ARCHITECTURAL AND IMPLEMENTATION TRADEOFFS FOR MULTIPLE-CONTEXT PROCESSORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
James P. Laudon
September 1994

ARCHITECTURAL AND IMPLEMENTATION TRADEOFFS FOR MULTIPLE-CONTEXT PROCESSORS

James P. Laudon

Technical Report: CSL-TR-94-634

September 1994

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford California, 94305-4055

Abstract

Tolerating memory latency is essential to achieving high performance in scalable shared-memory multiprocessors. In addition, tolerating instruction (pipeline dependency) latency is essential to maximize the performance of individual processors. Multiple-context processors have been proposed as a universal mechanism to mitigate the negative effects of latency. These processors tolerate latency by switching to a concurrent thread of execution whenever one of the threads blocks due to a high-latency operation. Multiple context processors built so far, however, either have a high context-switch cost which disallows tolerance of short latencies (e.g., due to pipeline dependencies), or alternatively they require excessive concurrency from the software.

We propose a multiple-context architecture that combines full single-thread support with cycle-by-cycle context interleaving to provide lower switch costs and the ability to tolerate short latencies. We compare the performance of our proposal with that of earlier approaches, showing that our approach offers substantially better performance for parallel applications. We also explore using our approach for uniprocessor workstations — an important environment for commodity microprocessors. We show that our approach also offers much better performance for multiprogrammed uniprocessor workloads.

Finally, we explore the implementation issues for both our proposed and existing multiple-context architectures. One of the larger costs for a multiple-context processor arises in providing a cache capable of handling multiple outstanding requests, and we propose a lockup-free cache which provides high performance at a reasonable cost. We also show that amount of processor state that needs to be replicated to support multiple contexts is modest and the extra complexity required to control the multiple contexts under both our proposed and existing approaches is manageable. The performance benefits and reasonable implementation cost of our approach make it a promising candidate for addition to future microprocessors.

Key Words and Phrases: multithreading, multiple-context processors, tolerating latency, lockup-free caches, microprocessor implementation, shared-memory multiprocessors.

Acknowledgments

This work has greatly benefited from all the people I had the pleasure of working with at Stanford. First, my sincerest thanks go to my advisor, Anoop Gupta. Anoop was the fearless leader of the DASH project and once we had finished building the DASH prototype, helped to steer me towards an indepth study of multiple-context processors. Anoop was also never short of good advice along the course of my explorations and impressed upon me the need to study a topic in enough detail to insure you understand all its subtleties. I would also like to thank my co-advisor, Mark Horowitz. Mark provided invaluable advice on processor implementation issues and had a knack for asking the hard questions that led to new insights. I also thank John Hennessy for his advice and support and Giovanni De Micheli for serving as my orals committee chairman.

I also thank the friends and colleagues that made my time at Stanford both enjoyable and enlightening. In particular, I would like to thank the other members of the DASH team: Dan Lenoski, Luis Stevens, Dave Nakahira, Truman Joe, Wolf Weber, and Kourosh Gharachorloo, for making the building of DASH such an exciting and learning experience. I thank Mike Smith for providing the twine scheduler and masm assembler and for being so willing to fix the problems I uncovered. Steve Goldschmidt and Helen Davis provided Tango-Lite, a tool integral to my simulation studies. Grant McFarland was kind enough to design the multiple-context register file as a class project. Thanks also to Margaret Rowland and Darlene Hadding for making the job of jumping the administrative hoops so easy and to Charles Orgish and Laura Schrager for keeping the machines running smoothly. Finally, I would like to thank the trailer crew: J. P. Singh, Todd Mowry, and Andrew Erlichson, for their friendship, interesting (and sometimes seemingly unending) conversations, and general high jinks.

I thank Jim Smith for giving me that initial push that started me on the road to my Ph.D. I also thank my parents for all the love, support, and encouragement they've given me over the years. Most importantly, I thank my wife, Karen, who helped me explore the bay area and allowed me to keep things in perspective when the hours at Stanford got too long.

This work was supported by DARPA under contract N00039-91-C-0138. I also gratefully acknowledge support by IBM through the CIS Fellow-Mentor-Advisor program.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The increasing computational requirements of applications has resulted in considerable interest in the design of shared-memory multiprocessors which can scale to large numbers of processors [CGB91, KCA91, BFKR92, Hag92, LLJ+93]. Most of these *scalable* shared-memory multiprocessors have a similar basic structure, as shown in Figure 1.1. Memory is distributed with each processing node and the nodes are connected via a low-latency, high-bandwidth network. Cache coherence is maintained through the use of a directory-based coherence protocol [Tan76, CF78, CKA91, Len92, Web93], where the directory is distributed along with the memory and keeps information on which caches are sharing each memory location. The distributed nature of memory on these machines results in low-cost accesses to cache and local memory; however, cache misses which require remote memory accesses are fairly expensive.

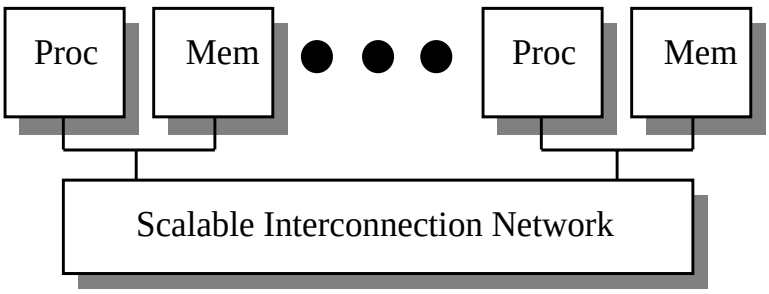


Figure 1.1: General structure of a scalable multiprocessor.

Since these multiprocessors harness together large numbers of high-performance processors, their peak performance is quite impressive. In practice, however, many applications realize only a small fraction of the peak performance. One of the key obstacles limiting application

performance is the long latency of remote memory operations [GHG+91, CB92]. By exploiting techniques to avoid or tolerate these long-latency memory operations, application performance can be significantly improved.

There are several ways to avoid memory latency. These include the caching of shared data, which allows the processor to keep a copy of frequently referenced data close by, thereby exploiting temporal and spatial locality in the data stream. The application can also be restructured to further take advantage of cache locality (e.g. by blocking the data accesses for the cache [LRW91]) and memory locality (e.g. allocating data on the local memory of the processor most likely to use it). While all of these methods improve the amount of computation a processor performs before requiring a long-latency memory access, often applications still end up spending a large portion of their time waiting on memory references [GHG+91]. To deal with this remaining latency, several latency tolerating schemes have been proposed, including relaxed memory consistency models [DSB86, SD87, AH90, GLL+90, Goo91], prefetching [CKP91, CB92, MLG92] and multiple-context processors [Smi81, HF88, WG89b, ALKK90, ACC+90, BR92, LGH92].

Relaxed memory consistency loosens the ordering constraints between specific memory operations to allow buffering and pipelining of memory references. Prefetching relies on being able to predict the future use of a data item. Data references whose access can be predicted in advance are then brought into the cache via nonblocking memory operation, with the goal that by the time the data is actually needed, it will reside in the cache. Multiple-context processors tolerate latency by overlapping the long-latency operations of one thread of computation with the execution of other thread(s) and are an interesting and flexible method for tolerating many forms of latency. However, while prefetching and relaxed consistency models are being incorporated into the latest microprocessor designs [Dig92a, JMY92, Hei93], multiple contexts has so far largely been ignored.

In this thesis, we focus on the multiple context solution. We examine the reasons behind its lack of acceptance by microprocessor designers. We then propose a multiple-context architecture capable of addressing these concerns. We look in detail at the performance and implementation complexity of both our proposed architecture and the current multiple-context approaches, showing that our proposed architecture makes it possible to build a multiple-context processor that provides good performance across a range of systems at a reasonable implementation cost.

1.1 The Latency Problem

In addition to the large memory latencies possible in a scalable shared-memory multiprocessor, application performance on these machines suffers from the latency of inter-process synchronization, and instructions which take multiple cycles to produce their result. In this thesis, we will refer to these three types of latency as: *memory*, *synchronization*, and *instruction* latency, respectively.

For most applications that have been studied on large-scale, shared-memory multiprocessors, memory latency dominates synchronization and instruction latency in determining an application's performance, even after providing coherent caches and structuring the applications to take advantage of the cache and memory hierarchy [GHG+91, KCA91, LGH92]. The importance of memory latency in determining parallel application performance is based on two conspiring factors. First, because different processes are cooperating on the same problem, data communication between the processes adds to the cache miss rate. This communication component of the cache miss rate is often much larger than uniprocessor cache miss effects, and can dominate the cache miss behavior [WG89a]. Second, the latency for cache misses is larger than in a uniprocessor due to the physical distribution of memory. Increased miss rates due to communication and the long memory latency have a multiplicative effect, resulting in a large amount of time being lost in the memory system.

