LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-474

DRIBBLE-BACK REGISTERS: A TECHNIQUE FOR LATENCY TOLERANCE IN MULTIPROCESSORS

Vijayaraghavan Soundararajan

October 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Dribble-Back Registers: A Technique for Latency Tolerance in Multiprocessors

by

Vijayaraghavan Soundararajan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1992

Signature of Author .....
Department of Electrical Engineering and Computer Science
May 14, 1992

Certified by .....
Anant Agarwal
Associate Professor of Computer Science and Electrical Engineering
Thesis Supervisor

Accepted by .....
Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Students

Dribble-Back Registers: A Technique for Latency Tolerance in Multiprocessors

by

Vijayaraghavan Soundararajan

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 1992, in partial fulfillment of the
requirements for the degree of
Bachelor of Science

Abstract

As parallel machines grow in scale and complexity, latency tolerance of synchronization faults and remote memory accesses becomes increasingly important. One method for tolerating this latency is by multithreading the processor and rapidly context switching between these threads. Fast context switching is most effective when the latencies being tolerated are short compared to the total run lengths of all the resident threads. If this condition is not met, it may become necessary to expend processor cycles to unload a blocked thread and load in a new one. This thesis presents the dribble-in, dribble-out register file, which facilitates fast context switching and the ability to hide the latency of loading and unloading context state. Through an analytical model and a simulation framework, we show that the dribble-in, dribble-out register file compares favorably against existing designs.

Keywords: register file, RISC, latency tolerance, multithreaded multiprocessor, context switching.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Electrical Engineering

Acknowledgments

I would like to thank everyone in the Alewife group for all of the help and good-natured advice they provided me over the course of this thesis. In particular, I would like to thank Beng-Hong Lim and David Chaiken for being patient, understanding, and friendly officemates, Donald Yeung for guiding me and introducing me to dribble-back registers, Kenmac for his advice and his ear-splitting music, Kranz for his critique during my oral presentation, Kubi for reminding me how much less fun I could actually be having, Dan N. for providing material for jokes, Jory for letting me thrash his machine with simulations, Gino for reminding me how my lines from Hamlet are universally relevant, Kirk for his awesome computing knowledge, Jonathan Babb for his back-of-the-napkin computations at 4 am, and Marc Schaub for some helpful thesis hints. I would especially like to thank my advisor, Professor Anant Agarwal, for teaching me more in 6 months of thesis work than I learned in 4 years of MIT.

I would like to thank Gary Muntz for showing me that people in the real world work on Friday nights also. I would like to thank Coca-Cola and Pepsi-Cola for providing me with artificial sleep in the form of caffeine. I would like to thank all of my friends for putting up with my complaining over the course of the term: without their support I would never have finished. I would especially like to thank Chee-Heng Lee and Donald Tanguay for keeping me company during the all-nighters we pulled together, and for providing me with late night car rides to and from Tech Square.

Finally, my gratitude goes out to my brother, whose accomplishments gave me incentive to work hard, to my sisters, for comic relief, and to my parents, for everything else.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As multiprocessors grow in size and in complexity, latency tolerance becomes an increasingly important issue. Sub-optimal partitioning of program data on a network of processors increases communication needs and puts heavy reliance on the bandwidth capabilities and switching speeds of the network for performance. Increasing the frequency and latency of remote data accesses causes processors to waste cycles waiting for data. Additionally, waiting for synchronization conditions to be met also forces idle processor cycles if the processor is unable to perform any other useful work, as figure 1-1 illustrates. The goal of latency tolerance is to overlap processor or wait

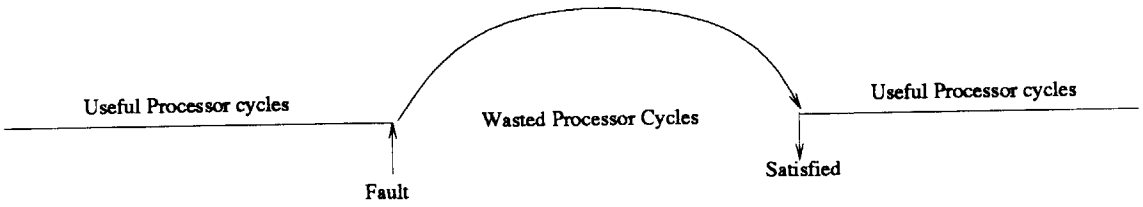

Figure 1-1: Processor idling in a single-thread system

cycles with useful work[7]. This is critical to the construction of large-scale parallel processors because access latencies get larger when the physical size of the machine increases even if for no other reason[2].

The main goal of multithreading is latency tolerance: overlapping communication costs with useful computation in order to minimize idle processor cycles[1, 2]. Multithreaded multiprocessors use parallelism to hide latency by concurrent support

of multiple threads of computation. These threads may be several programs or just different partitions of the same program running in parallel, although the processor only operates on one thread at a time, taking data and making references within that thread alone[2]. By maintaining several processes in parallel and making process switch overhead small, multithreading allows the processor to perform useful work even during remote memory accesses or on synchronization faults, rather than simply wasting cycles waiting for the requests to be satisfied.

1.1 Fine Multithreading vs. Block Multithreading

There are 2 flavors of multithreading: fine multithreading and block multithreading. Fine multithreading implies frequent context switching, and block multithreading implies infrequent switching. The HEP[5] is an example of a multiprocessor that uses fine multithreading. In the HEP context switching occurs on each instruction. There is a fixed number of processor resident threads that can be run concurrently (8 in the HEP). The processor switches threads on each cycle, switching to each context in turn regardless of the actually number of processes running; thus, the switching mechanism must be fast so that more time is spent executing than switching. If there are fewer processes running than hardware contexts available, than there are many wasted cycles since there will be unnecessary switching to unused contexts. As a result single-thread performance is poor. Multiple thread performance is maximized if the number of threads matches the number of hardware contexts.

Another form of fine multithreading is one in which context switching occurs on each memory request. Since there can be short execution runs between context switching (depending on the frequency of memory requests), the switching mechanism must again be very fast. In order to switch rapidly there must be minimal processor state to be saved upon switching. This form of fine multithreading also suffers from poor single-thread performance since that thread pays the context switch overhead on each memory request even though there are no other threads. There must also be

large amounts of network bandwidth to satisfy all of the memory requests, and lots of threads if, as in the HEP, switching occurs among a fixed set of hardware contexts (regardless of the number of threads running).

Block multithreading implies less frequent context switching as compared to fine multithreading. Context switching occurs on cache misses or on synchronization faults in order to tolerate the latencies of each. There are long runs between switches because caches are employed to minimize misses (so that we need to worry more about synchronization faults, which generally occur less frequently than misses). There are thus fewer requests in the network and bandwidth need not be increased in large amounts to sustain multiple threads. This type of multithreading provides good single-thread performance as well as an effective means tolerating latency. Figure 1-2 shows some of the differences between fine multithreading and block multithreading.

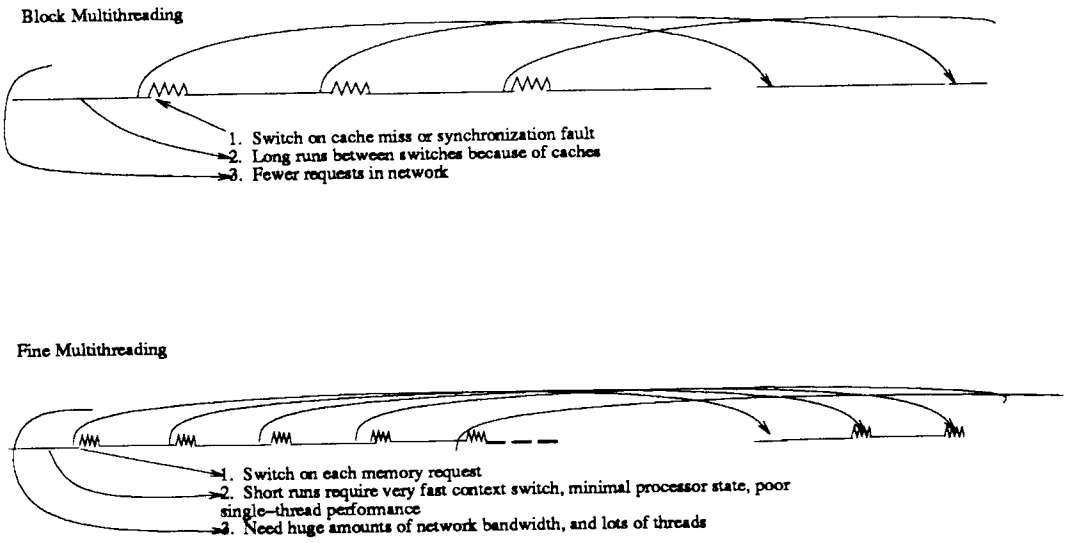

Figure 1-2: Block Multithreading vs. Fine Multithreading

1.2 Definition of Terms

Before moving on, in order to avoid confusion between context switching and context loading/unloading, we should clarify what we mean by context switching. We define

a context switch as the transfer of processor control from a processor-resident thread to another processor-resident thread in a multithreaded processor. No thread state needs to be saved into memory, because the thread state remains in the register file, and is not flushed; in fact, for a context switch, all that is necessary is to bump a context pointer to point to the new set of registers. We define loading a thread as the action of installing the state of a thread into a hardware context on a processor, and unloading a thread as the complementary action of saving the processor-resident state of a thread into memory. The processor's register file contains the hardware contexts, and memory is the storage place for the unloaded threads. Figure 1-3 displays the difference between loaded and unloaded threads. Context switching can occur on

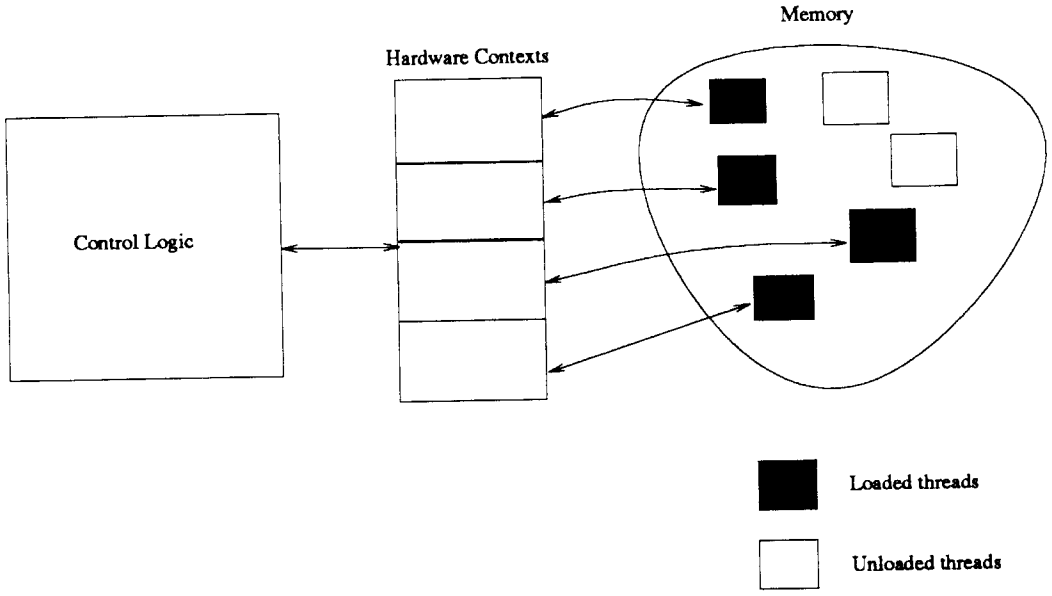

Figure 1-3: Processor, Register File, and Memory Architecture

cache misses if the latency in retrieving data is larger than the latency of switching threads, and it can occur on synchronization faults if the time for the synchronization condition to be fulfilled is larger than the switching latency[1, 2]. Figure 1-4 shows the difference between context switching and context loading.

