jumping, it records *path determiner locations* in a path determiner table and *determiner pointers* in the object table. Path determiner locations are object locations at which the debugger can optionally collect selective execution history information. In the object table, each source referent in the list of possible source referents for a merged object location has a determiner pointer. The determiner pointer indicates which path determiner must have been executed most recently if the object location is currently executing on behalf of that source statement.

7.2 Debugger modifications

At runtime, the Navigator debugger easily handles a request to set a breakpoint at a statement that has been copied. It uses the source table to map the source location to its corresponding (several) object locations, and it places a breakpoint at each of these locations. When any of the breakpoints is reached, the debugger uses the object table to report the (single) current program location.

A request to set a breakpoint at a statement in a merged region requires more complex processing. First, the Navigator debugger uses the source table to map the source location to its corresponding (single) object location **x**; it places a *primary breakpoint* at **x**. It then consults the object table to create a list **L** of source alternatives for **x**; it places a *determining breakpoint* at each path determiner associated with the source alternatives in **L**. It finds the object locations at which to put these breakpoints by looking in the path determiner table. The process of placing a determining breakpoint at each object location for a given path determiner identifier is called *activating* that determiner.

When a determining breakpoint is reached, the debugger stores a timestamp in a *determination cell* associated with the determining breakpoint's object location. Determining breakpoints are invisible to the user.

When a primary breakpoint at **x** is reached, the debugger examines the values of all of the determination cells associated with **x**. If control flow most recently came through the determiner associated with the source statement of the breakpoint request, this execution of **x** indeed corresponds to the desired breakpoint, and the debugger relinquishes control to the user. Otherwise, execution proceeds as if no breakpoint had been encountered.

8. Detailed description of path determination for merged regions

This section explains the details of Navigator's path determination method for merged regions. The compile-time portions of the method are described in terms of modifications to the cross-jumping algorithm; procedure discovery can be handled similarly. After a few initial definitions, the cross-jumping algorithm is explained. The steps required to update the source and object mappings as a result of the optimization are presented. A simple version of the path determination algorithm is described, followed by a more complicated version that works correctly for repeated cross-jumping. The solutions to several difficulties that arise from interactions between different optimizing transformations are explained. Finally, a series of runtime problems are considered.

8.1 Definitions

In an object code stream, an instruction **x** is the *immediate static predecessor* of an instruction **y** if **x** is the instruction positioned before **y** in the stream. An instruction **x** is an *immediate dynamic predecessor* of an instruction **y** if **x** can execute immediately preceding **y**. That is, either **x** is a jump to **y**, or **x** is the immediate static predecessor of **y** (unless **x** is an unconditional jump to some other instruction). An instruction has exactly one immediate static predecessor; it may have many immediate dynamic predecessors. A *code sequence* is one or more statically adjacent instructions. An *entrance* to a code sequence is any instruction *outside* the sequence that is an immediate dynamic predecessor of an instruction *inside* the sequence.

8.2 Compiler organization and data structures

The Cedar compiler generates object code for a fairly simple stack machine. The compiler performs inline procedure expansion on the parse tree representation of the program. Later, it repeatedly performs several object code optimizations (including cross-jumping) on the generated object code stream until the optimizations are no longer applicable. The other optimizations in this iterative process are: replacing a conditional jump around an unconditional jump by an opposite-sense conditional jump, removing branch chains and jumps to the next location, and examining groups of adjacent instructions for opportunities to delete instructions (notably POPs) or to combine them into a more powerful single instruction. All of the optimizations preserve the actual ordering of computations along any execution path, although the control flow may be altered.

The input to the cross-jumping phase of the compiler is a stream of generated code for a single procedure, represented as a doubly-linked list of two types of cells: *instruction* cells and *marker* cells. Each instruction cell contains a complete description of one machine instruction (either a *jump* instruction or a *code* instruction). A marker cell does not correspond to any object code; it is either a *label* marker or an *info* marker. Each jump instruction has a pointer to its destination cell, which is a label marker. Furthermore, all jumps to the same label are linked. This structure