Synchronization and instruction latency play a secondary, but important, role in determining multiprocessor utilization [KCA91, LGH92]. Synchronization latency includes any time one process spends waiting on another process, such as time spent in barriers to synchronize all processes and time spent acquiring and releasing locks protecting application critical sections. This latency varies widely by application and is very difficult to remove architecturally since it represents a control dependency between the processes. Finally, instruction latency arises due to pipelining of the processor, which causes some instructions to take multiple cycles to produce their result. Compilers schedule instructions to overlap much of this latency with execution of independent instructions, however, even aggressive compiler scheduling cannot tolerate all instruction latency.

The first step to addressing these three forms of latency is to reduce or avoid as much of the latency as possible. Latency reduction techniques include coherent caching and blocking transformations to lower memory latency, hardware and software primitives for fast synchronization, and fast functional units and pipeline bypassing for reduced instruction latency. Unfortunately, even

after aggressively applying these and other techniques, a processor can still spend a significant portion of its time idle [GHG+91, LGH92] and some form of latency tolerance, such as multiple contexts, must be employed to improve performance further.

1.2 The Multiple-Context Solution

Multiple-context processors share a single processor between several threads of computation, overlapping the latency encountered by one context with useful work by another context.[1] In order for multiple contexts to work well, the cost to switch between contexts needs to be much smaller than the long-latency operation to be tolerated.

The earliest multiple-context processors, such as the Denelcor HEP [Smi78] switched contexts each cycle, making the cost to switch contexts zero. This low switch cost allowed all three forms of latency to be tolerated. Unfortunately, each context in these early designs was limited to a single instruction being active in the pipeline. This constraint prevented pipeline dependencies from arising, allowing the processor design to be simplified, however, it placed two onerous burdens on applications. First, a large number of threads were necessary to fully utilize the processor — enough to both fill the pipeline and hide the memory latency. Second, the performance of a single thread was extremely poor, as each thread could issue a new instruction every pipeline-depth cycles at best. Therefore, any serial portion of an application could greatly impact the overall application performance. The limitations of these early designs were quite severe and most recent designs have instead focused on another technique for building multiple-context processors.

Multiple-context processors of this second type, exemplified by Weber and Gupta [WG89b] and the MIT APRIL [ALKK90], share the processor between a number of contexts; however, a single context utilizes all of the processor resources until it reaches a long-latency operation, such as a cache miss, at which point the processor switches to another context. Because blocks of instructions are executed between context switches, this second scheme has been referred to as *blocked* [FP91, KCA91]. The original HEP-style scheme has since been labeled *fine-grained* [KCA91].

Blocked multiple-context processors address the poor single-thread performance and need for large number of contexts of the fine-grained schemes, but they do so at the expense of increasing the context switch cost. The decision to switch contexts depends on determining

[1] In this thesis, we use the words *context* and *thread* interchangeably.

whether a cache miss occurred and this determination is made late in the pipeline. Without extensive modifications to the processor, the cost to switch contexts will be close to the depth of the pipeline, as the partially-executed instructions from the switching context will need to be flushed from the pipeline.

This high context switch cost prevents blocked multiple-context processors from being able to tolerate very short latencies, such as those caused by pipeline dependencies and primary cache misses where the miss is serviced by a secondary cache. Trading-off the ability to tolerate these smaller latencies for good single-thread performance makes sense for a multiprocessor, as multiprocessor performance is mainly determined by the effects of long-latency memory operations. However, the processor performance impact of multiple contexts should not be examined only for multiprocessor systems, as most multiprocessors are built using standard off-the-shelf microprocessors [Hag92, Thi92, LLJ+93, Cra93, GW94], in order to take advantage of their high performance and commodity costs. These off-the-shelf microprocessors are still primary sold for use in uniprocessors, making it important that multiple contexts are also able to address their latencies. Being able to tolerate shorter latencies is much more important for uniprocessors, where memory does not need to be distributed, making all memory access local, and not having to deal with communication traffic allows effective cache hierarchies to be built, reducing the frequency that even this nearby memory needs to be accessed. If commodity microprocessors are to ever incorporate multiple contexts, the switch cost must be lowered to allow uniprocessor latencies to be tolerated.

1.3 This Thesis

In Chapter 2, we examine previous multiple-context proposals in detail to determine if a multiple-context processor with a lower switch cost can be built. We then propose a new *interleaved* multiple-context processor which combines the cycle-by-cycle switching of the fine-grained schemes with the data caching and full single-thread support of the blocked scheme to achieve a low context switch cost and good single thread performance. We present some examples illustrating the potential performance advantages of the interleaved scheme over the blocked scheme, and then briefly discuss the implementation requirements of both approaches.

To quantify the performance advantages of the interleaved scheme, Chapter 3 presents a simulation study which explores the use of interleaved multiple-context processors in a shared-memory multiprocessor similar to the Stanford DASH [LLJ+93]. The interleaved scheme is

shown to provide significant performance gains over the blocked scheme in this environment. Chapter 3 also verifies that the performance advantages of the interleaved scheme hold over a number of multiprocessor architectural variations, such as when multiple contexts are combined with release consistency and nonblocking loads, and as memory latency and cache size and associativity are varied.

To evaluate the ability of the interleaved multiple-context processor in tolerating the smaller latencies of uniprocessors, we examine the effectiveness of multiple contexts on a multiprogrammed, high-performance uniprocessor in Chapter 4. We show that the blocked scheme does little to improve application throughput, whereas the interleaved scheme is able to show large increases in throughput due to its fast context switch and ability to tolerate instruction latency. Chapter 4 also shows that a multiprogrammed workload places larger demands on resources shared between the contexts, such as the caches and translation buffers, than was encountered for the multiprocessor applications. Despite the larger cache and TLB interference, the interleaved multiple-context processor is still quite effective in improving uniprocessor throughput.

Chapters 3 and 4 show that the interleaved scheme provides a significant performance advantage over the blocked scheme. To examine the other side of the performance/cost tradeoff, Chapters 5–7 examine the implementation costs for both schemes. Since the ability of the cache to support multiple outstanding requests is central to any multiple-context design, we first examine the design of the lockup-free cache in Chapter 5. We propose a lockup-free cache that keeps track of outstanding memory operations directly in the cache via a pending state. This pending state allows compatible requests to be merged and prevents conflicting requests to the same cache line from being issued, thereby simplifying the design of the lockup-free cache. The chapter also explores deadlock issues for the lockup-free cache and then finishes with a detailed performance analysis of the proposed lockup-free design.

We then examine the remaining issues involved in implementing both the blocked and interleaved schemes in Chapters 6 and 7, respectively. The costs beyond the lockup-free cache can be broken into two major components: state replication and context scheduling logic. We show that the amount of replicated state is small for both schemes. We also show that the context scheduling logic for the blocked scheme is simpler than the interleaved scheme, however, the scheduling logic for both schemes is relatively straightforward. Finally, Chapter 8 presents conclusions and future directions.

1.4 Contributions

The main contributions of this dissertation are:

- Development of the interleaved multiple-context architecture, which combines the low context switch cost and instruction latency tolerance of the fine-grained scheme with the coherent caches and good single-thread performance of the blocked scheme, allowing the interleaved multiple-context processor to tolerate much shorter latencies than the blocked scheme.
- A detailed performance study of the proposed and existing multiple-context architectures on a modern, deeply-pipelined processor, using both multiprocessing and multiprogramming workloads, which shows the proposed interleaved scheme to have significant performance advantages for both workloads.
- Proposal of a novel lockup-free cache design which allows only a single outstanding request per cache line, simplifying the cache implementation at a negligible cost in performance. A detailed performance study points out the importance of keeping occupancy of memory operations low and the negligible effect of collisions due to multiple requests to the same pending cache line.
- Detailed examination of the implementation issues for the proposed multiple-context architecture, showing that the amount of state which needs to be replicated is small. In addition, our implementation study shows that while the interleaved processor is more complex than a comparable multiple-context processor using the blocked scheme, this extra complexity is not overwhelming.

Chapter 2

A New Multiple-Context Processor

Existing multiple-context designs do not provide both a low context switch cost and full support for singlethreaded operation. Fine-grained processors have a very low switch cost, but poor single-context performance. On the other hand, blocked processors fully support the single context, but the cost of context switching is much higher than the fine-grained schemes.

This chapter starts by examining the previous multiple-context designs in more detail in Sections 2.1 and 2.2. In Section 2.3 we observe that the cycle-by-cycle switching of the fine-grained scheme can be combined with the coherent caches and full single-context support of the blocked scheme. This new multiple-context design, which we call *interleaved*, satisfies the twin goals of low switch cost and full single-context support. Several examples are presented to illustrate the differences between the new, interleaved multiple-context processor and the existing blocked scheme. We conclude Section 2.3 by presenting an overview of the implementation requirements of both the blocked and interleaved schemes. Related work is discussed in Section 2.4, and Section 2.5 summarizes the chapter.