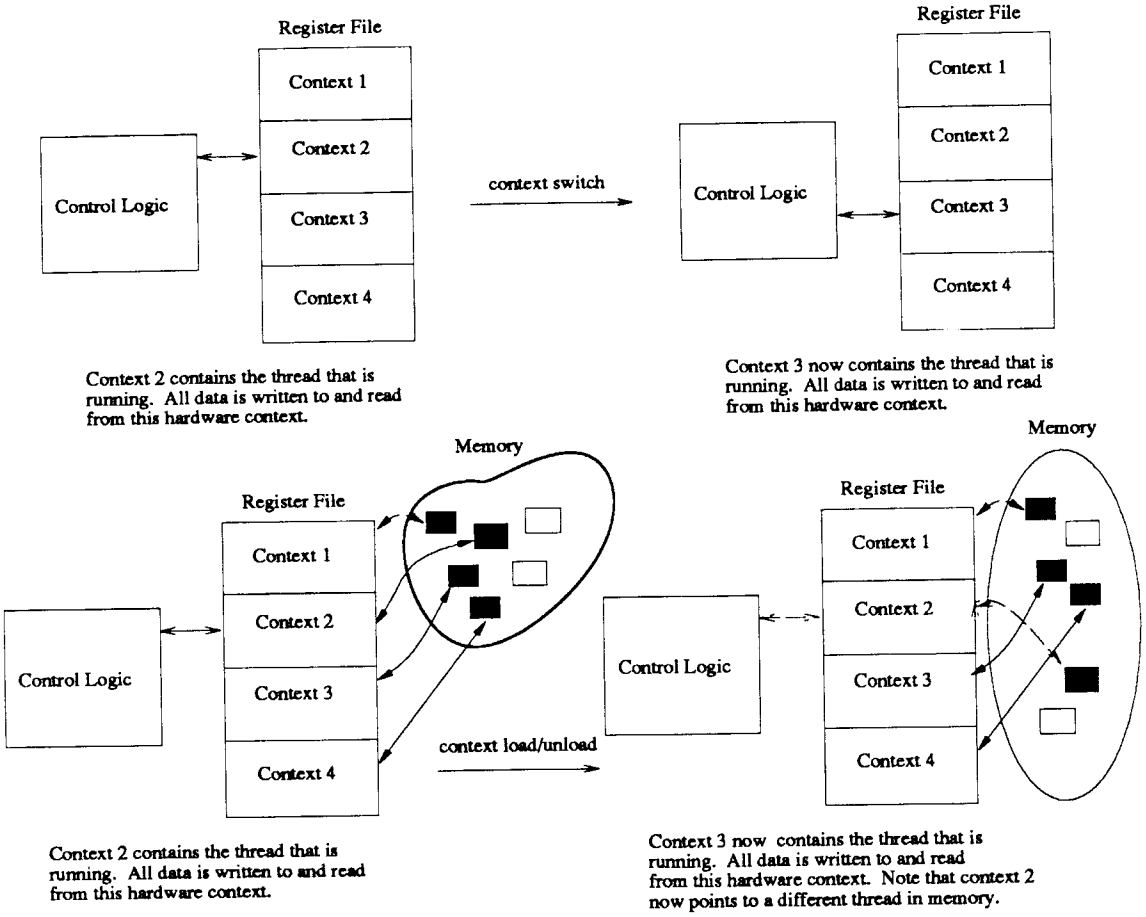

Figure 1-4: Context Switching vs. Context Loading

1.3 Contributions of this Thesis

In any type of multithreading, high-speed context switching is crucial. Having no contexts ready when a context switch is necessary or having too much state to save/restore upon a context switch both slow down the context switch time. In this thesis I investigate the register file design for fast context switching. I present the dribble-in, dribble-out (DIDO) register file (also known as dribble-back registers[9]), which is capable of reducing processor idle cycles by overlapping useful processor cycles with the loading of runnable threads and the unloading of threads stalled by synchronization faults. The DIDO register file is intended to tolerate latencies primarily due to synchronization faults. More precisely, DIDO reduces wasted processor cycles associated with loading and unloading a context state, which occurs only during synchronization faults. Because cache miss latencies are much smaller (in this paper we assume cache miss latencies on the order of the run length), on cache misses a context switch is performed rather than a load/unload operation. Even with frequent synchronizations the DIDO register file provides improved performance over typical register file designs.

I also present an analytical model for understanding the behavior and performance of this mechanism, as well as a preliminary VLSI implementation to understand the required data paths and control mechanisms. I also describe a simulation system for dribbling register files and corresponding simulations which compare the performance of the dribbler with other designs and serve to validate the model.

1.4 Organization of this Thesis

The rest of this thesis is organized as follows. Chapter 2 reviews existing register file designs, and introduces some aspects of the DIDO scheme. Chapter 3 presents a more detailed description of the architecture of the DIDO register file. Chapter 4 describes an analytical model for the DIDO. Chapter 5 describes the simulation methodology used to evaluate DIDO against existing register file designs. Chapter 6 describes the

results of the analysis, and presents conclusions and directions for future work.

Chapter 2

Background

Before describing the dribble-in, dribble-out register file, it is worthwhile to review other kinds of register file designs. These include

1. Single Register Set: only one context is stored in the register file at any given time
2. Multiple Register Set: multiple hardware contexts enabling support of many threads concurrently
3. Context Cache: register state is loaded and unloaded at the granularity of a single register

2.1 Single Register Set

In a single register set design, the register file holds one hardware context. A switch of control from one thread of computation to another (due to synchronization, cache miss, or completion of thread) requires the unloading of the controlling thread–called the resident thread because it resides in the register file at the time of the switch–and the loading of the new thread. This loading and unloading requires the transfer to and from main memory of the values of each register. When threads have long run lengths (i.e., run for long periods of time without remote memory accesses or synchronization faults) this approach is effective because loading/unloading is rare

enough that the overhead they incur is insignificant. However, if run lengths are short, the lack of latency tolerance hurts performance: for example, if the fault is a cache miss, loading/unloading is usually more time consuming than waiting out the fault. When waiting out the remote access is the only recourse that the processor has, it must endure idle cycles until the data is fetched. Figure 2-1 depicts context switching in a single register set design, assuming runnable threads exist in memory.

2.2 Multiple Register Sets

Multiple hardware contexts facilitates one method of tolerating fault latencies, namely rapid context switching. With fast context switching these multiple register sets can mask memory latencies and overlap communication costs with useful work. If remote memory access latencies are large then processor utilization can increase dramatically by as much as 40% over performance with a single context[2]. With cache miss access latencies hidden, hiding synchronization latencies becomes the major issue. In this thesis we consider latencies due to synchronization faults, and define run length as the time between when a thread begins execution and when it faults due to a synchronization.

When synchronization latencies are large relative to run lengths so that a fault is not satisfied even after all other resident threads have run, it is advantageous to unload the stalled context and replace it with a runnable thread. Figure 2-2 illustrates the situation. Notice that from point A to point B, no resident contexts are ready. However, by loading in new threads, 2 more run lengths are squeezed into this otherwise wasted time. Multiple register set protocols generally dictate running threads and context switching on large-latency faults until all are resident threads are stalled, at which time one thread is unloaded and a new one loaded in its place (figure 2-2 assumes this protocol). If run lengths are long, the time before the newly-loaded thread encounters a synchronization fault may be sufficient to allow the waiting threads' wait conditions to be satisfied. Context switching will be possible

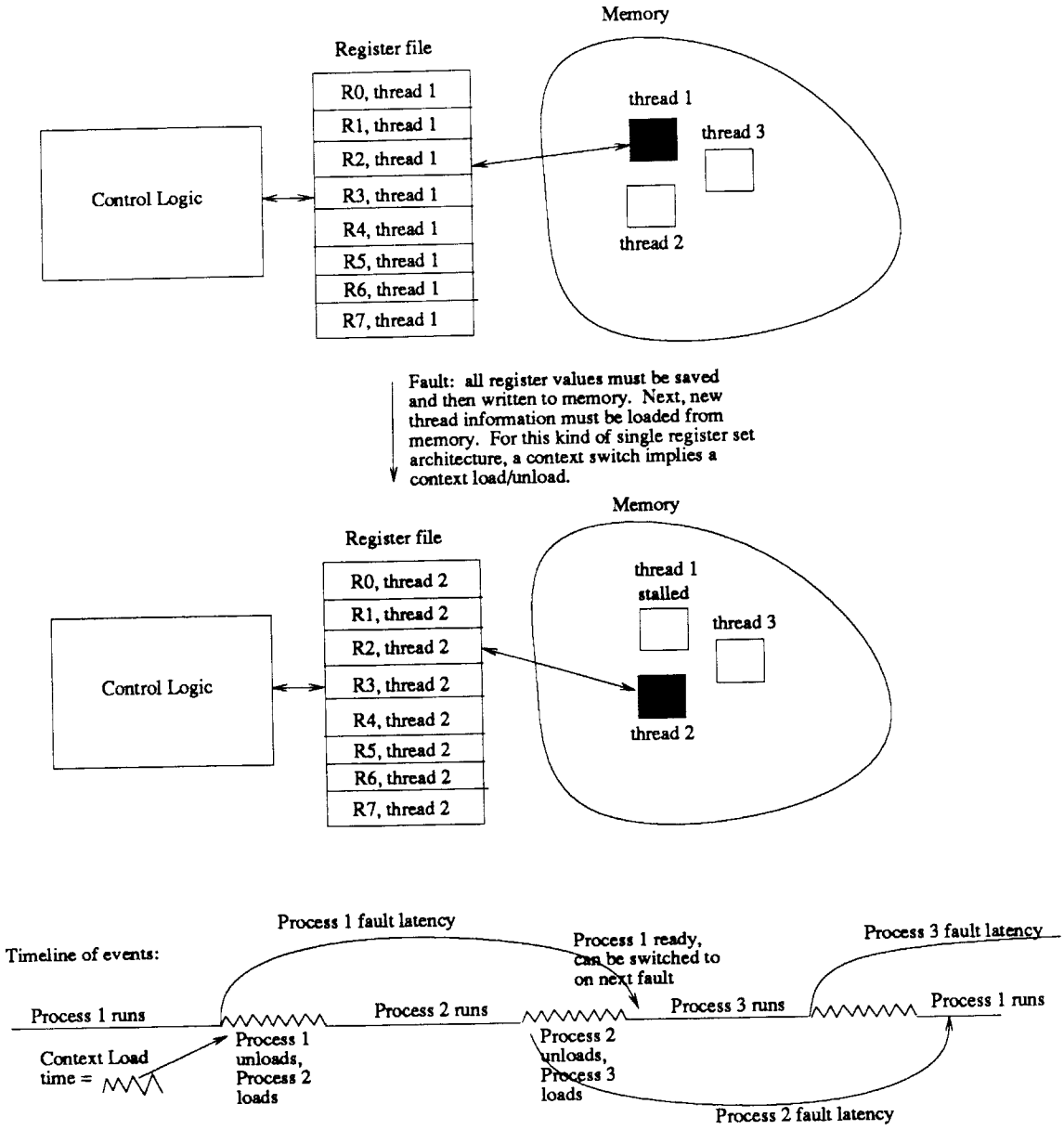

Figure 2-1: Context switching in a single register set system.