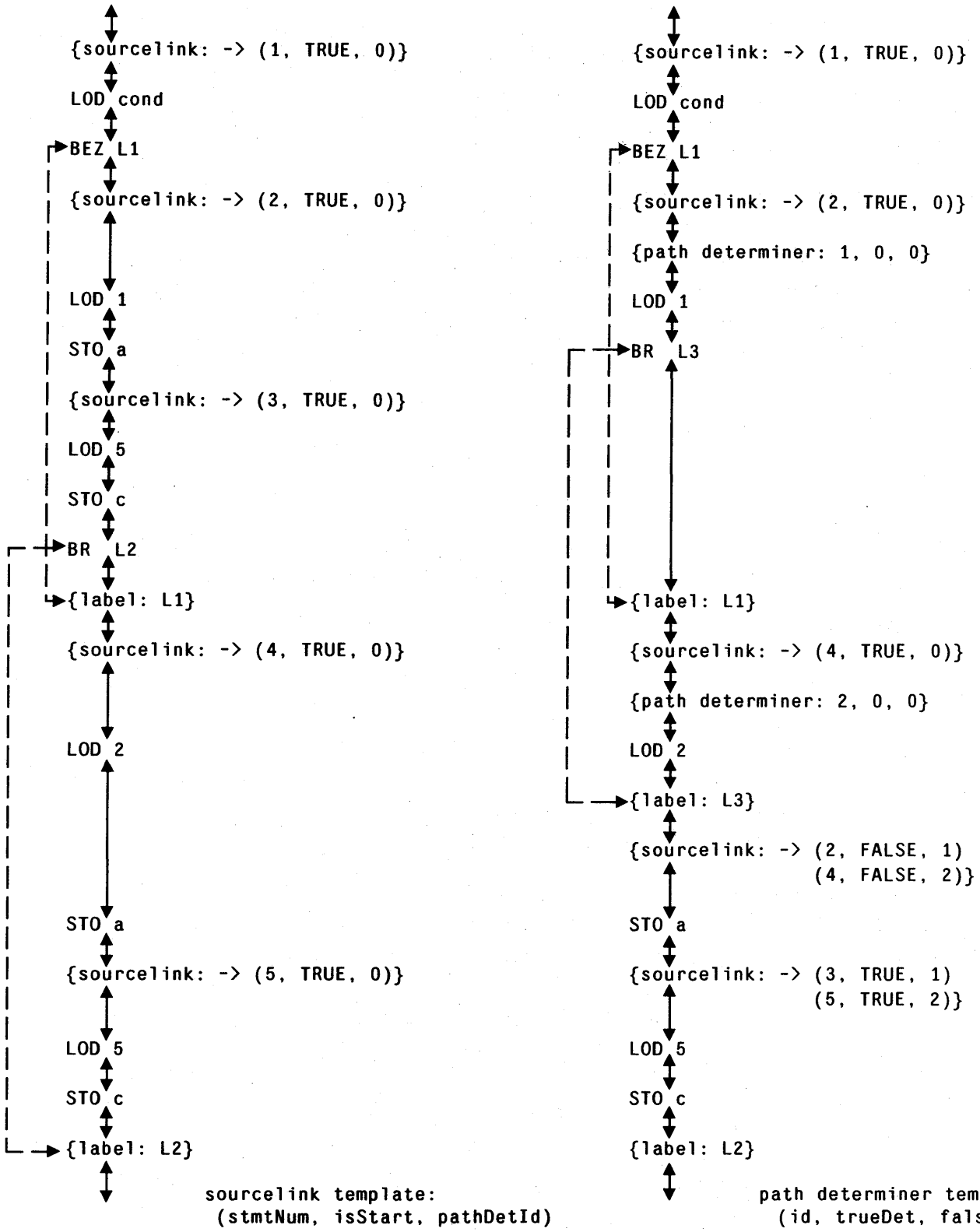

Figure 3. Effects of cross-jumping on the object code stream for the program fragment shown in Figure 1. Marker cells are enclosed in braces {} to distinguish them from instruction cells. Static links between code stream cells are shown as solid lines, dynamic links are shown as broken lines.

The cross-jumping transformation causes several differences between the unoptimized code stream, on the right, and the optimized code stream, on the left. The code between LOD 1 and L1 is replaced by BR L3, and L3 is inserted following LOD 2. The path determination algorithm is responsible for the remaining differences. Path determiners are inserted preceding LOD 1 and LOD 2. A new sourcelink cell is created following L3 to record the merging of a portion of statements 2 and 4. In the sourcelink cell preceding LOD 5, sourcelists for statements 3 and 5 are concatenated to record the merging of those entire statements.

facilitates finding all jumps to a given label, as well as finding a given jump's destination label and all other jumps to that label. Info markers are used to generate statement maps and symbol table information. A sample code stream is shown in Figure 3.

One type of info marker is the sourcelink cell. Every instruction between two sourcelink cells has the relationship to the source text that is described by the statically preceding sourcelink cell. A sourcelink cell is a pointer to a list of records called a sourcelist. Each sourcelist entry contains a source statement number `stmtNum`, a boolean variable `isStart` that indicates whether the next instruction is the start of the object code for that source statement, and a path determiner identifier `pathDetId`.

Sourcelink cells therefore contain sufficient information for the construction of both source and object tables. At the start of the object code optimization phase, all sourcelink cells have a single element in their sourcelist, with `isStart` true.

8.3 Cross-jumping optimization

The Navigator compiler's version of the cross-jumping algorithm examines the object code stream sequentially. When it finds a label, each immediate dynamic predecessor of the label defines the end of a *path* to that label. The algorithm compares paths pairwise for identical tail code sequences. It finds identical code sequences by searching the static paths in reverse order until unequal instructions are encountered.

When the algorithm finds identical code sequences, it designates one code sequence the *toDelete* sequence, and it designates the other the *toRemain* sequence. It inserts a jump from the beginning of the toDelete sequence to the beginning of the toRemain sequence, and then it deletes the toDelete sequence.

If the toDelete code sequence has internal labels, the algorithm is slightly more complex. As the backward comparison scan crosses such a label, it redirects each jump to that label to the corresponding point in the toRemain code sequence (it inserts a new label there, if necessary). This allows longer code sequences to be found in a single application of cross-jumping. Figure 4 presents an example of jump redirection.

8.4 Mapping between many source statements and one object location

Because the source table is only concerned with the object location for the start of a source statement, retaining information for later construction of the source table is straightforward. When the compiler encounters a sourcelink cell in the toDelete code sequence during the backward scan, it need only move the sourcelink cell to the current position in the toRemain sequence (sourcelists of resulting adjacent sourcelink cells are merged). When statement boundaries are in different places in the two sequences, or if cross-jumping merges only a portion of the object code for a statement, retaining information for the object table is more complicated.