2.1 Fine-grained Multiple-Context Processors

Multiple-context processors date back to the early 1960's, where they were used in the Control Data Corporation 6600 to time-share a CPU and memory interconnect between a number of peripheral processors [Tho64]. The earliest multiple-context processor proposals concentrated on the fine-grained architectures, with the seminal representative being the Denelcor HEP [Smi78, Smi81] built in the late 1970's. HEP consisted of a small number of processors interconnected to a number of memory modules via a packet-switched network. Fine-grained multithreading was

used by the processors to attack the problems of both pipeline stalls and memory latency. To address pipeline stalls, a context was prevented from issuing an instruction more than every eight cycles, which corresponded to the pipeline depth. Since an instruction could never encounter a pipeline dependency, no hardware or compiler resources had to be devoted to resolving pipeline hazards.[1] In addition, HEP provided the ability to tolerate the latency of a memory request by removing an instruction stream from the issue queue while memory was being accessed. Since HEP did not support data caches, large numbers of threads were required to hide both the pipeline and memory latency. For this reason, each HEP processor supported up to 128 active contexts.

HEP continues to evolve, and has been through two more incarnations. The Horizon [KS88, TS88] succeeded HEP. The processing elements were superscalar, capable of issuing three operations per cycle. The major architectural advancement of the Horizon was the addition of a lookahead field to each instruction, which encoded instruction dependencies, allowing multiple instructions to simultaneously use the pipeline. Lookahead provided limited compiler-resolved pipeline interlocks. Due to the size of the lookahead field, a maximum of eight instructions from the same context could be simultaneously executing. Unfortunately, like HEP, Horizon did not support caches. While instruction lookahead decreased the number of contexts needed for full pipeline utilization, a single context could still not fully utilize the pipeline. This is because the average response time of memory was expected to be around 50-80 cycles, and a lookahead of eight instructions is not nearly large enough to allow a pipeline this deep to be filled.

Horizon was a paper design. The Tera [ACC+90, AAC+91] computer is based on much of the Horizon design, and is currently being built by the Tera Computer Company. It enhances Horizon by improving the support for thread management and synchronization and by requiring lookahead only for memory operations. Memory lookahead helps to reduce the number of contexts needed to tolerate Tera's expected 70 cycle memory latency. Unfortunately, Tera ensures that register dependencies are respected by reverting to HEP's policy of limiting each context to a single instruction in the pipeline (although, unlike HEP, once an instruction has caused its memory reference to be issued, the next instruction can be issued if the lookahead for all previous instructions is satisfied). Since Tera is expected to have a pipeline of between 11 and 13 stages, at least this many contexts will be needed to fill the pipeline.

Kaminsky and Davidson also proposed another early fine-grained multiple-context processor [KD79]. The primary goal of their multiple-context proposal was not to hide memory or

[1] An exception to this is the divide unit, which could not sustain an issue rate of one divide per cycle.

pipeline latency, but rather to effectively utilize chip and pin resources. This is essentially another way of looking at the same problem, since increasing the utilization of chip and pin resources occurs by overlapping time that would be spent idle by one context with the active time of another context. The main focus of the paper is on a design methodology for utilizing chip resources efficiently when developing a multiple-context processor. Interestingly enough, as a case study, they look into the use of virtual registers, an idea that has been reexamined recently [ND91].

A few more recent architectures have been based on the fine-grained scheme, including MASA [HF88], DART [SB91], and the Stellar SPMP [SMM88]. However, most recent multiple-context designs have avoided the fine-grained scheme because of its inability to efficiently handle code with limited parallelism. As mentioned in Chapter 1, to overcome this limitation of fine-grained multiple-context processors, blocked multiple-context processors were proposed.

2.2 Blocked Multiple-Context Processors

The blocked multiple-context processor provides hardware for a small number of resident threads, but only executes a single thread at any given time. Long-latency operations are masked by switching to another thread. Several early blocked multiple-context processors were proposed for hiding specific long-latency operations or allowing costly resources to be shared. The Xerox Alto personal computer provided multiple microcode-level registers sets for sharing the CPU between the instruction set interpreter and the I/O devices [TML+82]. To prevent the I/O from interrupting atomic series of microcode operations, a field was added to the microcode to specify points at which context switching was allowed. Another early example was the Message-Driven processor [DCC+87], which provided two hardware contexts, one for normal processor execution and the other for handling high-priority messages.

The first published exploration of multiple-context processors in a shared-memory multiprocessor environment was performed by Weber and Gupta [WG89b]. They performed trace-driven simulation of three parallel applications, with their processor switching contexts at each primary cache miss. Even with a relatively fast memory (20–30 processor cycles), they found performance benefits for blocked multiple-context processors with a small number of contexts (two or four) per processor.

More recently, the Alewife multiprocessor being built at MIT [KCA91, ACD+91] is designed with a blocked multiple-context processor, APRIL [ALKK90]. APRIL supports four contexts in hardware and is based on a modified SPARC [Cyp90] processor. APRIL performs context

switching through a fast processor trap, switching contexts in 11 processor cycles. APRIL context switches whenever a memory request cannot be satisfied by the cache or local memory. In addition, APRIL provides the ability to force a context switch on a failed synchronization attempt. Using these two events as switch criteria, APRIL has the ability to tolerate the latency of both memory requests and synchronization.

While blocked multiple-context processors address the poor single-thread performance and need for large number of contexts of the fine-grained schemes, they do so at the expense of increasing the context switch cost. The high switch cost of the blocked multiple-context processor limits the types of latency which can be tolerated, and places an upper bound on the potential performance of the multiple-context processor. Assuming all memory latency is hidden by context switching, this upper bound is given in Equation 2.1.[2] The average runlength (the number of cycles between context switches) is broken into its components R_A, cycles active, and R_S, cycles stalled due to pipeline dependencies (which cannot be tolerated due to the high switch cost). C is the cost of a context switch. With memory latency completely tolerated, on average each context executes for $R_A + R_S$ cycles, followed by Ccycles for the switch to the next context. Of these $R_A + R_S + C$cycles, only R_A cycles are actually performing useful work. This upper bound can be fairly restrictive. For example, an application with a 20 instruction average runlength ($R_A = 20$), a CPI of 1.5 with a perfect memory system ($R_S = 10$), and a 10 cycle context switch cost will limit processor utilization to 50%.

$$Maximum Blocked Processor Utilization = \frac{R_A}{R_A + R_S + C} \tag{2.1}$$

In attempt to reduce this switch cost, a few blocked architectures have been proposed which replicate the pipeline registers [ND91, Omo91]. With this pipeline register replication, the context-switch cost could be a low as a single cycle (at least one cycle is needed to broadcast the switch decision to the entire chip for use by the TLB, pipeline forwarding logic, etc.) Unfortunately, pipeline register replication comes at a substantial implementation and performance cost. Replicating the pipeline registers results in a substantial increase in pipeline size, as latches which hold pipeline state are a significant fraction of the total pipeline area. In addition, the outputs of these replicated latches need to be multiplexed before being sent to the combinational portion of the pipeline. When these multiplexor delays are combined with the higher fanout and longer wire delays resulting from the area increase, it is difficult to imagine that the cycle time of the processor will not be significantly impacted. Instead of trying to reduce the switch cost

[2] A simplified version of this upper bound, not accounting for pipeline dependencies, is developed in both [SBCvE90] and [Aga92].

of the blocked scheme by brute force, a better approach can be developed by reexamining the fine-grained scheme.

2.3 Interleaved Proposal

If one looks closely at the two major problems with fine-grained processors, the need for large numbers of active contexts and the poor single-thread performance, it becomes apparent that the limitations of the fine-grained scheme are not due to the cycle-by-cycle context switching. Instead, two decisions incorporated into most fine-grained processors are the culprits. The first is the decision to not support data caching, which makes every memory reference a long-latency operation. The second is the decision to prevent a context from having more than one instruction in the pipeline, thereby increasing the minimum latency of each instruction to the pipeline depth.

By adding both caching and full single-thread support to the fine-grained scheme, it becomes possible to design a multiple-context processor which interleaves contexts on a cycle-by-cycle basis, yet effectively supports a single context. This *interleaved* multiple-context processor works as follows. Issuing of instructions is switched each cycle between available contexts in a round-robin fashion. Contexts become unavailable when they encounter a long-latency operation, and are made available when the long-latency operation completes. When a context becomes unavailable (an operation analogous to the context switch of the blocked scheme), the only instructions in the pipeline which are squashed are those of the context becoming unavailable. In addition, the cycle-by-cycle interleaving spaces out instructions from the same context, decreasing the probability that two dependent instructions from the same stream result in a pipeline stall.

2.3.1 Performance Advantages

The interleaved scheme achieves its lower switch cost by squashing only the instructions in the pipeline from the context encountering the long-latency operation. To illustrate this lower context switch cost, let us examine a context switch induced by a cache miss for both the blocked and interleaved schemes. We will assume that each processor has four active contexts (labeled A – D), and that context A encounters the cache miss. Figure 2.1 shows the overhead to tolerate the latency of the cache miss for the pipeline we will be using in our simulations. The exact details of each pipeline stage will be discussed later; the important impact of the pipelining is that the cache access occurs late in the pipeline, and therefore the context switch determination cannot be made until the WB stage. As mentioned earlier, the blocked scheme will need to squash

all instructions in the pipeline (including the instruction which caused the cache miss) before it can start the next context and therefore a context switch costs seven cycles on the pipeline shown.[3] However, for the interleaved scheme, instructions from all four active contexts are being interleaved. Since only the instructions from context A need to be squashed, the overhead to handle the cache miss of context A is reduced to two cycles. Of course, the context switch cost of the interleaved scheme will depend on the number of contexts being interleaved, but even with a small number of contexts will be much lower than that of the blocked scheme.