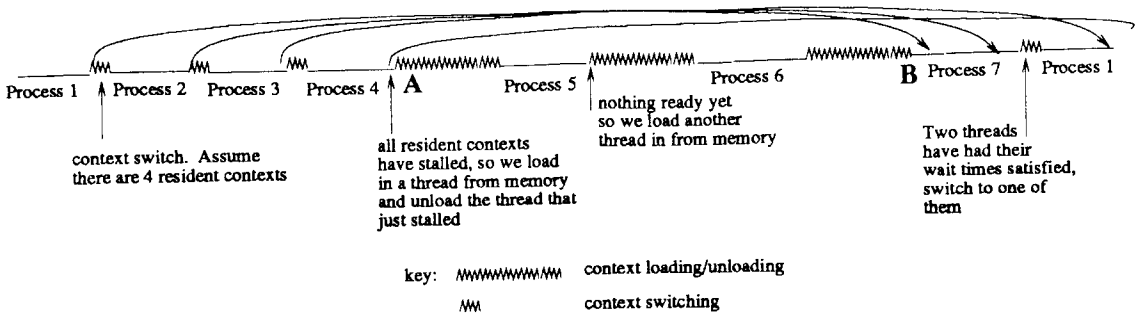

Figure 2-2: Multiple contexts alleviating long fault latencies by unloading stalled threads and loading runnable threads.

in this case and the latency of the new fault will be tolerated. However, if the run lengths are short, on the order of the load/unload time, then when all context but one stall, this "load upon demand" involves a computation/load sequence, where as much time is spent loading as is spent computing. (See figure 2-3: notice that from point A to B, no resident contexts are ready. Because the wait time is so large, it takes several context load/unloads before the resident contexts are available, and as much time seems to be spent loading as is spent computing.) Were there a large number of hardware contexts, then rarely would all of them stall and utilization would be very high. However, the number of hardware contexts must remain small to minimize complexity of the register file and supporting hardware, and the optimal arrangement would include the smallest number of contexts resident that would provide highest gain.

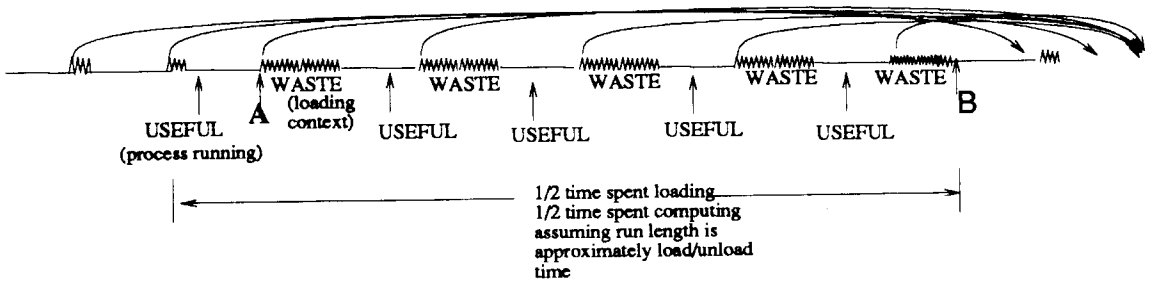

Figure 2-3: Loading/computation interleaving in multiple register set designs.

2.3 Context Cache

One drawback to each of the above schemes is that the loading cost can become a burdensome one if the run lengths are very small. One way to reduce the loading time is to decrease the amount loaded when loading a context. For example, if on a context switch only certain registers were used for the duration of that run length, it would be more efficient to load in only those registers. Not only would this save time in the loading and activation of a context, but if many contexts operated on few sets of registers, then many more contexts would fit into the register file.

The Context Cache[3, 11] takes this "load only what is needed" approach. In this design, the register file does not store a complete context, but simply the registers involved in the computations that occur or have occurred. Registers in a context that are rarely used are rarely loaded unless needed. One distinct advantage of this scheme is that it ensures that no excess register loads occur: only registers that are needed are loaded into the file. A context load/unload involves little latency and does not tie up the processor for as many cycles as the other schemes do. If a process uses only a few registers for all of its computations, then very little loading must be done and the bulk of the processor time is dedicated to computation. The disadvantage of this approach, however, is that if a program exploits all of the registers in the context, then there is little gain since all of the registers will be loaded eventually, as in the other designs. In addition, the fully-associative nature of the register file makes its decode logic more complex than any of other register file decoders. This added VLSI slows down the driving of word lines and subsequently reads and writes.

Chapter 3

The Dribble-in, Dribble-out Register File

This chapter presents the dribble-in, dribble-out register file. First, the architectural differences between DIDO and normal multiple register set designs are discussed. Following that is a discussion of the control system of DIDO.

3.1 Motivation

With the advent of on-chip instruction caches, the bus to off-chip memory need not be a bottleneck for data and instructions. If data accesses occur mostly within the register file (as opposed to accessing data in off-chip memory), the bus to off-chip memory can go unused for potentially long periods of time. In particular, if the compiler stores the most frequently used variables in registers rather than in main memory, fewer operations to main memory will be required. Load/store operations in RISC processors are the only instructions that explicitly require main memory (rather than register file) access. Although context loads/unloads also use this bus, there are still long periods in which the cache and data pathway goes unused: for example, studies have shown 30% of instructions in RISC processors are load/stores, leaving the cache/memory bus free 70% of the time[6].

Using these free cycles to the data cache and memory subsystem effectively can sig-

nificantly increase performance and decrease idle cycles. My proposal, the dribble-in, dribble-out register file, inspired by Sites' dribble-back registers for multiple register window architectures[9], is a variant of the multiple register set design described in Chapter 2 involving the addition of two extra ports to the register file in order to use the free cycles to the data cache.

3.2 DIDO Data Paths

A typical register file has three ports: two for reads and one for writes. Context loads/unloads consume all of the processor's time for their respective durations. On a context load/unload, these ports are used to load new context information and unload the old context information. Figure 3-1 displays the typical data paths for an architecture containing a three-ported register file. In contrast, the dribbler is able to parallelize instruction execution and context loading/unloading because it contains two separate dribble ports in addition to the typical register file ports. These two ports are dedicated exclusively to dribbling, allowing the processor to do useful work in parallel. With the simple addition of these two ports and the addition of minor control logic, processor utilization can increase dramatically. Figure 3-2 illustrates the additions required to implement dribbling. The dribble control logic block determines where in main memory the context information that is to be loaded/unloaded goes, and sequences through the register file unloading/loading as desired.

The dribble ports communicate to the cache/memory bus. A block diagram is shown in figure 3-3. Notice how the dribbler allows simultaneous load/unloading of context information and instruction execution. Any instruction that does not require an access to main memory (most instructions fetch operands exclusively from the register file and are classified in this category) is one in which the memory bus is free for dribbling. We assume an on-chip instruction cache for instruction fetching, so that we need only worry about data accesses to main memory. Only load/store instructions (i.e. instructions that access main memory rather than the register file for data) prohibit dribbling on a given cycle. This number of free cycles is critical

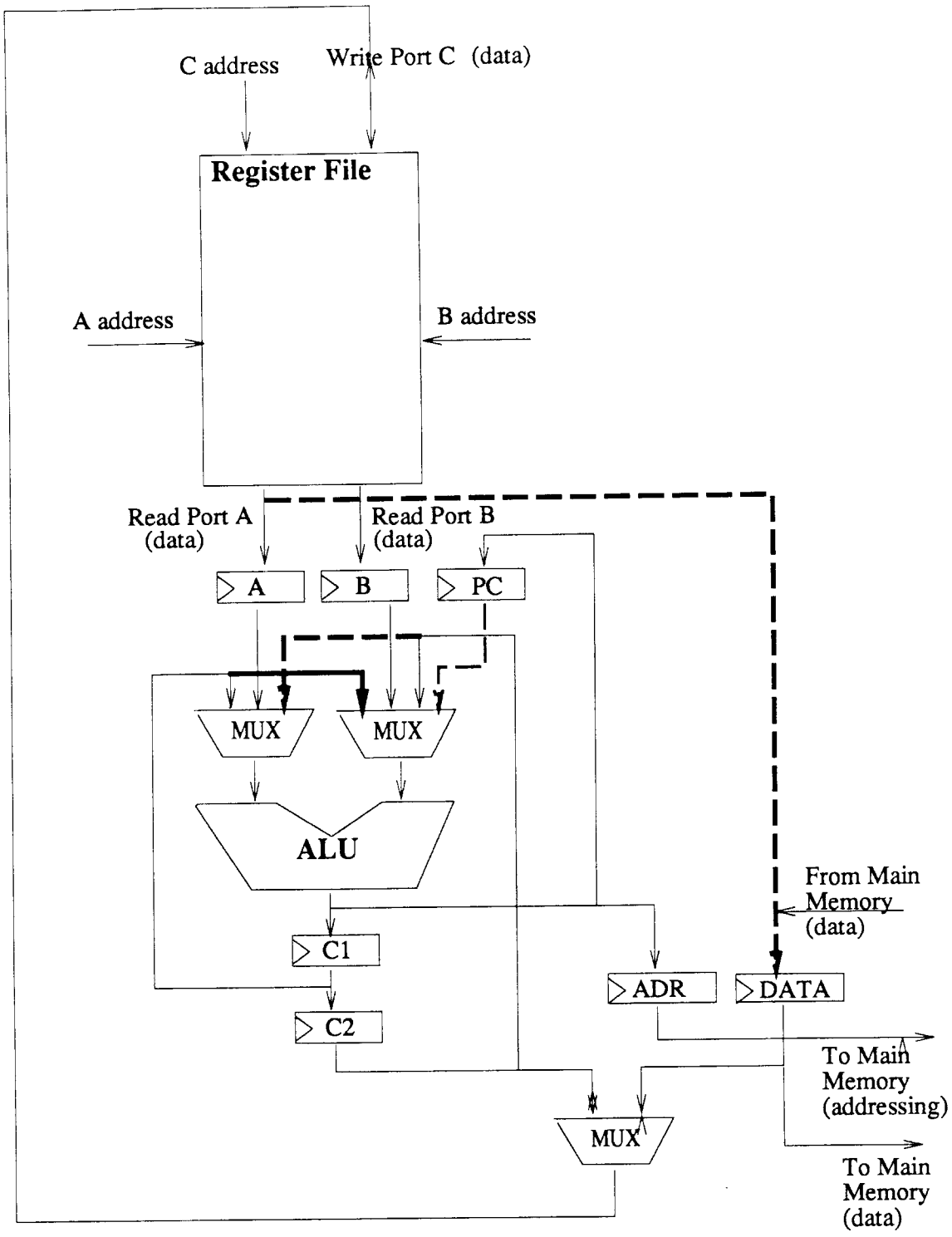

Figure 3-1: A typical three-ported register file RISC architecture.