A) Source Program Fragment

```
1       IF cond1 THEN
2        {IF cond2 THEN
3           a ← 1;
4         c ← 5}
        ELSE
5        {a ← 2;
6         c ← 5};
```

Effective Fragment After Cross-jumping

```
1       IF cond1 THEN
2        {IF cond2 THEN
3          {<load 1>;
            GO TO L1};
4         GO TO L4}
        ELSE
5        {<load 2>;
3,5 L1:   <store a>;
4,6 L4:   c ← 5};
```

B) Unoptimized Object Code

```
0       LOD cond1
2       BEZ L1
4       LOD cond2
6       BEZ L2      <-- entrances to sequence 1
8       LOD 1       <--

10      STO a
12 L2:  LOD 5       toDelete sequence
14      STO c       (path determiner identifier = 1)

16      BR  L3
18 L1:  LOD 2       <-- entrance to sequence 2

20      STO a
22      LOD 5       toRemain sequence
24      STO c       (path determiner identifier = 2)
26 L3:
```

Optimized Object Code

```
0       LOD cond1
2       BEZ L1
4       LOD cond2
6       BEZ L4      <-- branch redirected
8       LOD 1

10      BR  L5

12 L1:  LOD 2

14 L5:  STO a
16 L4:  LOD 5
18      STO c
20
```

C) Unoptimized Debugger Table

source	<->	object	
1		0	
2		4	*statement map*
3		8	
4		12	
5		18	
6		22	

Debugger actions to set a breakpoint at statement 4:

source 4 -> object 16
  => set primary breakpoint at object 16

object 16 -> source 4_1, 6_2
  => activate determiner 1
    => set determining breakpoints at objects 6 & 8
  activate determiner 2
    => set determining breakpoint at object 12

Optimized Debugger Tables

source	->	object	
1		0	
2		4	*source table*
3		8	
4		16	
5		12	
6		16	

object	->	source	
0		1	
4		2	*object table*
8		3	
12		5	
14		3_1, 5_2	
16		4_1, 6_2	

determiner	->	object	
1		6, 8	
2		12	*determiner table*

Figure 4. Cross-jumping example with multiple entrances.

A) shows the effect of the cross-jumping transformation at the source level.

B) demonstrates the application of the cross-jumping transformation. In this example, the toDelete sequence contains a label (L2, at object location 12). The cross-jumping algorithm inserts a new label L4 at the corresponding location in the toRemain sequence, and redirects the branch to L2 (BEZ L2, at object location 6) to L4.

C) presents the debugger tables corresponding to the object code above. In the determiner table, note that determiner 1 has two associated object locations.

The box at the lower left describes the steps necessary inside the debugger to set a breakpoint at statement 4 if cross-jumping has been applied. A total of four breakpoints are required.

The compiler retains sufficient information in the sourcelink cells to create both the source table and the object table. The following cases specify the bookkeeping necessary when the backward scan encounters a sourcelink cell in either code sequence.

1. If a sourcelink cell is encountered in the toRemain code sequence, the toDelete code sequence is searched for the nearest sourcelink cell preceding the current point. The sourcelist from the toDelete code sequence is appended to the sourcelist from the toRemain code sequence. If the search crossed an instruction cell, the `isStart` field of each element of the toDelete code sequence sourcelist is set to false as it is appended.
2. If a sourcelink cell is encountered on the toDelete code sequence only, the toRemain code sequence is searched for the nearest preceding sourcelink cell. That cell is copied to the current point in the toRemain code sequence, setting `isStart` to false for each element of the sourcelist. The sourcelist from the toDelete code sequence is then appended to that sourcelist.

In order to fully reflect the relationship between the source text and the newly merged object code, there must be sourcelink cells that correctly describe the first piece of object code in the new merged sequence, the first piece of object code following the toRemain sequence, and the first piece of object code following the toDelete sequence. If these locations do not already have sourcelink cells, appropriate sourcelink cells must be created.

8.5 Simple path determination

To permit the debugger to choose the correct source location when fielding a breakpoint at a merged location, the compiler must record not only that a given section of object code now corresponds to multiple source statements, but how control could pass into that code section. Path determiners supply the necessary extra information.

For each identical sequence, the compiler creates a unique *path determiner identifier.* When it encounters a sourcelink cell, it sets its `pathDetId` field to the path determiner identifier for that sequence. Furthermore, it marks each entrance to the sequence with the same path determiner identifier by inserting a new type of info cell, called a *path determiner cell,* in the code stream immediately preceding each entrance to the sequence.

As the compiler emits object code, it records an (object location, path determiner identifier) pair in a *path determiner table* for each instruction that is marked with a path determiner identifier. The set of all object locations that have the same path determiner identifier is called a *path determiner.*

A) Source Program Fragment

```
 1 IF cond1 THEN
 2    {x ← 1;
 3     a ← 2;
 4     IF cond2 THEN
 5        {b ← 3;
 6         d ← 4}
       ELSE
 7        d ← 4;
 8     e ← 6}
   ELSE
 9    {y ← 8;
10     a ← 2;
11     IF cond2 THEN
12        {b ← 3;
13         d ← 4}
       ELSE
14        d ← 4;
15     e ← 6};
16 f ← 7;
```

Intermediate Step

```
 1 IF cond1 THEN
 2    {x ← 1;
 3     a ← 2;
 4     IF cond2 THEN
 5        {b ← 3;
          GO TO L1}
       ELSE
6,7 L1:  d ← 4;
 8     e ← 6}
   ELSE
 9    {y ← 8;
10     a ← 2;
11     IF cond2 THEN
12        {b ← 3;
          GO TO L2}
       ELSE
13,14 L2:  d ← 4;
15     e ← 6};
16 f ← 7;
```

Effective Fragment After Cross-jumping

```
 1 IF cond1 THEN
 2    {x ← 1;
      GO TO L3


   ELSE
 9    {y ← 8;
3,10 L3: a ← 2;
4,11     IF cond2 THEN
5,12        {b ← 3;
            GO TO L2}
         ELSE
6,7,13,14 L2:  d ← 4;
 8,15     e ← 6};
16 f ← 7;
```