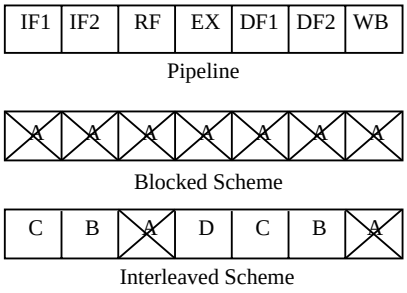


Figure 2.1: Example illustrating the lower context switch cost of interleaved scheme.

The interleaved scheme also benefits from the cycle-by-cycle context switching tolerating short instruction latencies. Like the pipeline of a single-context processor, the interleaved processor stalls when an instruction needs a result from a previous instruction which has not yet been computed. However, because there are no dependencies between instructions from different contexts, if sufficient instructions from other contexts are interleaved between dependent instructions from the same context, the instruction latency dependency can be hidden. An example of this instruction latency tolerance is shown in Figure 2.2. Context A executes two instructions: a LOAD into R4 (with the load result available for forwarding at the end of DF2), and an ADD which uses the value of R4 as one of its operands. With the blocked scheme, these instructions will issue back-to-back, and as shown in Figure 2.2, the pipeline dependency is resolved by introducing a two cycle pipeline bubble. This bubble allows the result to be properly forwarded from the DF2 stage of the LOAD to the EX stage of the ADD. Assuming that four contexts are active, the cycle-by-cycle switching of the interleaved scheme provides enough delay between the two dependent instructions that the result of the LOAD is available before the EX stage of the ADD. Thus, the pipeline does not need to stall to insure that the result from the LOAD is ready for the ADD. As with the context switch cost, the number of cycles of any given pipeline

[3] Alternately, the context switch cost can be viewed as the time needed by the next context to fill the pipeline.

dependency that can be tolerated by the interleaved scheme will depend on the exact context interleaving.

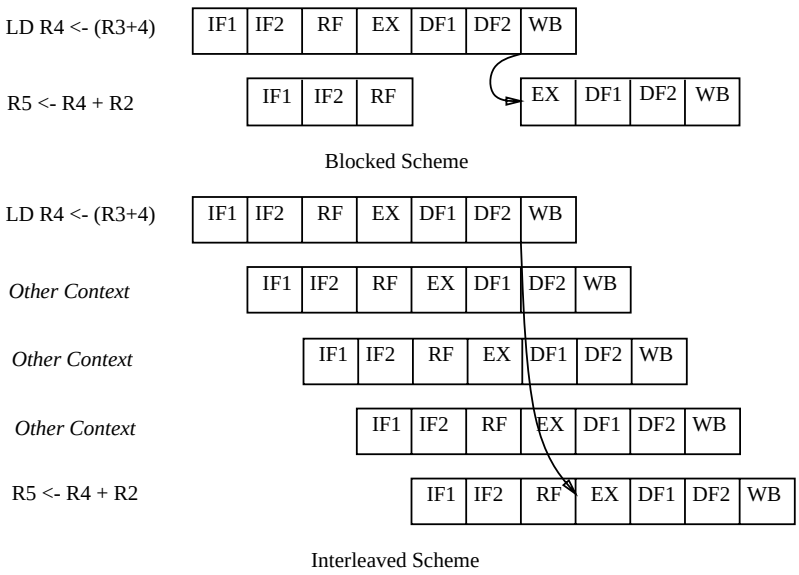


Figure 2.2: Example illustrating instruction latency tolerance by the interleaved scheme.

To further illustrate these two advantages of the interleaved scheme, we show the execution of four threads for both the blocked and interleaved schemes in Figure 2.3. The four threads are:

A Issues two instructions, with the second causing a cache miss.

B Issues one instruction, followed by a two cycle pipeline dependency, followed by two more instructions, the last of which cache misses.

C Issues four instructions, with the fourth causing a cache miss.

D Issues six instructions, with the last causing a cache miss.

The blocked scheme is shown on the upper timeline. Context A starts executing, issuing its two instructions, the second of which causes a cache miss. The pipeline must be flushed at this point before context B can execute, as shown below the timeline. Context B then executes one instruction, stalls due to the pipeline dependency, and then executes until it encounters its cache miss, at which point the pipeline is flushed and C starts executing, and so on. The interleaved scheme executing the same set of threads is shown on the lower timeline. The processor starts with all four contexts being interleaved. As we can see, this interleaving is enough to separate

the dependent instructions from Context B, completely hiding the instruction latency. The lower switch cost of the interleaved scheme is also illustrated in Figure 2.3; the switch cost associated with a cache miss is reduced from the seven cycles of the blocked scheme to two cycles for context A, three cycles for contexts B and C. Note that as contexts are made unavailable, the number of contexts being interleaved on the pipeline decreases, until we reach the point where only the single context D is being interleaved. As a result of the lower switch cost and pipeline dependency tolerance, the interleaved scheme was able to complete all four threads well before the blocked scheme.

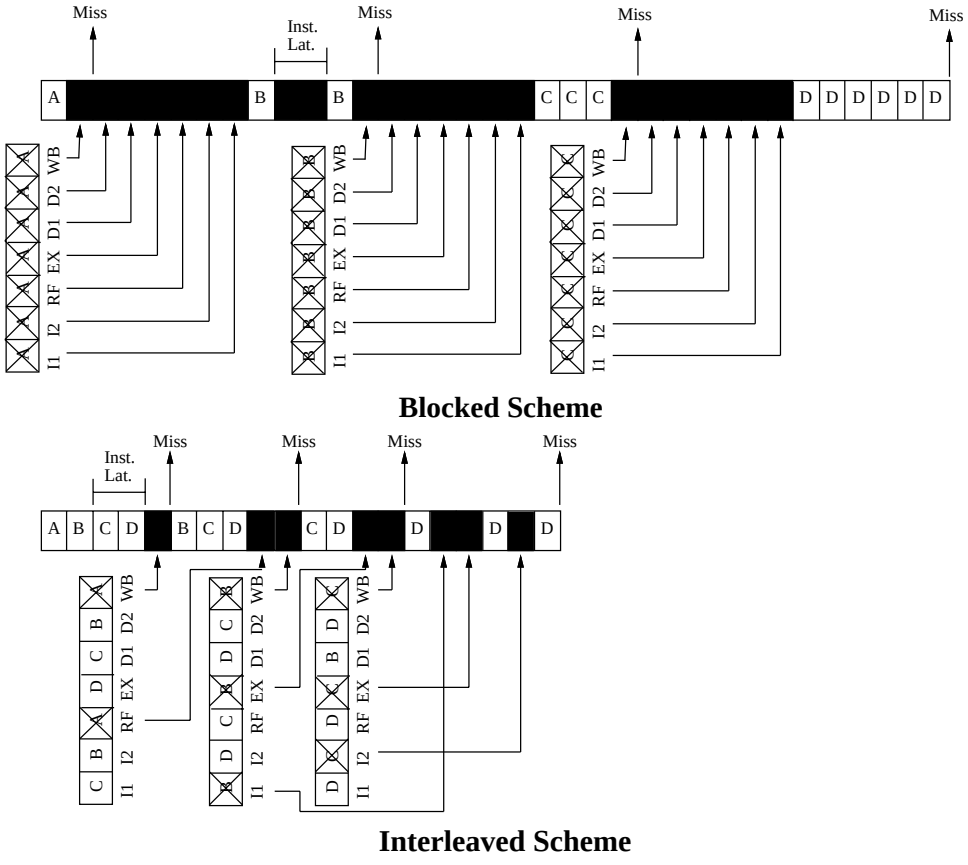


Figure 2.3: Comparison of the blocked and interleaved multiple-context schemes for a set of four threads.

2.3.2 Implementation Requirements

Now that we have outlined the performance advantages of the interleaved scheme, we will briefly discuss the hardware requirements for the two schemes to get a feel for their complexity.

The first requirement of all multiple-context processors is that the cache be capable of handling multiple outstanding memory requests. These caches are called *lockup-free*, and are more expensive than standard blocking caches. We will explore the design of these caches in detail in Chapter 5. Beyond providing a lockup-free cache, the blocked and interleaved multiple-context schemes have differing requirements, and we briefly discuss these requirements here. Chapters 6 and 7 will explore these requirements in more detail.

Blocked Scheme

In order to reduce the context switch cost to a minimum, the blocked multiple-context processor needs to replicate all the process-specific state on the processor. This state includes the program counter, the register file, and any miscellaneous process-specific state from the processor status word. In addition to this state replication, the blocked multiple context processor must provide some mechanism for causing a context switch on long-latency operations.