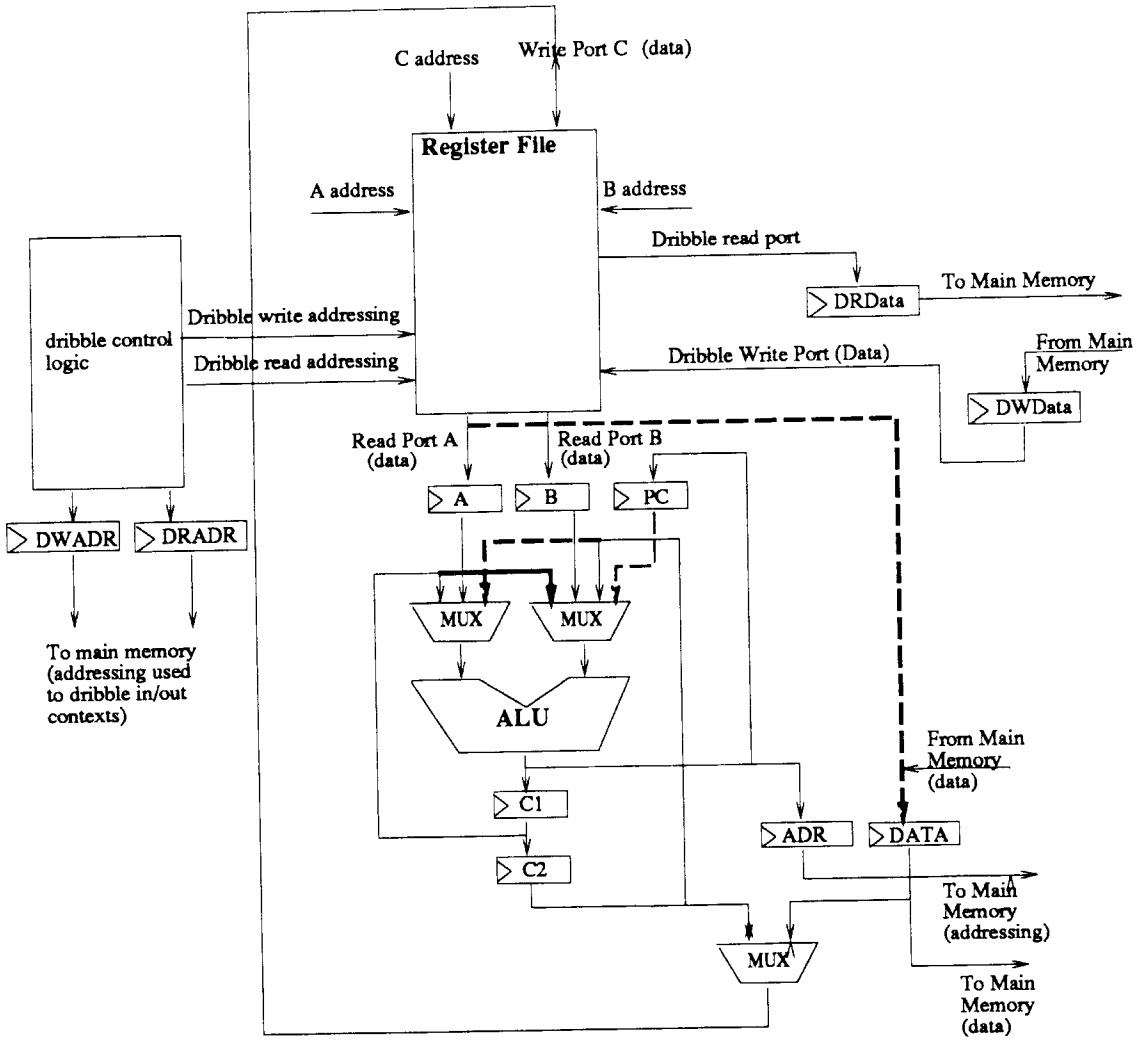

Figure 3-2: A Possible DIDO Data Path.

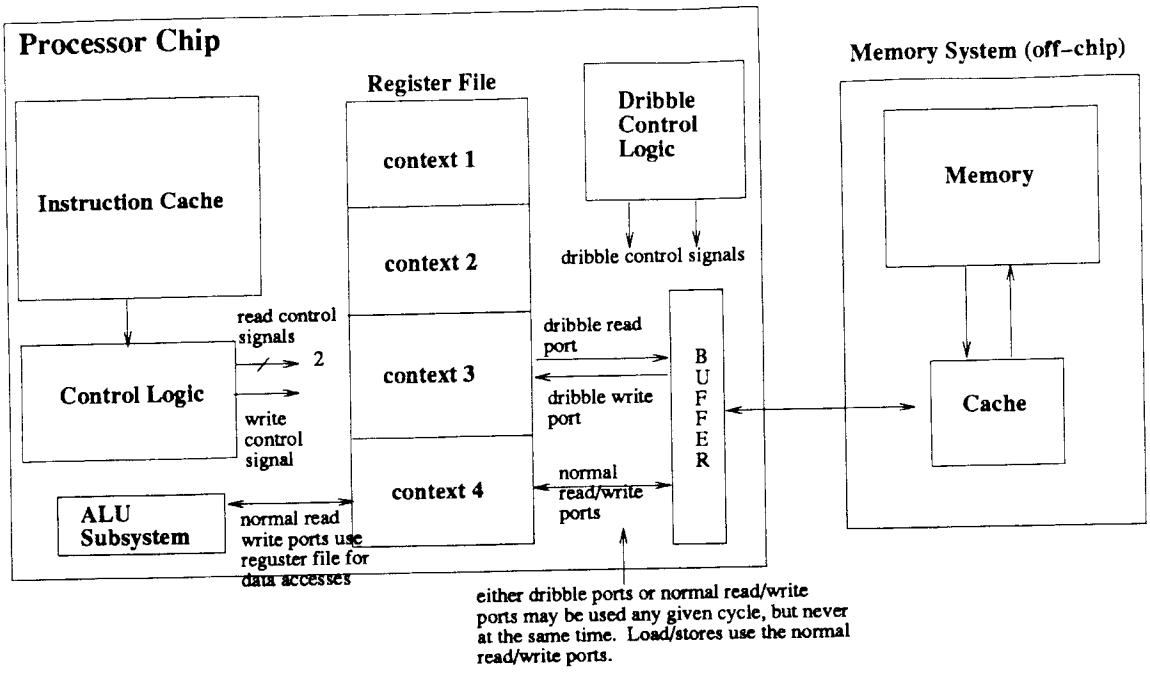

Figure 3-3: Dribbler block diagram.

to the operation of the dribbler: if too many instructions prohibit dribbling, then the dribbles rarely complete before the run lengths expire, and the gain the dribbler affords is minimal. If somehow all resident threads fault before the dribbler has replenished one of the contexts, then no useful work can occur until the dribbler has finished the context load. In the multiple register set case with short run lengths this case is common, and when this happens the full load cost is incurred. With the dribbler, however, only the amount of the dribble that could not be finished during the run lengths must be awaited before processor operation can resume, as figure 3-4 illustrates. In figure 3-4, the DIDO's amortization over a run length causes the latency of a forced load (i.e., the load that occurs after process 2 stalls, because no other resident threads are runnable) to be smaller than the latency in the multiple register set case. One assumption we make is that there exists enough parallelism so that there are always threads to run, because we wish to measure the ability of the dribbler at keeping the register file full of work to do. When task partitioning divides work into short segments with frequent synchronization, then it is valid to assume that there are many of these small threads, since many small ones would be required to

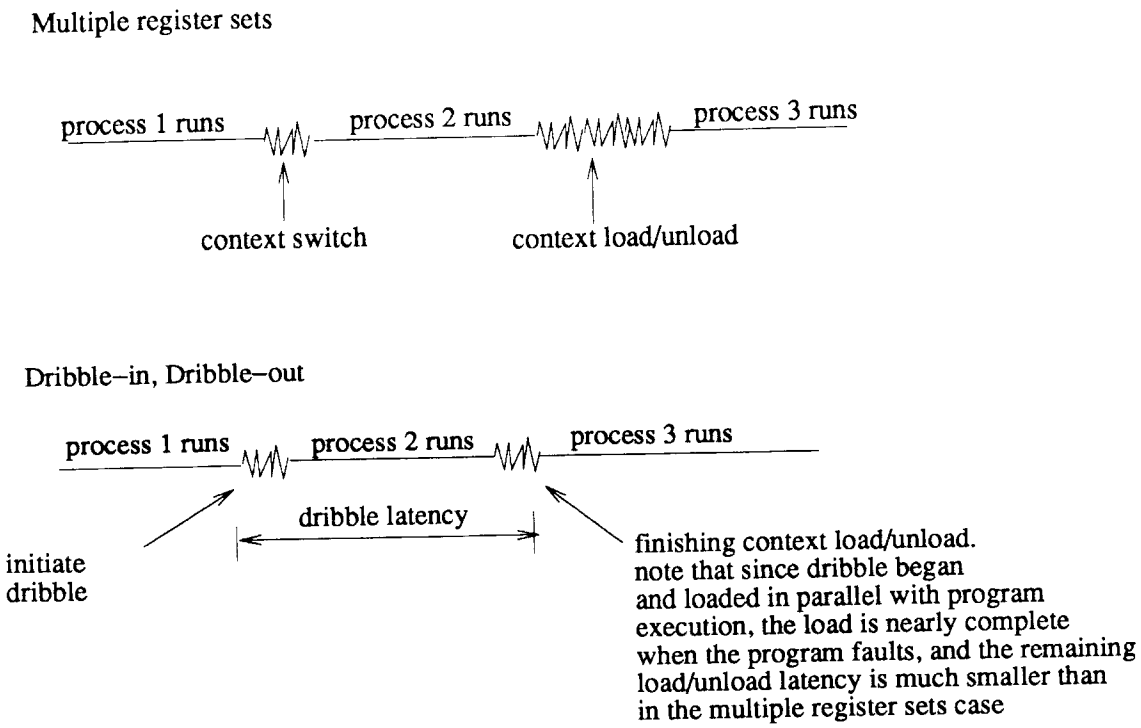

Figure 3-4: Loading/unloading latency differences, multiple register sets vs. DIDO

finish the task (as opposed to a coarse-grained partitioning into a few large threads). In addition, we assume that wait times for the satisfaction of synchronization faults are long, because short run lengths and long wait times are often difficult to resolve in multiprocessors: we wish to show that the dribbler can perform well under these conditions.