B) Annotated Flowgraphs

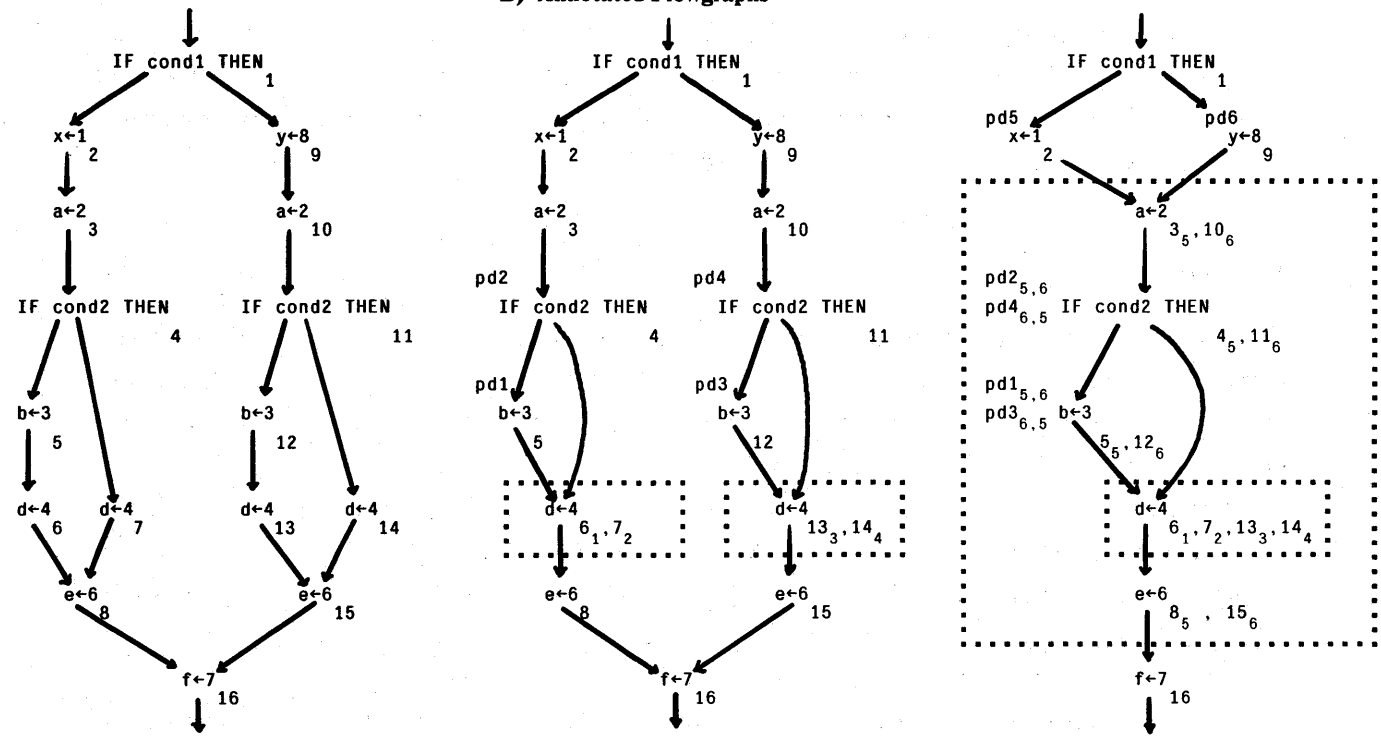

Figure 5. A complicated case of repeated merging.

A) shows the effect of repeated application of the cross-jumping transformation on a source fragment. The first step merges statement 6 with statement 7 and statement 13 with statement 14, while the second step merges statements 3 through 8 with statements 10 through 15. This multiple merging can be performed during a single sequential pass through the object code. A fine point: when `L2` is seen during the final merge, a copy of determiner 6 is inserted before `GO TO L2` (similarly, `L1` causes a copy of determiner 5 before `GO TO L1`). When `GO TO L2` (`GO TO L1`) is discovered to be part of the merging sequence determined by 6 (5), those determiners are removed because they do not represent *entrances* to the merging sequences.

B) illustrates the same transformations on flowgraphs. Merged regions are enclosed in dotted lines. A statement's sourcelink cell is shown at its lower right (subscripts are `pathDetIds`). Path determiners marking a statement are shown at its upper left (subscripts denote `trueDet, falseDet`). Consider the statement `d←4`. It executes on behalf of statement 7 if determiner 2 has a later timestamp than determiners 1, 3, and 4. However, determiners 2 and 4 always have the same timestamp value, as do 1 and 3. This ambiguity is resolved by examining the `trueDet` and `falseDet` determiners: if determiner 5 has a later timestamp than determiner 6, path determiner 2 is considered true; otherwise determiner 4 is true. Therefore, statement `d←4` executes on behalf of statement 7 if the sequence of execution is `x←1; a←2; IF cond2 THEN; d←4`.

8.6 Full path determination

If cross-jumped regions were never considered for further application of the cross-jumping algorithm, the path determination procedure described in the previous section would be adequate. However, not only can a single pass through the generated object code create multiply merged regions, but the compiler applies the cross-jumping algorithm and other object code optimizations to the code stream repeatedly, until no changes occur. These iterated optimizations cause additional compile-time complications.

The problems arise if a candidate sequence for a new merge (either the toDelete sequence or the toRemain sequence) contains either a portion of a previously merged region or an entrance to a previously merged region. Figure 5 shows an instance of repetitive merging.

If a candidate sequence contains a portion of a previously merged region, the candidate sequence contains at least one sourcelink cell with more than one element in its sourcelist. Each element of such a sourcelist mentions a path determiner identifier for a previously merged sequence in its `pathDetId` field. The algorithm does not mark these sourcelink cells with the new path determiner identifier for the candidate sequence. Instead, the debugger will rely upon the path determiners specified in the unmodified candidate sequence's sourcelist to distinguish among all of the candidate sequence's source possibilities. As before, the compiler merges the candidate sequence sourcelist with the sourcelist (which may or may not have multiple elements) from an appropriate sourcelink cell in the other sequence participating in the new merge.