Memory latency can be tolerated by providing hardware mechanisms to force a context switch on each data cache miss. When this hardware detects a cache miss, any partially executed instructions in the pipeline are marked to not update any processor state, and the next context immediately starts filling the pipeline. Eventually the processor will switch back to the original context, which will reissue its load or store instruction, and hopefully the data for this instruction will now be loaded into the cache.

Synchronization latency in a blocked multiple-context processor can be tolerated by providing an *explicit context switch* instruction. With an explicit context switch instruction as the basic mechanism, a number of synchronization latency tolerance policies can be implemented [ALKK90]. Note that since the blocked processor runs one context until that context encounters a long-latency operation, providing an explicit switch (or some form of watchdog timer [WG89b]) is essential to avoid deadlock between synchronizing processes which are loaded on the same processor.

The explicit switch instruction can also be used to tolerate instruction latency on a blocked multiple-context processor. The blocked scheme can tolerate instruction latency only if the cost of the instruction latency is larger than the cost of an explicit context switch instruction. By

inserting an explicit switch instruction before an instruction which uses the result of a previous long-latency instruction, the latency of the previous instruction can be tolerated. Unfortunately, the blocked scheme does not have a mechanism for tolerating instruction latencies which are shorter than the cost of the explicit switch instruction.

Interleaved Scheme

The interleaved scheme also requires each context to have its own copy of the program counter, register file, and any miscellaneous process-specific state from the processor status word. Unlike the blocked scheme, this state is replicated not to keep the switch cost down, but rather to allow the multiple contexts to be simultaneously active.

Issuing instructions from multiple active streams requires the interleaved multiple-context processor to have a more sophisticated instruction issue unit than the blocked processor. In addition, sharing the pipeline between multiple contexts requires the pipeline to be able to identify which context an instruction was issued from to properly implement result forwarding and to allow instructions from only one context to be squashed when that context needs to be made unavailable because of a long-latency operation.

Finally, the interleaved multiple-context processor must provide some mechanism for making contexts unavailable for the duration of long-latency operations. For memory operations, this can be accomplished by providing one *transaction buffer* per context which tracks the outstanding request of that context. When a cache miss occurs, this buffer is loaded with the address of the memory operation. Any instructions in the pipeline from the context encountering the miss are squashed and a signal is sent to the instruction issue unit to prevent further instructions from this context from being generated. When the reply to the cache miss returns, the transaction buffer is marked invalid, instruction issuing from this context is enabled again, and the context restarts by issuing a repeat of its memory request.

For the interleaved scheme, policies for synchronization latency tolerance can be built on top of a *backoff* instruction. The backoff instruction has the effect of making the context unavailable for the number of cycles specified by the instruction. Issuing this instruction is equivalent to the context encountering a cache miss of latency equal to the backoff value.

The interleaved scheme actually has two mechanisms which it can employ to tolerate instruction latency. As mentioned earlier, instruction latency can be tolerated by the cycle-by-cycle context interleaving. Since instruction latencies tend to be short, the cycle-by-cycle switching of the interleaved scheme should be very effective in tolerating most of this latency. Longer

instruction latencies, that are likely to result in a stall even after interleaving, can be handled by the same backoff mechanism used to tolerate synchronization latency. This backoff can either be triggered automatically by a hardware scoreboard, in which case the backoff value will be exactly the amount needed to resolve the dependency, or it can be triggered by a backoff instruction.

2.4 Related Work

Recently, there has been an explosion of research into using an interleaved-like approach to increase the amount of instruction-level parallelism available to superscalar processors. This increase in research interest has been motivated by commercial microprocessors which are beginning to exploit instruction-level parallelism [Dig92b, JMY92]. This trend towards using more instruction-level parallelism to boost microprocessor performance is expected to continue. However, the amount of parallelism in a single stream is limited [Wal91, LW92]. As attempts are made to issue more and more instructions per cycle, the laws of diminishing returns take over, limiting performance improvements. To get around these diminishing returns, several researchers have proposed using multiple instruction streams.

Daddis and Torng [DT91] propose a dynamic superscalar architecture in which instructions from several contexts are used to fill a common instruction window. A group from the Media Research Laboratory of Matsushita have proposed an architecture aimed toward the processing element of a large-scale multiprocessor [HKN+92]. The processor consists of a number of independent instruction fetch and decode units which share several functional units. The architecture is superscalar in that multiple instructions from different threads can be issued in a single cycle; however, each thread can only issue a single instruction per cycle. Prasadh and Wu [PW91] and Keckler and Dally [KD92] both propose adding multiple contexts to a VLIW (very long instruction word) architecture. In both of these proposals, multiple threads, statically scheduled for a large number of functional units, are dynamically interleaved at runtime to improve functional unit utilization.

Unfortunately, most of the interleaved proposals for superscalar processors add multiple contexts to a processor with a very wide instruction issue in order to make the processor operate with reasonable efficiency at an expensive point in the design/complexity spectrum. Rather than making the processors extremely complex in order to get the last few percentage points of functional unit utilization, a more reasonable design would be to employ multiple simpler processors. Each processor would support more modest amounts of instruction-level parallelism, thereby making

its design much simpler. Interleaved multiple contexts could then be added to these simpler processors to tolerate memory, synchronization, and instruction latency.

2.5 Summary

In this chapter we proposed an *interleaved* multiple-context processor that combines the cycle-by-cycle switching of the fine-grained schemes with the coherent caches and full single-context support of the blocked scheme. This results in a processor with a much lower context switch cost than the blocked scheme, because the number of instructions in the pipeline that need to be squashed to tolerate a long-latency operation will often be much smaller. In addition, the cycle-by-cycle context switching helps to tolerate the latency of dependent instructions from the same context by separating them with instructions from other contexts.

We now need to quantify the performance advantages of the interleaved multiple-context processor. We have developed an event-driven simulation environment for comparing the blocked and interleaved schemes. In Chapter 3 we will examine the benefits of employing the two multiple-context schemes in a scalable shared-memory multiprocessor. Because it is important for multiple contexts to also be useful in a uniprocessor setting, Chapter 4 will explore the advantages of the interleaved scheme over the blocked scheme for a multiprogrammed, high-performance uniprocessor.

Chapter 3

Multiple Contexts: Utility for Multiprocessors

Chapter 2 qualitatively argued that a multiple-context processor built using the interleaved scheme should outperform one built using the blocked scheme. In this chapter this performance advantage is quantified for a shared-memory multiprocessor via simulation.

We start this chapter by describing our base architecture and simulation environment in Section 3.1. Event-driven simulation of the SPLASH [SWG92] application suite on a model of a shared-memory multiprocessor similar to the Stanford DASH [LLG+90] will be used to compare the blocked and interleaved schemes.

In Section 3.2, we present the results of the performance comparison between the two schemes. Multiple contexts prove to be extremely effective for three of the SPLASH applications (speeding them up by factors of 2.0 to nearly 3.5), show more modest gains for three of the applications (1.15 to 1.6 speedups), and do very little for the final application. The interleaved scheme does outperform the blocked scheme, especially for the three applications showing the largest gains from multiple contexts. For these three applications, the (geometric) mean speedup for eight contexts per processor is 2.75 for the interleaved scheme, compared to 1.94 for the blocked scheme. We end this section by exploring the characteristics of the individual SPLASH applications which determine both the general effectiveness of multiple contexts and the differences between the two schemes.

In order to ensure that our comparison of the blocked and interleaved schemes holds across a range of system assumptions, we then explore several variations to our base architecture and latency tolerance policies in Section 3.3, including varying memory latencies and cache sizes

and associativities. The interleaved scheme retains its performance advantage over the blocked across all the variations.

Section 3.4 examines the use of multiple contexts in combination with two alternative latency tolerance techniques: relaxed memory consistency models and nonblocking loads. Being able to combine well with these other latency tolerance schemes is important as the other schemes provide latency tolerance within a single thread and therefore work well when multiple contexts cannot be taken advantage of due to limited workload parallelism. Multiple contexts combines with both schemes without performance penalty. However, because multiple contexts can be used to tolerate the same latencies as relaxed memory consistency and nonblocking loads, only fairly modest performance gains are seen for combining the latency tolerance schemes over multiple contexts alone.

Interference in resources shared between the contexts, such as the caches and TLBs, is a frequently raised concern for multiple-context processors, and in Section 3.5, we show this the shared resource interference to be modest. In particular, cache interfence between the contexts is a frequently raised concern, and we show that, while multiple contexts generally increase the data cache miss rate, for larger caches this increase is small. Even with smaller caches where the miss rate does rise somewhat, much of the extra memory latency resulting from the increased misses can be tolerated by the multiple contexts.

3.1 Evaluation Methodology

Before we present our simulation results we need to first describe our evaluation methodology. We start by presenting the base processor and system architecture. We then describe the simulation environment, and finally discuss the benchmark applications and simulation configurations used in the study.