3.3 Control system

A finite state machine bubble diagram of the dribbling mechanism is shown in figure 3-5. The key to its effectiveness is its polling behavior. When the first synchronization fault occurs, the dribbler begins its operation, initiating the unloading of that context and the loading of another runnable thread from memory. All of this loading/unloading occurs as instructions are executed out of another context. From this time on, whenever the dribbler finishes the loading of a context, it polls all of the resident contexts to see if any are stalled. If any are stalled, and if there are

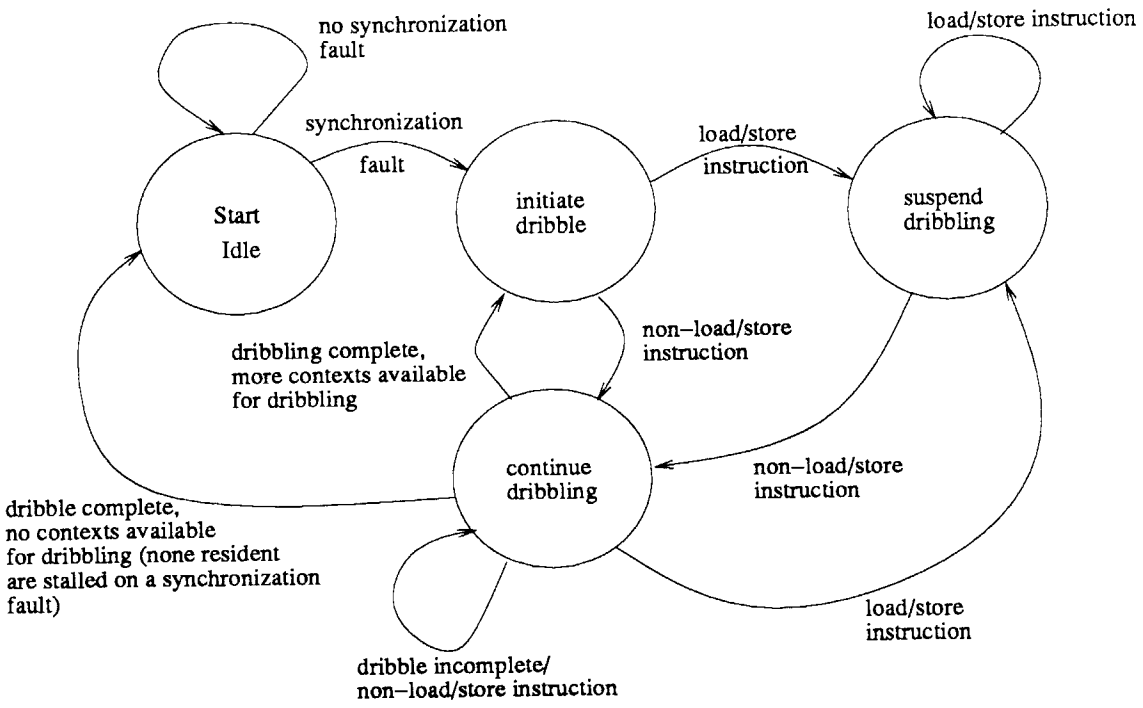

Figure 3-5: Dribbler High-Level Control FSM

any runnable threads that can be loaded in, another dribble is initiated. If all contexts in memory are stalled, then no instructions can be executed until the dribbling is complete, unless a context's synchronization wait is satisfied during the dribble. In this case a context switch to the ready context occurs and the dribble continues transparent to the processor. Note that the dribbler will always be operating if there are any stalled contexts, so that the wait for a dribble to finish (i.e., when all processor resident threads have faulted) will never be as large as the wait for a dedicated load/unload. See figure 3-6: notice that the dribbler's polling behavior requires it to dribble if a stalled contexts is resident and a runnable thread exists in memory.

On cache misses the dribbler attempts to context switch. If a context switch is not possible, the processor waits out the latency of the miss. This protocol is used because cache miss latencies are much smaller than synchronization fault latencies (for example, a typical remote memory access in a 2-D mesh interconnection may require 40 cycles, while a synchronization fault may require 1000 cycles to be satisfied). We would not choose to unload a thread faulted on a cache miss because the act of

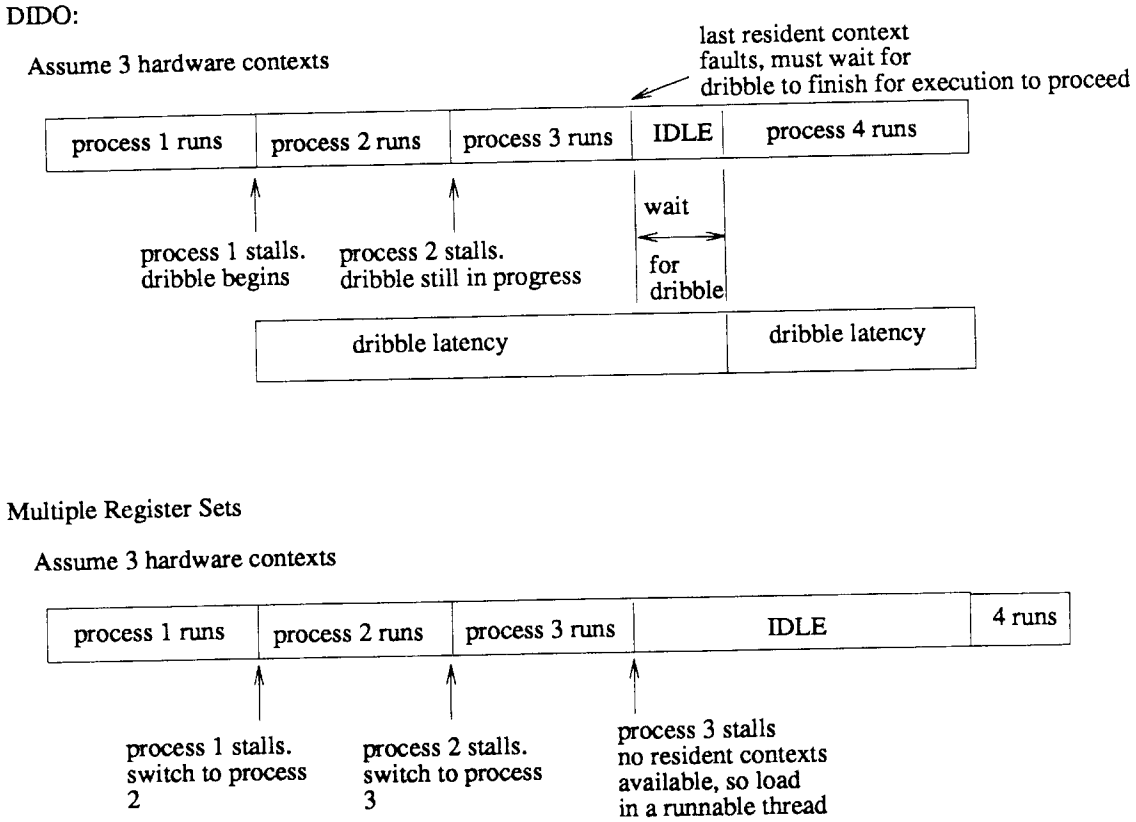

Figure 3-6: Comparing latency tolerance in dribbler vs. multiple register sets

unloading the thread is potentially more costly than waiting out the thread (this is rarely true for a synchronization fault, as loading is generally on the order of 100 cycles[8]), and usually switching to and running other processes provides enough time for the remote request to be serviced and for that context to become runnable again.

The most obvious gain of this polling is that we reduce the cost of waiting for a context to load in the rare case that everything has stalled. Second, by keeping the register file full of ready contexts, we increase the chance that on a cache miss there will be a context to switch to and the latency of the cache miss will be tolerated (i.e., the cache miss will be unnoticeable since we will be running useful work while waiting for the miss to be satisfied). Third, we increase the chance of a ready context to switch to on a synchronization fault. In fact, we can think of DIDO as a software managed cache of process contexts in which dribbling is prefetching on free cycles in order to avoid load/unload latencies. Providing us with these methods of latency tolerance the dribbler seems a promising idea. Our simulations will show that it does indeed increase processor utilization.

Chapter 4

Mathematical Analysis of DIDO

To gain insight into factors affecting the performance of a dribble-in, dribble-out register file, this chapter proposes a simple model for the utilization of a processor employing a DIDO register file. A first-order approximation is presented in order to obtain intuition concerning performance limits. We will then see how well this model correlates to the results of our experimental data.

4.1 Queueing Model

We present the model in stages: first, we assume no load/store operations occur and derive an expression for processor utilization in that case. Next, we approximate the effects of load/store operations and derive an expression for processor utilization in that case.

4.1.1 No load/store instructions approximation

The dribbler continually polls the resident contexts for any that are stalled due to synchronization faults. Whenever it detects that a context is stalled, it looks for any non-resident contexts that are runnable. If there is a runnable thread that is not resident, a dribbling process is initiated: the stalled thread is unloaded, and the runnable thread is loaded into the register file via the dribble ports as execution in the register continues uninterrupted (except for the context switch overhead).

As contexts stall, they are collected one at a time by the dribbler. The dribbler will only remove contexts stalled by synchronization points. To approximate the processor utilization in the absence of load/store operations, we can model the register file as a queueing server in which register frames (register frame is synonymous with thread) are added to the register file at some rate by a dribbler, and register frames are consumed at some other rate by an execution process that causes synchronization faults. Figure 4-1 displays a graphical representation of such a queue. A register

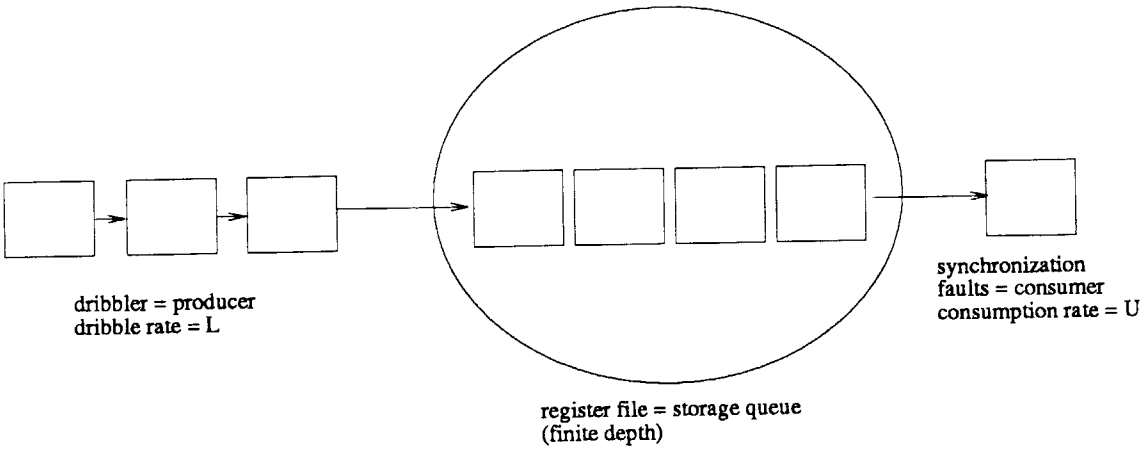

Figure 4-1: DIDO as a queueing server.

frame is added into the queueing server the moment the dribble of a frame completes. Similarly, a register frame is considered to be consumed the moment a synchronization fault occurs. The utilization of the processor is the fraction of the time the register file is busy, and is simply the utilization of the queueing server.