Now consider the case in which a candidate sequence contains an entrance to a previously merged region, i.e., the candidate sequence contains a path determiner cell. It is tempting to suppose that the compiler could delete the path determiner cell from its current location and insert a copy of it at each entrance to the candidate sequence. Unfortunately, there are programs for which this strategy fails to provide full path determination. These programs are characterized by multiple control-flow paths that are wholly contained in a merged region. Such a program is shown in Figure 5. Suppose that the second cross-jumping step in that figure were to move determiners 1 and 2 up to the location of determiner 5 (and similarly move determiners 3 and 4 up to the location of determiner 6). It would then be impossible to distinguish between the execution of statements 6 and 7, or between statements 13 and 14.

The concept of merging path determiners from two identical sequences is similar to merging sourcelink cells from two identical sequences: a marker logically belongs to one of the two sequences, and this fact must continue to be represented in the new merged region to be created. (The two situations differ in that a path determiner actively gathers runtime information, while a source marker becomes a passive table entry.) As described above, the solution for the sourcelink cell is to mark it with the appropriate path determiner identifier. This solution works for path determiners also, but with a different marking method: the algorithm marks the path determiner cell with the path determiner identifier of the candidate sequence by setting the `trueDet` field of the path determiner cell to that path determiner identifier. Since the debugger must be able to tell whether execution came to the path determiner along the candidate sequence or along the other

sequence participating in the new merge, a path determiner cell also has a `falseDet` field in which the path determiner identifier of the other sequence is recorded. Finally, if the candidate sequence is the toDelete sequence of this merge, the algorithm moves the path determiner cell to the current point in the toRemain sequence.

To see that this algorithm works, suppose that an object location x has a possible path determiner **p2** whose `trueDet` field is **p1**. At runtime, the execution of x corresponds to an entrance **p2** to an inner merged region if and only if the most recent entry to the outer merged region came through an entrance whose path determiner is **p1**.

Dealing with the added complexity of repeated cross-jumping requires some runtime modifications. The following path determiners are activated as a result of a breakpoint request: first, all path determiners that mark a source alternative, and second, (recursively) any path determiner that is a `trueDet` or a `falseDet` of a previously noted determiner. Timestamp checking at a primary breakpoint also reflects determiner nesting: a timestamp value in a determination cell is valid only if the timestamp value of its `trueDet` determiner is not earlier than the timestamp value of its `falseDet` determiner.

Timestamps also figure in another complication. Suppose that the user requests a breakpoint in a merged region *after control has already entered that region.* This situation might occur when single-stepping, for example. Since path determiners can cover the final merged region in a fairly baroque way, a breakpoint can activate determiners for part of a region without activating determiners for some other part of the region. Therefore, the debugger must record the time that each determining breakpoint is set. At each primary breakpoint, the debugger checks the timestamps and the set times to ensure that all necessary determining breakpoints were set early enough. If the set time for a determining breakpoint x is later than the latest timestamp for the remaining determining breakpoints, the debugger includes the source statement corresponding to x in the list of source possibilities.

Each iteration of the cross-jumping algorithm is a transformation from one correct representation of the program to another; the described modifications correctly reflect the effects of that transformation on the mappings between source statements and object locations. Because the transformation is applied repeatedly, the number of path determiner cells inserted in the program is not minimal. If the cross-jumping algorithm could merge multiple paths at once, or if the order of merges could be optimally arranged, fewer path determiner cells might result. It is possible that a post-optimization pass over the code stream and the path determiner table could coalesce multiple path determiners and achieve the minimal number. This minimization step could also create equivalence classes of path determiners for each final merged region, eliminating the need to record the time that each determining breakpoint is set.

Analysis of the current cross-jumping algorithm shows that the number of inserted path determiners is linear in the number of merging paths. Proofs that correct path determination can always be performed using these algorithms are presented in [16].

8.7 Interactions with other optimizations

Because inline procedure expansion occurs earlier in the compilation process than the object code optimizations, an expanded inline procedure can be cross-jumped with statements outside that inline procedure (but inside the calling procedure). Even in these cases, Navigator's mechanisms for handling code merging allow setting and fielding breakpoints and reporting the current program location. However, the possibility that a program region can be copied and then merged constrains the solution for providing a procedure traceback; each sourcelink cell must have a separate inline table pointer, since that statement may be moved away from the other statements resulting from the same expansion.

A more serious problem is that other control-flow optimizations can occur *after* cross-jumping. Path determiner cells are intended to be ignored by compiler routines (including other optimizations) that operate on the object code stream. The difficulty arises when one of these later optimizations alters the code stream so that path determiner cells no longer mark each of the ways to enter the merged region. One solution to this problem would be to check for the presence of path determiner cells and inhibit the detection of further optimizing patterns. However, this is not sufficient for optimizations such as branch-chain removal, in which lexically surrounding instructions are not examined. A better (although ad hoc) solution involves ascertaining how each optimization affects the placement of path determiner cells and individually inhibiting unanalyzed or troublesome optimizations.

8.8 Debugger difficulties

Runtime complications are not limited to the problems caused by iterated optimization. For example, two paths that contain a directly or indirectly recursive call can be cross-jumped. Hence determination cells must refer to a particular invocation of a procedure. This also takes care of multiple processes executing a cross-jumped procedure.

The allocation of determination cells presents another problem. If the compiler were to allocate any necessary determination cells in the procedure's local frame, the association between determination cells and a particular procedure invocation would be implicit. However, this strategy would be quite expensive, as runtime space would be consumed even for procedures in which no breakpoints had been set. Therefore, when a breakpoint is placed in a merged region, the debugger allocates determination cells in a memory region of its own. The timestamp value in each determination cell is associated with a single procedure invocation by having a pointer to the activation frame for that procedure invocation.