3.1.1 Base Architecture

Figure 3.1 shows the base multiprocessor architecture selected for this study. This base architecture draws heavily upon the experience gained from building the DASH multiprocessor [LLG+92, LLJ+93]. The multiprocessor consists of a number of nodes connected together by a high-bandwidth, low-latency interconnect. Each node consists of a processor, cache, and a portion of the global memory. The caches are kept coherent using a distributed, directory-based protocol similar to that of DASH [LLG+90]. Our base architecture differs from DASH in two

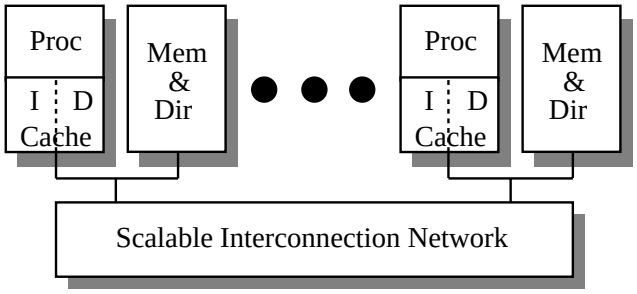


Figure 3.1: Base multiprocessor architecture.

major respects. First, the DASH cluster of four processors is changed to contain a single, high-performance processor. This modification was made in order to remove the potential memory bandwidth bottleneck of the cluster bus. As a secondary consideration, removing the DASH clustering allowed the simulation model to be simpler, which in turn allowed larger problems to be simulated. Second, DASH employs a two-level data cache hierarchy. We have selected a single-level data cache hierarchy since our base architecture contains a fairly large (64 Kbyte) writeback primary cache, and therefore most misses will be due to communication, and multi-level hierarchies do not help with communication misses. In our variational analyses where we explore primary caches smaller than 64 Kbytes, we will back the primary cache with a large secondary cache.

The processor was selected to be representative of current, high-end RISC microprocessors. It executes the MIPS II instruction set [Hei93], except that the delayed branches of the MIPS architecture have been removed. Delayed branches are an artifact of the first-generation RISC processors that do not extend well into future generations. The integer pipeline of the processor is based on the MIPS R4000 [Hei93], but is slightly more aggressive. As Figure 3.2 shows, a seven stage integer pipeline is modeled. The first two stages comprise fetching the instruction from the instruction cache. The instruction is decoded and the register operands fetched in the RF stage. In addition, the target for a branch instruction is computed in the RF phase. The ALU operation, memory virtual address calculation, or branch outcome is computed in the EX phase. The data cache access takes two cycles, DF1 and DF2. Finally, the operation is written back in the WB cycle. The R4000 has a separate Tag Check stage between DF2 and WB, which has been folded into the DF2 stage for our processor.

High floating-point performance is demanded by many scientific applications, therefore we decided to model a floating-point pipeline based on the DEC Alpha 21064 [Dig92b], which

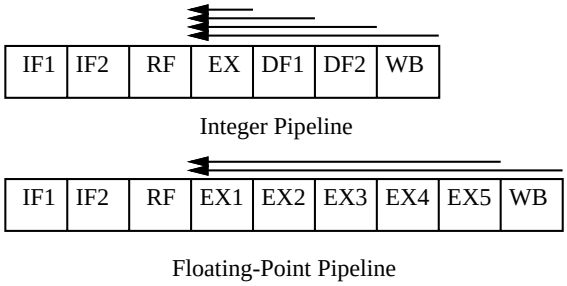


Figure 3.2: Processor pipeline.

has a more aggressive floating-point design than the R4000. The modeled floating-point pipeline replaces the EX through WB stages of the integer pipeline with five EX phases followed by a result writeback phase, resulting in a nine stage floating-point pipeline. Both pipelines forward results whenever possible to reduce operation latency. The arrows above the pipelines in Figure 3.2 denote possible result forwarding paths. The integer pipeline can forward from any pipe stage back to the execute stage. The floating-point pipeline allows forwarding only from the EX5 and WB stages to the first execute stage. The exception to this is floating-point loads, which are available at the end of EX3 (DF2), and can be forwarded from that point on.

Table 3.1: Branch target buffer parameters.

Size	2 Kbytes
Associativity	direct-mapped
Prediction Scheme	2-state
Default Prediction for BTB Miss	not taken
Misprediction Penalty	3 cycles

Because the ALU provides result bypassing, most operations have an effective latency of a single cycle. However, a small number of operations require longer to complete. As is evident from the pipeline, load operations are followed by two delay slots; the load result is not available until the end of DF2 for forwarding to the following EX phase. Branches have an even longer latency. Since the branch condition is evaluated in the EX phase, taken branches could potentially cost four cycles, however, a branch target buffer (BTB) [LS84] is used to reduce the branch penalty. The BTB can produce a predicted target in a single cycle, resulting in no penalty for a correctly predicted branch. A mispredicted branch still requires the incorrectly fetched instructions to be squashed and the BTB updated. By updating the BTB in parallel with

fetching the mispredicted branch instruction, a minimal penalty of three cycles results.[1] The characteristics of the processor's BTB are given in Table 3.1.

Table 3.2: Long-latency operations.

Operation	Issue	Latency
Integer Divide	45	45
Integer Multiply	21	21
Shift	1	2
Load	1	3
Floating-point Add/Subtract/Convert/Multiply	1	5
Floating-point Divide	61 (31)	61 (31)

The latency and issue rate of operations which take greater than a single cycle are given in Table 3.2. Most integer operations execute in a single cycle, with the exception of integer divide, multiply, and shift. While the latency of floating-point operations is five cycles, the floating-point units are fully pipelined, allowing them to issue a new operation each cycle. However, floating-point divides have a much higher latency and are not pipelined. For floating-point division, operation on single-precision data is faster than for double-precision, and the single-precision values are shown in parenthesis in the table.

3.1.2 Simulation Environment

Our simulator is built on top of Tango-Lite [GD90, DGH91], which provides an execution-driven simulation of a parallel program on a uniprocessor. The details of using Tango-Lite to simulate our target pipeline on a machine with a different pipeline are discussed in Appendix A. This section focuses on the processor and memory simulator which are coupled to Tango-Lite to provide instruction and memory latencies. Our simulator models pipeline interactions and cache effects in great detail. In contrast, the memory system is somewhat idealized in order to speed up simulation. We first describe the pipeline model and then present the memory system model.

Processor Simulation

The processor simulator models all major pipeline dependencies, including *load*, *execution result*, *execution issue*, and *control-transfer* hazards. These hazards are tracked in the simulator through

[1] Achieving this misprediction penalty requires the BTB to be dual-ported. This very aggressive BTB design was chosen to minimize the cost of branches for the single-context processor.

a scoreboard [Tho70] which maintains information on the functional unit usage and register usage of all operations in progress. Instructions are not allowed to issue until the desired functional unit is available and all register dependencies (true, anti-, and output) are satisfied.

Since application performance can be greatly affected by the compiler, it is important that the code be highly optimized to remove as many redundant operations as possible. To this end, all applications are compiled using the most recent MIPS CC and F77 compilers (version 2.1) with the -O2 level of optimization. In addition, the code needs to be properly scheduled in order to eliminate as many pipeline dependencies as possible. Since we are simulating a pipeline different from that of our MIPS R3000-based workstations, the standard MIPS code scheduling will be inadequate. Fortunately, code scheduling is performed by the MIPS assemblers, not compilers. To properly schedule the code for our pipeline, we simply replace the MIPS assembler with the Twine parameterizable scheduler and MASM assembler, developed at Stanford for the superscalar TORCH project [Smi92].

Table 3.3: Context switch costs.

Switch Cause	Blocked	Interleaved
Cache Miss	7	1–7
Explicit switch instruction	3	NA
Backoff instruction	NA	1–3

In addition to modeling pipeline dependencies, the processor simulator also handles the multiplexing of the multiple contexts on the processor. The costs to context switch are shown in Table 3.3, with the exact cost for the interleaved scheme depending on the dynamic context interleaving. The switch-spinning policy [ALKK90], where each unsuccessful synchronization operation results in a context switch, is used for tolerating synchronization latency. The blocked processor is assumed to have the ability to use the explicit switch instruction to tolerate instruction latency, and the interleaved scheme uses a backoff mechanism triggered by a hardware scoreboard to tolerate instruction latency remaining after the context interleaving.

Memory System Simulation

The memory simulator consists of a detailed model of the TLB and data cache connected to a simplified network and memory system. The TLB provides the ability for pages shared between the contexts to use a single TLB entry, and the default TLB parameters are shown in Table 3.4.

The data cache is lockup-free [Kro81], with the simulator modeling all the internal states. The default data cache parameters are given in Table 3.5. In contrast to the detailed model of the data cache, the instruction cache is modeled as ideal — all instruction accesses are assumed to be cache hits in order to speed up simulation. Modeling the instruction cache as ideal should not result in much simulation error, since the code size of all but one of the applications is less than 50 Kilobytes (PTHOR has a code size of approximately 80 Kilobytes) and an instruction cache the same size as the data cache should mainly encounter cold misses.

Table 3.4: Base TLB parameters.

Size	128 entries
Associativity	Fully Associative
Replacement Policy	Random
TLB Miss Penalty	50 cycles
Page Size	4 Kbytes

Table 3.5: Base data cache parameters.