Let the service rate or consumption rate be U. That is, 1/U is the average time between synchronization faults, i.e., the average run length. In other words, U is simply the probability of a synchronization fault on a given useful cycle. (We say useful cycle because we don't have synchronization faults when, for example, servicing a cache miss.)

Let the arrival rate be L. That is, 1/L is the time between completions of dribble operations. In other words, L is the probability that a dribble completes on a given cycle, and a process is added.

The utilization, ρ, of the queueing server is given by

$$\rho = \frac{L}{U}. \tag{4.1}$$

Clearly, ρ has a maximum of 1. That is, if the rate of synchronization faults is less than the rate of dribbling, then in the steady state, there will be no idle time. Of course, this formula only applies when $L \leq U$, as in queueing systems.

As an example, suppose $U = 0.01$ and $L = 0.002$. That is, on average there are synchronization faults every 100 cycles, and there are 500 cycles between dribbles. Then processor utilization is

$$\rho = \frac{L}{U} = \frac{0.002}{0.01} = 0.2. \tag{4.2}$$

We ignored load/store operations in the above analysis because on load/store operations dribbles cannot occur since the data bus is used for data acquisition rather than dribbling.

This rudimentary model assumes that the queue length can become infinitely long. This is tantamount to having a huge number of register frames. To model a finite number of register frames we would have to use much more complex queueing formulae for bounded length queues. We have developed such a model, but in practice 3-4 register frames yields close to the performance of infinite register frames, so this approximation is valid.

This model also assumes that the dribbler is continually running. If the run length is on average shorter than the dribble length, then the dribbler will rarely complete a dribble before a synchronization fault occurs, so that the dribbler will nearly always have register frames to remove and will be running nearly continuously, as figure 4-2 suggests. Notice the dribbler always has contexts to remove (but won't do so unless there are runnable threads to replace them with). The arrows pointing down indicate which synchronization point prompts which dribble, and the arrows pointing upward indicate the dribbles which provide the appropriate context. However, if the run length is on average longer than the dribble duration, then when the dribble completes there will not necessarily be a stalled context to remove, so that the dribbler

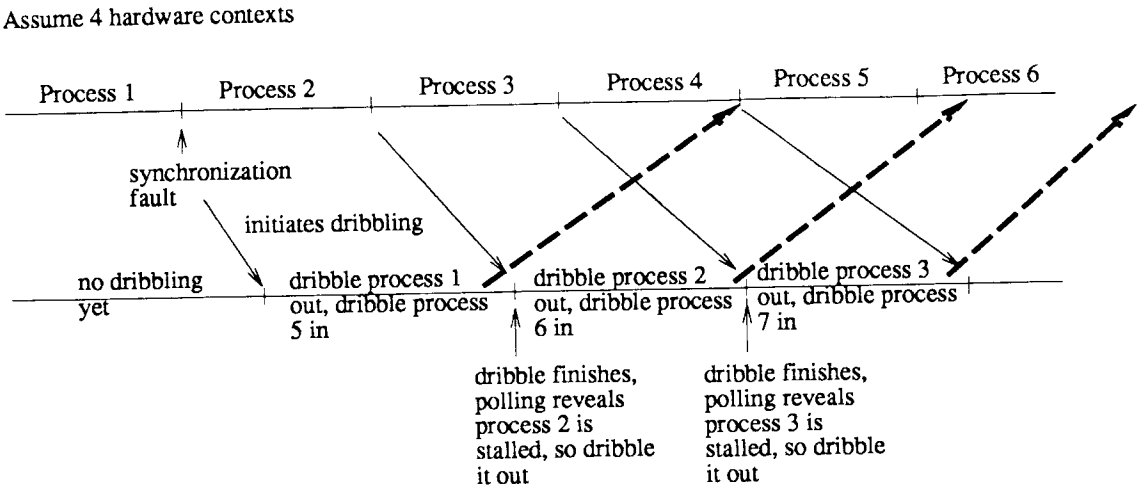

Figure 4-2: Dribbler operation when run lengths are on average shorter than dribble length.

will not be continually dribbling, as figure 4-3 illustrates. Notice the dribbler does not always have contexts to dribble out, and is idle for some portion of every run length. This case is one in which the utilization tends toward its maximum of 1, since $1/L < 1/U$. If the dribbler is doing its job it will not have to work for a portion of every run length, as 4-2 suggests.

Figure 4-4 shows the agreement of this simple model with simulation. The simulation is run assuming single-cycle load/stores (due to caches), enabling a dribble duration of 64 cycles. In addition, load/store instructions (which would otherwise interrupt dribbling) are disabled. As the figure indicates, this model is quite accurate for short run lengths. In these regions the dribbler is continually removing stalled contexts from the register file because the run lengths are usually less than the dribble latency so the dribbler is constantly required to clean out the register file.

At run lengths greater than the dribble time, this model breaks down because the run length, determined by a Bernoulli process (which determines when synchronization points happen), can take on values less than the expectation value. This behavior is not modeled in this simple queueing model. If the run length happens to be shorter than the dribble time at some time, then the dribble may cause a wait. This type of corner case is ignored in the model, but the accuracy is still excellent. Even at

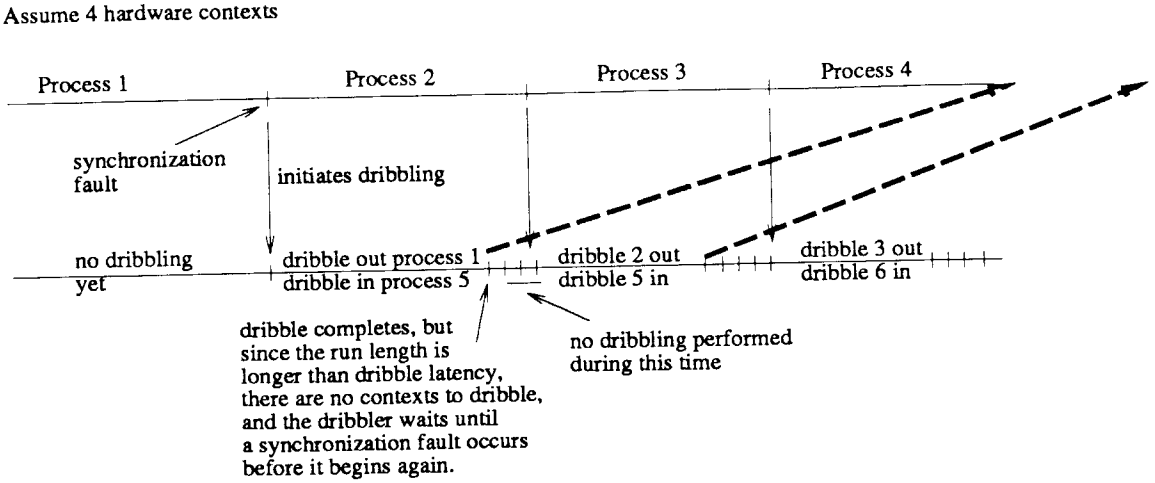

Figure 4-3: Dribbler operation when run lengths are on average long than dribble length.

run lengths of 50 (where dribble time is 64) the error is only 6.4 percent, and at run lengths of 30 and 40 there is perfect agreement between model and simulation. It is at these small run lengths that the dribble rarely finishes without being able to find another context to remove. The context switch time in all of these simulations was set to be 1 cycle, and the wait time for the satisfaction of a synchronization was set to be large in order to allow more threads to be scheduled in before any had their wait times satisfied. We wished to minimize successes due to a series of synchronization fault requests being satisfied, since such satisfactions would increase utilization artificially, and would not be a function of the effectiveness of the dribbler. By making the wait times large (on the order of 2000 cycles) and calibrating the wait with the total number of runnable threads, a steady queue of ready threads was achieved (i.e. wait times would be satisfied just as the final runnable threads were being loaded, so that when these loaded threads stalled the threads first stalled would again be ready for scheduling, simulating an infinite supply of threads).

With this primitive model we can gain some valuable insight as to the behavior of dribble-back register files. First, to first order utilization does not depend on the number of contexts, but simply the ratio of average run length to average dribble duration. Having 3 hardware contexts is just as effective (but less expensive from an

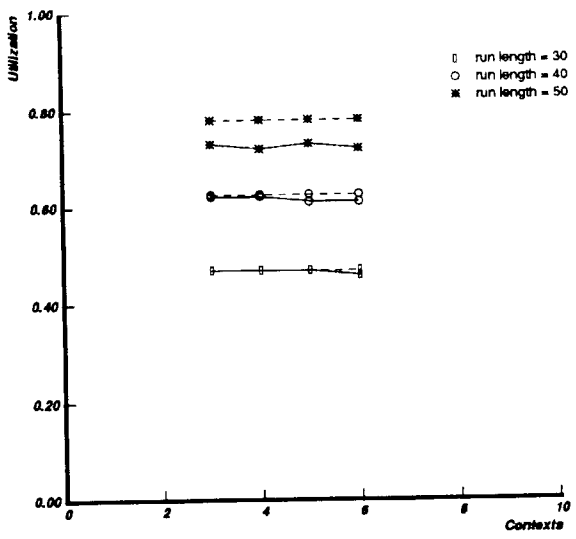

Figure 4-4: Queueing model vs. simulation, ignoring load/store instructions. Dashed lines are model predictions, solid lines are simulation results

implementation standpoint) than having 6 hardware contexts in solving this problem of long synchronization latencies. In addition, we see that with just 3 hardware contexts we approach the limit of utilization for a particular synchronization frequency: the close agreement with the model (which assumes infinite hardware contexts) at short run lengths confirms this conjecture. At larger run lengths $1/L < 1/U$, we expect in the steady state a utilization close to 1.

4.1.2 Load/store instructions approximation

The reason we must consider load/store instructions is that these instructions require use of the same memory bus that the dribbler uses, so dribbling cannot be done on these cycles. We can make a simple adjustment to the above model to approximate behavior with load/stores. Disregarding load/stores that result in local cache misses and require remote memory accesses (we assume these happen infrequently enough for their effects to be ignored), we can think of single-cycle load/stores as simply extending the effective length of a dribble (i.e, the dribble takes more time). The number of useful cycles does not change: rather, some of the instructions become load/stores when they weren't before (in the previous model). Thus, the frequency of

synchronization faults remains the same as before, and only the frequency of dribble completions changes.