9. Features and drawbacks of path determination

The path determination method usually provides transparent debugging capabilities for control-flow optimized programs. In addition, it has the practical properties that we desire. Runtime path determination costs execution time or space (excluding space for the tables, which need not reside in main memory) *only* if a breakpoint has been placed in a merged region. The cost is proportional to the number and merging complexity of merged regions that have breakpoints set inside them.

It is clear that there are cross-jumped regions for which exact path determination is not possible. If an entire path generates the same object code as some other joining path, both path determiner cells will be placed at the same place. Figure 6 illustrates this situation. These programs are probably not very interesting in practice, but if they occur, the path determination algorithm can correctly identify the source alternatives. Since path determiners are identified during the optimization process, the lack of distinct places to put them could be used to inhibit complete merging of two paths by cross-jumping.

The method has another drawback. Since path determiners generate no code, they must be activated in order to provide exact path determination. Thus, if a runtime error or interrupt is encountered inside a merged region, Navigator can only list the source alternatives. Similarly, if a merged region contains a procedure call, procedure tracebacks that include that area can only list the source alternatives. In some cases, the user can inspect the values of variables to determine the exact path. If this is not sufficient, the user need only restart the execution with a breakpoint at the offending location, rather than recompile the program with cross-jumping disabled. It might be useful in such instances to decouple path determiner activation from breakpoint insertion. A proposed new Navigator command would explicitly activate all path determiners within a suspect procedure.

Source Program Fragment	Effective Fragment After Cross-jumping
`1 b ← 5;`	`1 b ← 5;`
`2 IF cond THEN`	
`3 a ← 1`	
`ELSE`	
`4 a ← 1;`	`2,3,4 a ← 1;`

Figure 6. Program fragment with undeterminable paths.

10. A different runtime path determination algorithm

A debugger can use the path determiners identified in Section 8 in a different way to distinguish among source alternatives in a merged region at runtime. The compiler could *generate code* at each determiner to load some cell with an indication that control flow came that way. This method would always provide transparent debugging, but it would increase object code size by at least n stores for every n-way merge, and it would also increase data space. Unfortunately, this would probably consume more space than the cross-jumping algorithm saved. Moreover, the optimized program would run more slowly than the unoptimized version. These space and speed penalties could be largely avoided if hardware for execution tracing were available. The information gathered by the execution tracing hardware could be searched for the most recent appearance of a path determiner location. However, any execution history recording mechanism that uses a fixed-size storage area could fail to distinguish among multiple paths in a merged region. As an example, consider the program in Figure 7, and suppose that the statements inside `Proc` generate a large amount of history information.

Source Program Fragment	Effective Fragment After Cross-jumping
1 IF cond THEN	1 IF cond THEN
2 {Proc[a];	2 {*<load a>*;
3 Write["hi"]}	GO TO L
ELSE	ELSE
4 {Proc[b];	4 {*<load b>*;
	2,4 L: *<call Proc>*;
5 Write["hi"]}	3,5 Write["hi"]}

Figure 7. Difficult case for hardware-supported execution-history mechanism.