Size	64 Kbytes
Associativity	Direct Mapped
Line Size	32 bytes
Read Occupancy	1 cycle
Write Occupancy	1 cycle
Invalidate Occupancy	2 cycles
Cache Fill Occupancy	1 cycle

Previous studies have shown that relaxed consistency complements the use of multiple-context processors [GHG+91]. In order to take advantage of this synergy, the memory system operates under release consistency [GLL+90]. The unloaded latency of memory operations are listed in Table 3.6, and are based on the memory latencies of the Stanford DASH [LLG+90]. Contention for the caches is modeled, which can increase these base latencies. While cache contention is modeled, the network and memories are modeled as contentionless to speed up simulation. Simplifying the network and memory system allows us to simulate larger problems, while still providing a sufficient model of the memory system behavior, as cache contention is likely to dominate network and memory contention [Aga92].

Table 3.6: Default memory latencies.

Hit in Primary Cache	1 cycle
Reply from Local Memory	25–45 cycles (uniform distribution)
Reply from Remote Memory	75–135 cycles (uniform distribution)
Reply from Remote Cache	96–156 cycles (uniform distribution)

3.1.3 Applications Suite

To evaluate the multiprocessor performance of the two multiple-context alternatives, we use the SPLASH applications. The SPLASH suite contains a variety of parallel scientific and engineering applications written in either C or Fortran. The applications use the Argonne National Laboratory macro package [LOB+87] for synchronization and sharing. An overview of the seven SPLASH applications is presented in Table 3.7. More information on the computational behavior and important data structures for each application can be found in Appendix B. In addition, a more detailed discussion of the applications can be found in [SWG92].

Table 3.7: SPLASH suite summary.

Application	Language	Lines	Description
Barnes-Hut	C	2700	hierarchical N-body gravitation simulation
Cholesky	C	2000	Cholesky factorization of sparse matrices
LocusRoute	C	6400	routes wires in VLSI standard cell designs
MP3D	C	1500	simulates rarefied hypersonic flow
Ocean	Fortran	3300	simulates eddy currents in an ocean basin
PTHOR	C	9200	simulates digital logic circuits
Water	C	1500	simulates water molecule interaction

Application Input Selection

Because we would like our simulation results to indicate behavior of the application on the real system, selecting realistically-sized input sets for the applications is very important. Ideally, our experiments would be performed on a multiprocessor system that allowed us to run the full-scale applications while unobtrusively gathering relevant statistics. Unfortunately, such a system does not exist even for single-context processors, and we must rely on simulation. Simulation has advantages over hardware in that the simulator can make measurements without distorting the results, and any desired statistic can be gathered, provided enough simulator detail. However,

simulation does have a big disadvantage: speed. For example, our detailed simulation results in approximately a thousandfold slowdown compared to the base hardware. In addition, since we are simulating multiple processors on a uniprocessor workstation, this slowdown must be multiplied by the number of processors being simulated.

With these sort of slowdowns, an application which executes for several minutes on the actual multiprocessor will take weeks of simulation time. This is clearly unacceptable, as we are planning to do many simulation runs across the SPLASH applications to evaluate our architectural features. Therefore we must somehow reduce the application simulation time.

Simulation time can be reduced by taking advantage of the properties of the applications being simulated. First, some applications perform essentially the same computation each time-step over a slowly-changing data set. Since the application behavior is similar for each time-step, the number of time-steps simulated can be greatly reduced. Barnes-Hut, MP3D, Ocean, PTHOR, and Water all exhibit this behavior. This is an ideal way to reduce simulation time, as the application is using its full-size input set.

Not all applications have this repetitive structure. However, some applications are intended as the inner loop of a much larger problem. Both LocusRoute and Cholesky fall in this category.[2] For these problems, the parallel application must run in a few seconds or minutes, so it is possible to simulate the entire application. Note that the entire SPLASH suite is covered by these two classes of applications.

Table 3.8: Application data sets.

Application	**Input Set**	**Iterations**
Barnes-Hut	4K particles	3 time-steps
Cholesky	BCSSTK23 (3134x3134 matrix, 24K non-zeros)	NA
LocusRoute	Primary2.grin (26K cells, 3.8K wires)	NA
MP3D	150K particles, 2.6K space cells	4 time-steps
Ocean	258x258 grid	3 time-steps
PTHOR	NTT (11.5 K elements)	20 clock cycles
Water	256 molecules	2 time-steps

By reducing the number of time steps simulated or running with a problem which would complete quickly on the real machine, we can simulate reasonable-sized inputs for the SPLASH applications running on a 16-processor machine. The applications and their input sets are given

[2]Cholesky may be used in applications with requirements ranging from factoring many smaller matrices quickly to factoring a few large matrices once. In the interests of tractable simulation, we assume operation in the former mode.

in Table 3.8. For the applications where we have reduced the number of time steps, we reset the simulation statistics after the first time step. The first step often has much different behavior than the other steps, as the caches of all processors except the initializing processor are empty. For LocusRoute, we gather statistics only during the parallel routing section of the computation. Since reading the design would be done once before evaluating multiple placements, gathering statistics only during the parallel routing section is appropriate. Statistics are also gathered only during the numeric factorization phase of Cholesky. The other sections of Cholesky could be parallelized, but this has not been done yet, so they are not included in the statistics.

3.1.4 Simulation Configurations

One of the more important decisions made when designing a multiple-context processor is selecting the number of contexts to be supported in hardware. Chapters 6 and 7 will show that a significant portion of the cost of a multiple-context processor depends directly on the number of contexts, therefore, we want to keep this number small. For our simulations, we vary this number of contexts between one and eight.

As we add more contexts per processor, the number of application processes is increased to keep all contexts busy. Because of this increasing number of application processes, lack of application parallelism and load imbalances can affect the results. To explore the potential effects of limited application parallelism and varying context workload, we simulated the SPLASH applications using the input data sets from Table 3.8 and assuming a perfect memory system (all memory accesses cost a single cycle).

Speedups for the applications from this simulation are given in Figure 3.3. Since our evaluations will be simulating a 16 processor system, and we will at most simulate eight contexts per processor, we are interested in the speedups over 16 – 128 processes. Over this range, Barnes-Hut, LocusRoute, MP3D, and Water exhibit nearly linear speedup, so lack of application parallelism should not be a factor in their multiple-context simulations. Ocean does not speedup quite as well, but there still should be enough parallelism for at least small numbers of contexts. Cholesky and PTHOR on the other hand, are already starting to saturate at 16 or 32 processes. For these two applications, the restricted amount of additional parallelism will likely limit the effectiveness of multiple contexts. However, inclusion of these two applications is interesting, as in any multiprocessor workload, several applications may be operating without much parallel slack.

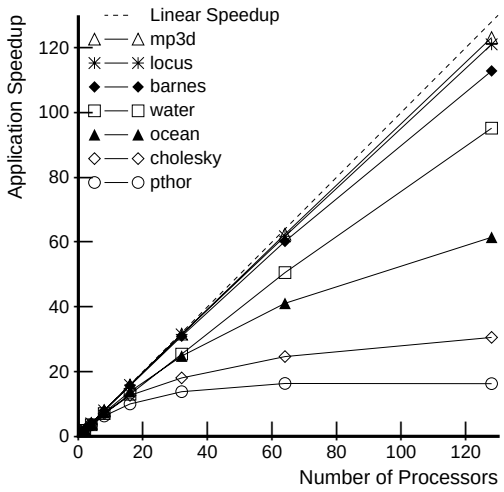


Figure 3.3: SPLASH application speedups with a perfect memory system.

3.2 Effectiveness of Multiple-Contexts

Speedups resulting from adding multiple contexts to our base processor are shown in Figure 3.4 for all seven SPLASH applications. The SPLASH applications are listed along the bottom of the graph, with two sets of bars per application. The left set of bars corresponds to the application speedup when using up to four contexts per processor, the right set up to eight contexts per processor. As we can see from the graph, the largest speedups due to multiple contexts occur for MP3D, Barnes, and Water, with the interleaved scheme showing speedups of 3.5, 2.9, and 2.1 respectively with eight contexts. The same eight-context speedups for the blocked scheme are 2.9, 2.1, and 1.2. For these three applications, blocked speedups with four contexts per processor are quite close to the eight-context speedups. As we will see in Section 3.2.2, this is due to the applications nearing the upper bound on processor utilization. Because the interleaved scheme has a higher upper bound on processor utilization, it is able to still show reasonable performance increases going from four to eight contexts per processor.

Ocean, LocusRoute, and PTHOR show more modest gains. The speedups for the interleaved scheme are 1.6, 1.3, and 1.2 respectively; for the blocked scheme 1.5, 1.1, and 1.1. For these applications, there is no performance advantage in going from four to eight contexts per processor. Of all the applications, only Cholesky shows no gains at all.

Figure 3.4 also shows that the interleaved scheme outperformed the blocked scheme for all applications, with substantial differences between the two schemes exhibited for Barnes and Water.