Using the same notation as before, we note that 1/U, the average time between synchronization faults, remains constant even with load/store instructions. 1/L, the dribble service time, increases by the number of load/store cycles in 1/U. Denote the fraction of instructions of 1/U that are load/stores by α. If we make an approximation that we are always dribbling, and assume that the probability of a load/store on any cycle is α (as opposed to splitting up the formula into cases in which $1/L < 1/U$ and $1/L > 1/U$, we are assuming that the probability a cycle is not available for dribble is α), we obtain

$$\frac{1}{L_{eff}} = \frac{1-\alpha}{L}, \tag{4.3}$$

where $\frac{1}{L_{eff}}$ is represents the increased effective dribble length. Thus,

$$\rho = \frac{L_{eff}}{U}. \tag{4.4}$$

This formula is valid when $\frac{1}{L_{eff}} > \frac{1}{U}$, with an error of less than 10% in this region. When this is not the case, we expect a utilization that approaches 1, since the dribbling finishes within the duration of the run length even with the load/stores included.

We see in figure 4-5 that with moderate values for the percentage of load/stores (in this case, percentage of load/stores = 20, cache hit rate = 91%), this model is able to give good estimation of performance, and predicts the trends accurately.

We can see that for quick first-order approximations of utilization the queueing model suffices. It accurately predicts trends in performance (namely, the non-reliance of utilization on the number of hardware contexts), and provides less than 15% error in most cases. However, it cannot be used when the average run length exceeds the dribble latency. In Chapter 5 we will discuss the simulation methodology in more detail, and in Chapter 6 compare DIDO versus the context cache and regular multiple register sets in order to determine whether DIDO merits attention as a multiprocessor register file design.

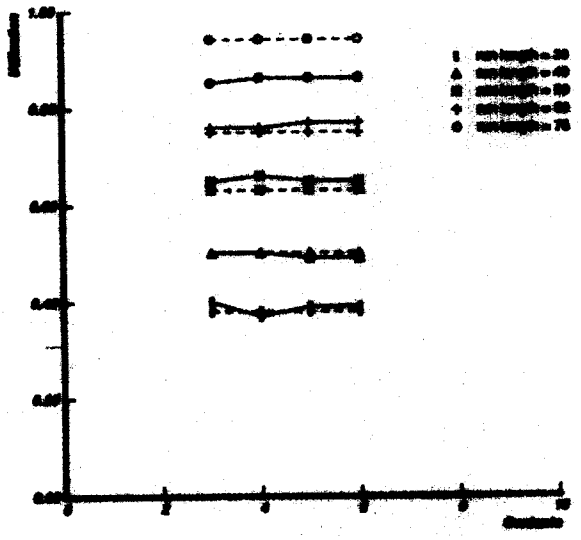

Figure 4-5: Queueing model vs. simulation, including load/store instructions. Dashed lines are model predictions, solid lines are simulation results

Chapter 5

Experimental Framework

This chapter describes the experimental methodology, the data relevant to the research, and the experiments that were carried out. To evaluate the performance of the DIDO register file a functional simulator was written. Functional simulators for multiple register set and Context Cache designs were also written and a comparative evaluation was used to determine the effectiveness of the DIDO scheme.

5.1 Simulators

5.1.1 General Information

Each simulator takes as input information the following:

1. the number of runnable threads
2. the number of hardware contexts (in the Context Cache simulator this number is converted into an appropriate number of registers)
3. the number of registers in a context
4. the frequency of synchronization faults
5. the frequency of load/store operations
6. the cache hit rate on load/store operations

7. the length of the wait time

8. the lengths of the individual threads

Each of these (except for the number of registers in a context and the number of hardware contexts) is a workload parameter used to drive the functional simulator.

Each simulator takes these workload parameters and synthetically generates register file traces from them. A register file trace is simply a list of register accesses on a cycle-by-cycle basis. Although register file accesses are generally done as two reads and one write per cycle, for ease of analysis and implementation the traces produced by the simulators contain only one read or one write reference per cycle. These traces are accumulated and then fed into the functional simulators.

The functional simulators sequence through these traces. We assume exponentially distributed run lengths for our analysis and simulation. In order to simulate exponential distributions, a Bernoulli test (i.e., a coin flip with an appropriately biased probability of success or failure) is performed on each instruction sequencing cycle to see whether a synchronization fault is to occur. In the limit of a large number of trials a Bernoulli process closely approximates an exponential distribution. When a synchronization fault occurs, the process that faulted is assigned a wait time. This process cannot be reactivated and run until the wait time has completed.

The frequency of load/store instructions is also modelled as a Bernoulli process. On any instruction execution cycle that did not result in a synchronization fault, a separate coin flip is performed to determine whether that instruction is a load/store instruction. If it is not a load/store, sequencing proceeds through the trace. If it is a load/store, however, a final coin flip determines whether the data access results in a cache hit or miss. On a cache hit, an instruction is still executed, but on a cache miss, a wait time (exponentially distributed) is assigned to the process that missed. As with synchronization faults, this process cannot be reactivated and run until the wait time has completed. The decision tree that we observe on each useful cycle is displayed in figure 5-1. Load/store operations are assumed to be single-cycle (excluding cache misses), and the cache is assumed to be large enough to store all

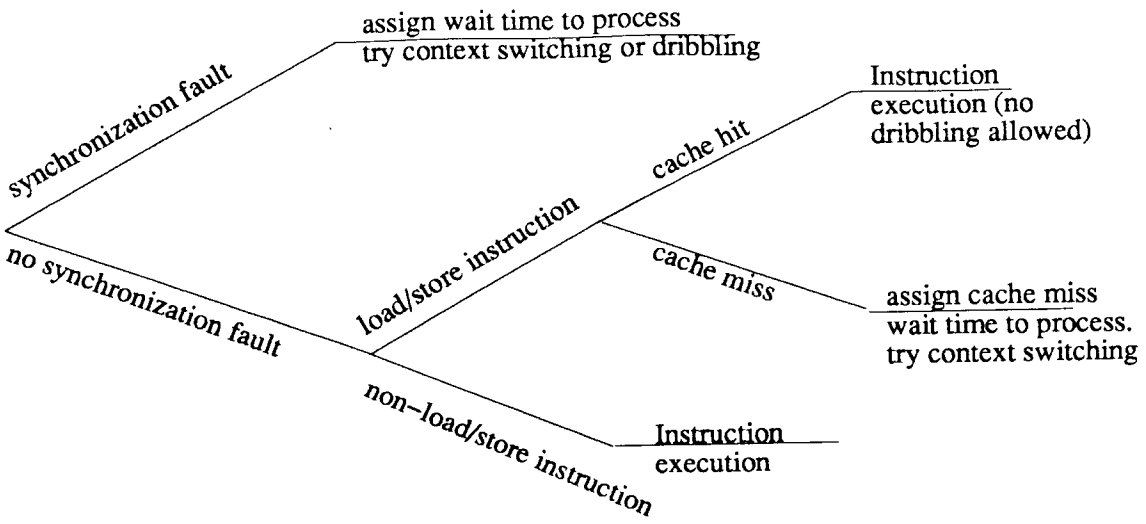

Figure 5-1: Decision tree for useful cycles

context information, so that there are no cache misses when loading the register file with a context.

The simulators require a finite number of threads and a finite wait time specified. However, the analyses presented within this thesis assume there are always runnable threads and that wait times are infinite. With wait times short, there is potential for threads that stall to be quickly ready again, thus eliminating, for example, the need for unloading. We wish to force the need for loads in order to evaluate the dribbler against the Context Cache and multiple register sets under worst case conditions. To force loading, waits are made long enough that all of the threads will have each been scheduled at least once before the wait time can be satisfied.

5.1.2 Multiple Register Sets Simulator

The simulator for multiple register sets (MRS) operates according to simple rules. Whenever a synchronization fault occurs, a context switch is attempted. If no resident contexts are free to switch to, a context load is initiated. If there are no runnable threads the scheduler idles until a process becomes ready to run.

On load/store instructions that result in cache misses, the scheduler attempts a context switch. If there are no free contexts available, the scheduler idles until one is

ready. A load is not performed in this case because cache miss latencies are generally shorter than load/unload overhead.

The default context switch overhead is 1 cycle. Although this value is smaller than usual best-case context switch time in multiprocessors (which is usually 4-5 cycles of pipeline flushing and pointer swapping), as long as context switch time is less than fault latencies this number changes utilizations only slightly. It was chosen this small in order to de-emphasize the context switch overhead relative to the dribbling and loading overhead. The default average cache miss latency is 40 cycles. The time for a load is 64 cycles: physically, this implies that we save out 32 registers and load in 32 registers, with 1 cycle used for each transaction. The context switch overhead is made low so as to de-emphasize its effect on utilization. In practice, unless the context switch overhead is on the order of the cache miss latency or load overhead, it is negligible, at least for the purposes of this thesis.

5.1.3 Dribble-in, Dribble-out Register File Simulator

The simulator for the dribble-in, dribble-out (DIDO) register frames operates exactly as described in the description chapter. When the first synchronization fault occurs, the dribbler initiates a context load. After each dribble completes, the dribbler polls the resident contexts for any contexts stalled on synchronization faults.

On load/store instructions all dribbling is suspended for the duration of the instruction to allow the load/store to complete. A context switch is attempted on cache misses, and if no contexts are available then dribbling is suspended until the cache miss is serviced. If a ready context is available, dribbling is suspended until the context switch is complete. It is then assumed that the cache miss only disables the dribbler when the data returns (i.e., when the wait is satisfied). On a cache hit dribbling is suspended for 1 cycle. As with multiple register sets, contexts that are stalled on cache misses are not unloaded, and the dribbler ignores such contexts when polling for contexts to remove from the register file.

The default context switch overhead is 1 cycle. The default average cache miss latency is 40 cycles. The total time to dribble in a new context and dribble out an

old context is 64 cycles.

5.1.4 Context Cache simulator

The Context Cache simulator operates differently from each of the other simulators because there is no crisp differentiation between a context load and a context switch in terms of simulated behavior. The registers in the register file are not partitioned into contexts. Rather, as registers are used they are loaded into the register file and the register is tagged with a process and register offset identification. A context switch involves loading only the registers necessary for a particular computation (i.e., the registers referenced on that particular instruction), and if the registers are already present, such loading is unnecessary. Synchronization faults disable processes for the appropriate wait time, and disallow any registers used by that process to be loaded or referenced until the wait time is satisfied.

On load/store operations that result in cache misses, the process is tagged as stalled until the satisfaction of the cache miss. Cache hits result in instruction executions, and do not otherwise affect performance.

When the Context Cache fills and registers must be chosen to be removed, the scheduler looks first for registers belonging to stalled contexts. It will never remove registers that correspond to the current context. In addition, there is a most-recently-used counter, which prevents registers that have been used recently (recently is defined by the value of this counter) to be removed.

The context switch time for the Context Cache is 1 cycle. The default cache miss latency is 40 cycles. The time required for a register load is 1 cycle, but if it must displace a register already resident in the Context Cache, the load/unload time is 2 cycles–1 cycle corresponding to the time required to unload the old register, and 1 cycle corresponding to the time required to load in the new register.

5.2 Experiments

Because the register traces are randomly produced, no real benchmark programs are actually run through the simulators. However, parameters for the workload model are average numbers taken from real parallel applications[6, 8].

The workload model provides trace lengths for each of the threads. The simulator runs until one thread completes. This methodology is chosen in order to prevent end effects like running out of threads and waiting continually as threads complete from corrupting the data measured.