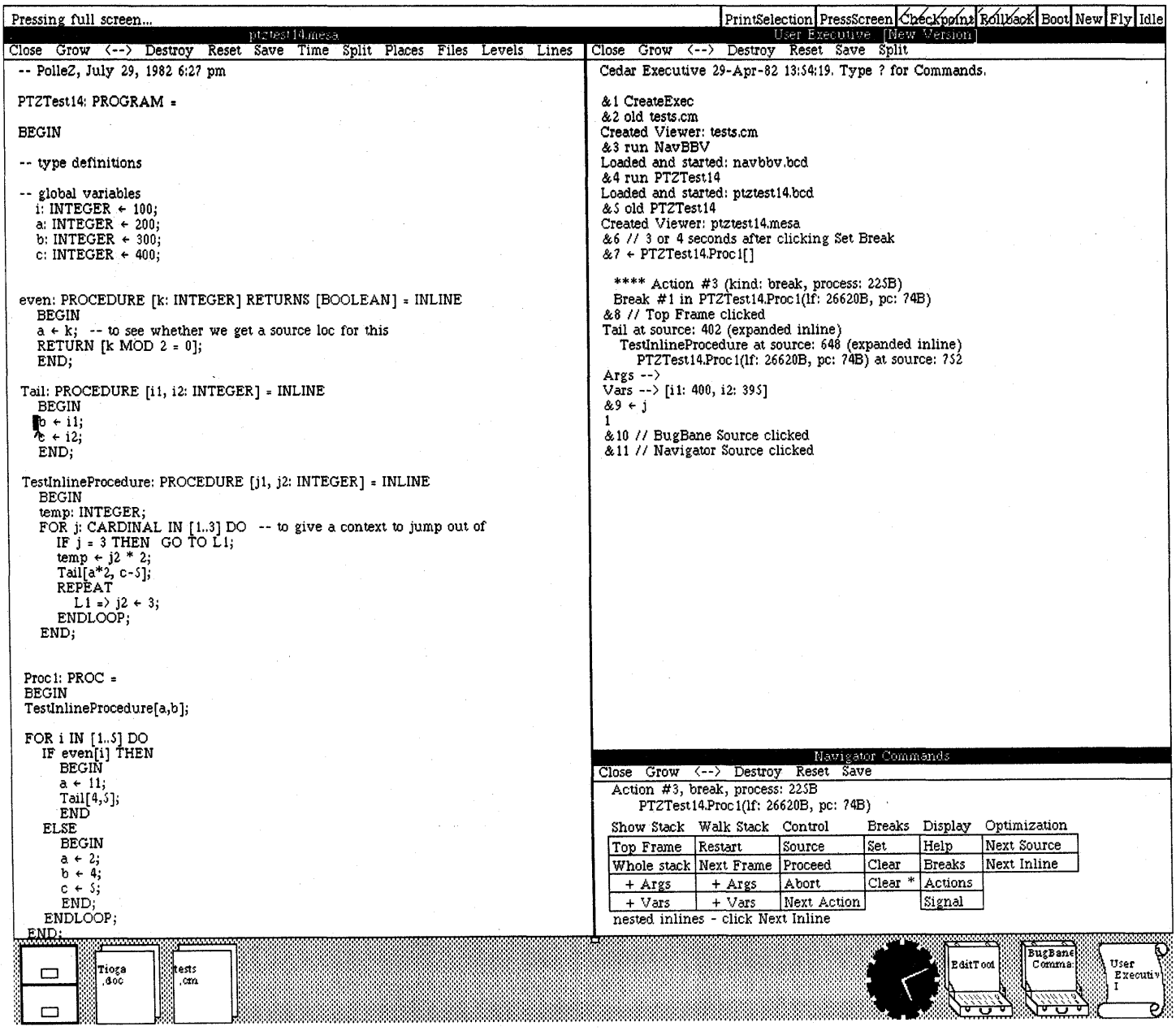

Figure 8. Navigator Operation.

11. Current state of the implementation

Navigator has a running prototype. The compiler modifications for Navigator comprise approximately 1500 lines of Cedar source code, or about 3% of the size of the original Cedar compiler. Navigator's runtime routines add about 1000 lines of source code to the debugger. Preliminary evaluation of the untuned completed system shows that extra compilation and execution costs of using Navigator are small: the compiler is roughly 15% slower, and debugger responses are not noticeably altered.

Figure 8 shows an optimized Cedar program being debugged using Navigator. The display is used in conjunction with a mouse (a pointing device) [10]. The area (window) of the screen labelled **Navigator Commands** contains a command menu for the Navigator debugger. Commands are organized into related command columns; this paragraph refers to a command by ***command column/command name***. A limited amount of user feedback appears in the **Navigator Commands** window. User programs are invoked from the **User Executive** window; debugger commands like **Display/Breaks** or **Show Stack/Top Frame** print their results there. The source text of the program being debugged is also on the screen in the **ptztest14.mesa** window. The user sets a breakpoint by selecting one or more source characters and clicking a mouse button over the **Breaks/Set** command area; the current program location is displayed by a highlighted area in the source text when the **Control/Source** area is clicked. The commands in the **Optimization** column (an additional nontransparency of the Navigator debugger) are necessary because the window system can only highlight one contiguous screen region at a time.

12. Relationship to other work

The cross-jumping algorithm was inspired by a method sketched by Teeple and Anderson in an unpublished manuscript [12]. In their approach, determining breakpoints are placed at those *n-1* entrances to an *n*-way merged region from which code was deleted; their determining breakpoints are also invisible to the user. When control reaches a determining breakpoint, the debugger single-steps the program until the primary breakpoint is reached, at which time it is known that control flow came through that piece of deleted code. If the primary breakpoint is reached when not single-stepping, control flow must have passed through the path without the determining breakpoint. Since Teeple and Anderson's technique was never fully designed or implemented [11], its description is incomplete. Among other things, they failed to realize that merging less than an entire statement still requires path determination, and they did not discover that repeated cross-jumping can cause nested determiners.

The use of single-stepping as a path determination mechanism is not very satisfactory. It can be difficult to determine when normal execution can be resumed in cases in which control can pass either from the determining breakpoint into the merged region (and hence to the primary

breakpoint), or from the determining breakpoint to some other portion of the program. Single-stepping can also be very slow, particularly in cases where a cross-jumped region consists of a procedure call followed by a statement on which a breakpoint is to be placed. Finally, single-stepping from the determining breakpoint is only applicable to control-flow optimizations, whereas doing something at an invisible breakpoint and then continuing can be used to deal with other classes of optimization.

Hennessy [6] addresses the problem of transparently displaying values of variables in the presence of selected local and global code reordering optimizations. The local optimizations include common subexpression elimination, redundant store elimination, and code reordering. The global optimizations include code motion, induction variable elimination, and dead store elimination. With the assumption that the global flowgraph used during the optimization process is available to the debugger at runtime, Hennessy has developed algorithms that can usually detect when a variable has an incorrect value (in terms of the source program) and can sometimes correct this value. The algorithms were implemented and tested on a small group of programs to demonstrate their correctness and feasibility, but they have not yet been incorporated into any program development system.

The Firmware Development System (FDS) project at IBM Yorktown [14] planned to construct a programming environment with extensive optimization capabilities as well as an interactive high-level debugger that was aware of the optimizations. Unfortunately, the project lasted only long enough to produce a design document, so no information is available regarding its performance or utility [13]. The FDS system was intended to furnish two modes of compiler/debugger operation. In "full optimization" mode, the proposed debugging system would have tersely explained the ways in which optimizations had affected the program. For a sophisticated user, FDS operation in this mode could almost have been called correct. In "source-unchanged" mode, optimizations could involve only compiler-introduced temporaries, or could occur only within a statement. Optimizations would have thus been restricted in such a way that the mappings between the source text and the object code remained straightforward. Of course, recompilation would have been necessary to invoke "source-unchanged" mode if an error were encountered during a "full optimization" execution.

13. Conclusions and future work

To my knowledge, Navigator is the only working system for debugging optimized programs in a production environment. In most cases, Navigator can provide transparent debugging for control-flow optimized programs, in which the ordering of computations along any execution path is preserved. When transparent performance cannot be achieved, Navigator provides correct debugging. Navigator sacrifices transparent performance for runtime errors, interrupts, and

procedure calls inside merged regions in order to preserve the efficiency needed for routine use.