Some architectural assumptions (like single-cycle load/stores and a very large cache with a very high hit rate to store thread information and avoid thrashing) are made to simplify the implementation of the simulators. These assumptions reflect an important aspect of this thesis: it is not the actual numbers that we obtain through our simulations that necessarily matter. In some cases assumptions are not completely consistent with hardware implementations previously performed. What is important is the relationships between the various factors that influence the behavior of the simulated hardwares, and the relative performances of the hardwares to each other. By making the same base architectural assumptions for each type of register file we hope to make useful contributions without having completely hardware-accurate functionality.

The metric of study is processor utilization. This is measured as the total number of useful cycles divided by the total number of cycles executed. A useful cycle consists of any cycle on which an instruction is executed. A wasted cycle is any other kind of cycle, including cycles spent waiting for remote memory accesses and synchronization faults, cycles spent loading/unloading contexts (unless overlapped with instruction executions as in DIDO), or context switch cycles.

We present processor utilization figures for each register file under equivalent workload parameters for various run lengths and load/store rates, and draw conclusions about the performance of DIDO by comparison with the other two register file architectures in Chapter 6.

Chapter 6

Results

6.1 Experimental Results

In this chapter we present the results of our simulations. Tables 6.1 and 6.2 display the data obtained from the sets of simulations run. ρ is processor utilization. The DIDO scheme appears to outperform multiple register sets over all ranges of grain sizes. In addition, it performs well even when load/store instructions are relatively frequent, indicating the amortization of instruction execution afforded by dribbling works at reducing loading latencies even when dribbling is not done under optimal conditions.

At extremely small run lengths, the dribbler performs slightly worse than the context cache. We can explain this behavior by noting that the context cache loads specific context information only upon demand. When run lengths are short, there is less chance for every register in a context to be accessed, and as a result loading all registers (as the dribbler does) causes extra register loads. The Context Cache does not provide this overhead, and the elimination of excess register loads results in excellent fine grain performance. Because fewer registers are touched per run length, the information for more contexts can be stored in the Context Cache.

At moderately small run lengths, however, and average load/store conditions[6], the dribbler outperforms the Context Cache. At run lengths slightly larger than the dribble overhead the DIDO provides dramatic increases for processor utilization

threads	contexts	run length	ρ (DIDO)	ρ (MRS)	ρ (CCache)	load/store ratio	cache hit rate
30	3	30	.38	.28	.49	.20	.91
30	4	30	.40	.29	.49	.20	.91
30	5	30	.39	.31	.49	.20	.91
30	6	30	.39	.32	.49	.20	.91
30	3	40	.50	.33	.57	.20	.91
30	4	40	.50	.35	.58	.20	.91
30	5	40	.49	.36	.58	.20	.91
30	6	40	.49	.38	.58	.20	.91
30	3	50	.65	.37	.61	.20	.91
30	4	50	.66	.39	.61	.20	.91
30	5	50	.65	.41	.62	.20	.91
30	6	50	.65	.43	.62	.20	.91
30	3	60	.76	.42	.63	.20	.91
30	4	60	.76	.42	.63	.20	.91
30	5	60	.77	.44	.63	.20	.91
30	6	60	.77	.46	.64	.20	.91
30	3	75	.85	.43	.66	.20	.91
30	4	75	.86	.45	.66	.20	.91
30	5	75	.86	.48	.66	.20	.91
30	6	75	.86	.50	.66	.20	.91
30	3	100	.89	.49	.70	.20	.91
30	4	100	.89	.49	.70	.20	.91
30	5	100	.89	.51	.70	.20	.91
30	6	100	.89	.54	.70	.20	.91

Table 6.1: Comparison of DIDO, MRS, and Context Cache, load/store percentage = 20%, cache hit ratio = 91%

threads	contexts	run length	ρ (DIDO)	ρ (MRS)	ρ (CCache)	load/store ratio	cache hit rate
30	3	30	.33	.26	.49	.31	.91
30	4	30	.35	.27	.49	.31	.91
30	5	30	.33	.28	.49	.31	.91
30	6	30	.34	.30	.49	.31	.91
30	3	40	.44	.30	.58	.31	.91
30	4	40	.43	.32	.58	.31	.91
30	5	40	.44	.34	.58	.31	.91
30	6	40	.44	.35	.59	.31	.91
30	3	50	.51	.33	.62	.31	.91
30	4	50	.54	.36	.62	.31	.91
30	5	50	.54	.38	.62	.31	.91
30	6	50	.52	.40	.63	.31	.91
30	3	60	.68	.36	.64	.31	.91
30	4	60	.69	.38	.64	.31	.91
30	5	60	.67	.40	.64	.31	.91
30	6	60	.68	.43	.64	.31	.91
30	3	75	.82	.38	.66	.31	.91
30	4	75	.83	.41	.66	.31	.91
30	5	75	.82	.43	.67	.31	.91
30	6	75	.83	.45	.67	.31	.91
30	3	100	.86	.41	.70	.31	.91
30	4	100	.86	.44	.70	.31	.91
30	5	100	.85	.47	.70	.31	.91
30	6	100	.85	.49	.70	.31	.91

Table 6.2: Comparison of DIDO, MRS, and Context Cache, load/store percentage = 31%, cache hit ratio = 91%

than the context cache. At such run lengths, there is higher probability that more registers will require access. In such instances, the Context Cache loads more and more registers, and each process will have to remove more state from the Context Cache in order to fit its information within the register file. Because there are fewer spurious loads with larger run lengths than with smaller run lengths (i.e., when the run lengths are large, there is a high chance that all registers will be accessed, so that DIDO and MRS do not load as many registers that go unused), the Context Cache does not cut much from the average loading overhead of the multiple register set design. DIDO, on the other hand, still cuts a great deal from the typical multiple register set overhead, and as a result provides high utilization even when run lengths are comparable to dribble time. Even at above average load/store conditions, the dribbler outperforms the Context Cache at run lengths slightly higher than the dribble overhead. However, the Context Cache is specifically intended for use in an extremely fine-grained (20 instructions per context switch) processor[Nuth2], so a comparison to the Context Cache in this instance is not a completely meaningful or fair comparison.

Another important result of these simulations that confirm our analytical findings is that the added performance that the dribbler affords does not increase significantly with more than three hardware contexts. Not only does three contexts seem sufficient to provide maximal performance, but this performance approaches infinite register file size performance.

.6.2 Conclusions

The dribble-in, dribble-out register file scheme is one that affords increased performance over other register file designs when run lengths are not too short and cache misses are not overly frequent. In particular, DIDO beats multiple register sets under average load/store conditions and all run lengths because of the load/unload latency reduction its polling behavior guarantees. The Context Cache outperforms DIDO at extremely short run lengths, but its approach is specialized toward this range and for longer run lengths DIDO provides better processor utilization.

For DIDO, increasing the number of hardware contexts beyond three does not significantly improve performance. As little as three hardware contexts provides performance close to that expected of an infinitely large register file. In this limit a simple queueing model can accurately predict processor utilization. DIDO also alleviates cache miss latencies by increasing the probability that on a fault there will be a runnable resident context available to which the stalled thread can switch. By reducing waits caused because no contexts are available for switching DIDO allows fast context switching,

There are some disadvantages to DIDO when compared to multiple register sets and the Context Cache. First, it requires an on-chip instruction cache so that the bus to off-chip memory is not used on every cycle (as it is in SPARC). Second, its effectiveness relies on the number of load/store operations in each process. With a large number of load/stores fewer cycles are free for dribbling and the gains DIDO provides are decreased. As mentioned earlier, a smart compiler that stores frequently used variables in the register file rather than in off-chip main memory can partially solve this problem. Third, it requires more complicated control circuitry and a more complicated RAM cell design (to accommodate the extra ports) than multiple register sets. As a result of the extra ports and the extra capacitance that word lines would have to drive, DIDO reads and writes will be slower than reads and writes in a multiple register set. DIDO is less complicated to implement than the Context Cache, however, requiring less complicated decode logic (owing to the fully-associative nature of the Context Cache.

During the early stages of the project, a VLSI implementation of the register file was undertaken to test the idea and to determine feasibility of the design. The main concern from a feasibility standpoint is whether the two dribbler ports add too much capacitance to the memory cell such that performance is significantly degraded. SPICE simulations showed that it is indeed possible to properly size the RAM cell transistors to correctly store information. The project was a good demonstration as all vital parts of the register file were implemented, including an array of the 5-ported cell and crucial pieces of the control circuitry, namely the register sequencing logic,

in order to perform dribbling. The sequencing logic can be implemented very easily: by using parallel shift registers one can sequence through the word lines of the read ports and write ports without requiring any counters or decode logic for the individual registers. The DIDO is estimated to be 5-7% slower. Although multiple register sets is cheap to implement (it can be cheaply realized by leveraging off currently available register window architectures like SPARC), we see by these results that the performance gains a DIDO provides seem to overshadow the costs of implementation.

6.3 Future Work

There are many directions for future work in this area. We have shown that the dribbler merits consideration in multiprocessor design. A more extensive study using real traces and more complete trace-driven simulation (rather than randomized trace-driven simulation) could and definitely should be performed. In addition, more detailed functional simulators that more accurately model the hardware could be written and interfaced with current network and memory simulators to provide more accurate evaluations of performance. Running the dribbler with real applications using more realistic simulations would provide a better indication as to the true gains it can afford over multiple register sets.

Another direction for future research on this topic involves different heuristics for determining when to dribble, and studies under different conditions. For example, how effective is the dribbler when there is less parallelism than assumed here? In addition, this analysis assumed all threads were created equal, but another topic of study might be speculative loading of threads: i.e., given some threads need to be performed before others, is there some degree of prefetching of threads before they may be used that would increase performance and speed program execution? The dribble-in, dribble-out register file presented here leaves many questions for future work before it can be considered as an alternative to today's register files, but its outlook is promising.

Bibliography

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[2] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings 16th Annual International Symposium on Computer Architecture*, June 1989.

[3] Peter R. Nuth and William J. Dally. A Mechanism for Efficient Context Switching. Artificial Intelligence Lab Technical Memo

[4] Peter R. Nuth. Parallel Processor Architecture: A Thesis Proposal. MIT VLSI Memo, 1990.

[5] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[6] Jaswinder Pal Singh, Wolf-Dietrich Weber and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. To appear in SIGARCH, Computer Architecture News, Spring 1992

[7] Kiyoshi Kurihara. *Performance Evaluation of Large-Scale Multiprocessors.* Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1990.

[8] Beng-Hong Lim. *Waiting Algorithms for Synchronization in Large-Scale Multiprocessors.* Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1991.

[9] Richard L. Sites How to Use 1000 Registers. In *Proceedings, 1st Caltech Conf. VLSI*, California Institute of Technology. January 1979

[10] Stephen A. Ward and Robert H. Halstead, Jr. Computation Structures. Cambridge, MA: MIT Press. 1990

[11] Herbert H. J. Hum and Guang R. Gao. Efficient Support of Concurrent Threads in a Hybrid Dataflow/von Newmann Architecture In *Third IEEE Symposium on Parallel and Distributed Computing*, 1991