The invisible breakpoint technique can be used to augment Hennessy's algorithms for discovering the correct value of a variable. For example, an invisible breakpoint can store a final value for an eliminated store that reaches a breakpoint location, or it can save an old value of a variable when a breakpoint comes between a store moved earlier in the program and its original location. This use of invisible breakpoints to repair places where optimization removes information is currently under investigation.

Much work remains to be done in the area of debugging optimized programs. Code reordering optimizations create additional statement mapping problems, making it more difficult to avoid anomalous debugger behavior. In particular, expressing the new (optimized) relationship between a given computation and other computations is an open research area.

Acknowledgments

Sue Graham, John Hennessy, and Dan Swinehart participated in many long discussions about debugging optimized code. Ed Satterthwaite and Dick Sweet answered numerous questions regarding the internals of the Cedar compiler. Russ Atkinson was a valuable resource concerning the Cedar debugger. The counterexample to performing full path determination via the simple path determination algorithm, shown in Figure 5, resulted from an insight by Ron Goldman. Doug Brotz, Peter Kessler, Jock Mackinlay, Larry Stewart, and Dan Swinehart provided helpful comments on drafts of this paper.

References

1. Allen, F. E., and Cocke, J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, Rustin, R. (Ed.), Prentice-Hall, Englewood Cliffs, NJ, 1972, 1-30.

2. Aho, A. V., and Ullman, J. D. *Principles of Compiler Design*, Addison-Wesley, Reading, M, 1977.

3. Evans, T. G., and Darley, D. L. On-line debugging techniques: A survey. *AFIPS FJCC Proceedings Vol. 29*, 1966, 37-50.

4. Geschke, C. M. Global program optimizations. Ph.D. Dissertation, Computer Science Department, Carnegie-Mellon University, 1972.

5. Harrison, W. H. Position paper on optimizing compilers. *Proceedings of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, January 1981, 88-89.

6. Hennessy, J. L. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1982), 323-344.

7. Johnson, M. S. A software debugging glossary. *SIGPLAN Notices (ACM) 17*, 2 (February 1982), 53-70.

8. Loveman, D. B. Program improvement by source-to-source transformation. *Journal of the ACM 24*, 1 (January 1977), 121-145.

9. Mitchell, J. G., Maybury, W., and Sweet, R. Mesa language manual, version 5.0. Report CSL-79-3, Xerox PARC, Palo Alto, CA, April 1979.

10. Myers, B. A. Displaying data structures for interactive debugging. Report CSL-80-7, Xerox PARC, Palo Alto, CA, June 1980.

11. Teeple, D. W. L. Personal communication. November 26, 1981.

12. Teeple, D. W. L., and Anderson, J. C. The debugging of optimized code. Unpublished manuscript, MacDonald, Dettwiler & Associates Ltd., Richmond, B.C., Canada, March 1980.

13. Warren, H. S., Jr. Personal communication. March 3, 1982.

14. Warren, H. S., Jr., and Schlaeppi, H. P. Design of the FDS interactive debugging system. Report RC7214, IBM T. J. Watson Research Center, Yorktown, NY, June 1978.

15. Wulf, W., Johnsson, R. K., Weinstock, C. B., Hobbs, S. O., and Geschke, C. M. *The Design of an Optimizing Compiler.* Elsevier North-Holland, New York, 1975.

16. Zellweger, P. T. Interactions between high-level debugging and optimized code. Ph.D. Dissertation, Computer Science Division—EECS, University of California, Berkeley, to appear in 1983.

Erratum

There is a small flaw in the compiletime cross-jumping and path determination algorithms described in Section 8. Under rare circumstances, the resulting set of path determiners will be unable to distinguish between two regions with different preceding control-flow that have been cross-jumped. The flaw compromises only transparent behavior, not correct behavior; programs that demonstrate the flaw are oddly structured and are unlikely to occur in practice.

In order to provide complete and correct path determination, the cross-jumping algorithm must mesh properly with the path determination algorithm. In this case, small changes to one or both algorithms can eliminate the problem. The final object code for programs need not be changed.

The problematical feature in the cross-jumping algorithm is that either merging code sequence is allowed to contain embedded labels, provided that all jumps to labels in the toDelete sequence are redirected to corresponding locations in the toRemain sequence. Allowing embedded labels is desirable because it permits longer code sequences to be merged in a single application of cross-jumping, thereby reducing the number of path determiners.

The unnecessary ambiguities can arise in two different ways: either from a single application of cross-jumping or from repeated applications. In the first kind, cross-jumping merges two code sequences whose object code is identical, but whose label structure (and thus whose control flow) is not. The two sequences have a common entrance that precedes two *non-identical* instructions, causing the common entrance to be marked with determiners for both sequences. Therefore, the two original sequences are undistinguishable whenever the merged region is entered through that entrance. If more determiners were inserted, the differing control flow could be used to avoid the ambiguity. In the second kind, a merging sequence contains two or more subsequences that could be merged via cross-jumping, but that have not yet been. In the current path determination algorithm, a sourcelist entry is always distinguished by the determiner for the *first* sequence it was a part of. Subsequent repeated cross-jumping within the sequence creates two or more source alternatives that are marked with the same determiner, and that therefore cannot be told apart.

A safe solution is to disallow embedded labels in sequences that are candidates for cross-jumping. For complicated cases, this alternative requires more cross-jumping iterations, resulting in more path determiners. The additional path determiners use more table space, and cause more determining breakpoints for a given primary breakpoint. The determiner minimization phase proposed in Section 8.6 would remove excess determiners, but compilation time would increase.

Two other solutions are currently being investigated. The first would allow embedded labels, but avoid the special cases in which they create ambiguity. The second would alter the path determination algorithm: for example, a sourcelist entry could contain a list of determiners rather than containing a single determiner and possibly some enclosing nested determiners.

Further details can be found in [16].