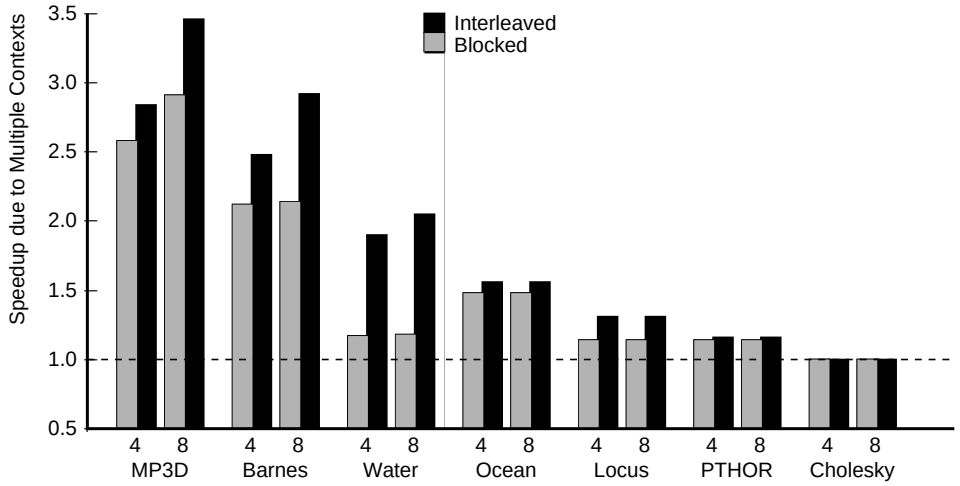


Figure 3.4: Speedups with four or eight contexts per processor for both the blocked and interleaved schemes.

In fact, with four contexts per processor, the interleaved scheme outperforms the eight-context blocked scheme for all applications except MP3D, where the performance of the four-context interleaved processor is very close to that of the eight-context blocked processor. We will explore the performance differences between the two schemes in more detail in Section 3.2.2.

3.2.1 Application Classification

To explain the differing performance of multiple contexts for the SPLASH applications we begin by examining the breakdown of the applications' single-context execution time, shown in Figure 3.5.

Figure 3.5 divides the execution time of the measured, parallel section of the applications into five categories. Starting from the bottom up, the first category, *busy*, accounts for time spent on work specified by the application. The next two categories account for time spent stalled due to *instruction* latency. Pipeline dependencies of four or fewer cycles (four being the maximum stall due to a floating point add/subtract/multiply result hazard) are labeled *short*, while all longer pipeline dependencies are labeled *long*. Time spent in the *memory* system is next, covering processor time stalled due to data cache misses. Finally, the *synchronization* category accounts for time spent in synchronization, such as waiting for a critical section or at a barrier. Figure 3.5

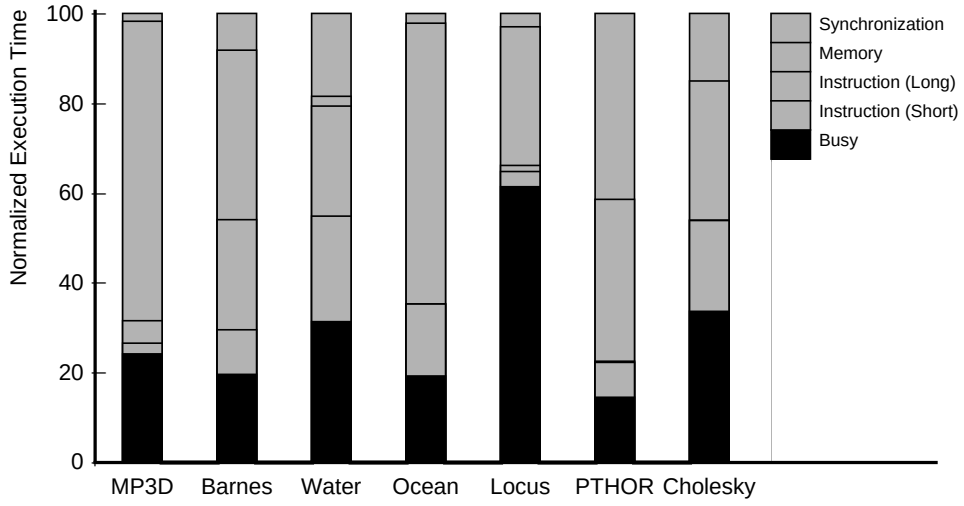


Figure 3.5: Breakdown of SPLASH execution time for the single-context processor.

shows the average of these categories for all n processes, computed using Equation 3.1.

$$Fraction\ Spent\ in\ Category\ i = \frac{\frac{1}{n}\sum_{j=1}^{n} Time\ Spent\ in\ Category\ i\ by\ Process\ j}{Total\ Measured\ Execution\ Time} \quad (3.1)$$

For almost all the applications, the single-context processor utilization (the fraction of time spent busy) is quite poor, with the processor utilization falling in the 15–35% range. The lone exception is LocusRoute, which has a much better single-context processor utilization of 60%. Memory latency is a significant contributor to this poor processor utilization for all applications except Water, reinforcing the importance of multiple contexts being able to tolerate memory latency as efficiently as possible. Figure 3.5 also substantiates the importance of multiple contexts being able to tolerate synchronization and instruction latency. Instruction latency is small for most applications, however, for both Barnes and Water, the processor spends more time stalled due to pipeline dependencies than it does doing actual work. Synchronization overhead also tends to be small for most applications, with the exception of PTHOR, where synchronization latency accounts for over 40% of the measured execution time.

The general effectiveness of multiple contexts for a specific application will depend on both the single-context processor utilization and the availability of extra concurrency. Single-context performance determines the maximum application speedup — an application with lower single-context performance has more room to benefit from latency tolerance. The amount of extra

application parallelism is also important for multiple-context performance, since it is this extra parallelism which will be used to tolerate the long-latency operations. The single-context performance and latency distribution from Figure 3.5 and ideal speedups from Figure 3.3 are summarized in Table 3.9. The table also includes latency distributions for use in contrasting the performance of the two different multiple context schemes in the following section.

Table 3.9: Application characteristics important for determining multiple-context effectiveness.

			Latency Distribution			
Application Name	**Excess Concurrency**	**Single-context Performance**	**Sync.**	**Mem.**	**Inst. (Long)**	**Inst. (Short)**
MP3D	High	Low	2%	88%	7%	3%
Barnes	High	Low	10%	47%	31%	12%
Water	High	Low	27%	3%	36%	34%
LocusRoute	High	Med/High	8%	80%	4%	9%
Ocean	Med	Med	3%	77%	0%	20%
PTHOR	Low	Low	48%	42%	0%	9%
Cholesky	Low	Low	23%	47%	0%	30%

Based on the first two columns of Table 3.9, we can divide the SPLASH suite into three categories:

1. Extra application parallelism, poor single-context performance: MP3D, Barnes, and Water.
2. Extra application parallelism, good single-context performance: LocusRoute.
3. Limited application parallelism: Ocean, PTHOR, and Cholesky.

The first group consists of applications which have substantial amounts of latency and large amounts of parallelism which multiple contexts can exploit in order to tolerate that latency. This is the ideal situation for multiple contexts, and not surprisingly these applications showed the largest speedups for multiple contexts. The second group contains applications which have large amounts of parallelism available, but are already achieving good single-context performance. Multiple contexts can help to remove the remaining latency, but since this latency is small the performance gains due to multiple contexts will also be small. Of the SPLASH applications, LocusRoute is the sole example of this type of application.

Finally, the last group consists of applications which have limited amounts of extra parallelism. These applications pose a problem, since multiple contexts relies on extra application parallelism for its latency tolerance. Ocean, PTHOR, and Cholesky fall in this group. Ocean has the largest

additional parallelism of this group, and is able to get reasonable speedups with up to four contexts per processor until the extra overhead resulting from dividing the problem more and more finely outweighs the gains from the latency tolerance of the additional contexts. PTHOR and Cholesky have much more limited amounts of additional parallelism, and not surprisingly, exhibited the lowest speedups. For Ocean and Cholesky, the lack of application parallelism is due mainly to the small problem sizes we had to chose to keep our simulation time down. With larger input sets, multiple contexts should be able to show larger gains for these applications. The parallelism in PTHOR, on the other hand, is limited by the input circuit's topology, and most circuits simply do not provide enough parallelism for large numbers of threads [Sou92].

3.2.2 Comparison of Interleaved and Blocked Schemes

Looking now at the differences in performance between the two schemes, we would expect this difference to be fairly small for applications in the second and third categories, since multiple contexts are only going to be modestly effective for these applications. However, multiple contexts should be very effective for the first category, and for these applications the lower switch cost of the interleaved scheme should allow it to significantly outperform the blocked scheme. In addition, the difference between the two schemes should be largest for applications with significant amounts of short instruction latencies, since these cannot be tolerated by the blocked scheme. Table 3.9 shows Water to spent a large percentage of its time in short instruction latencies and Water does indeed exhibit the largest performance differences between the two schemes.

To further examine the performance difference between the two schemes and to verify that our classification is explaining the differing application speedups, a breakdown of the multiple-context execution time is shown for the blocked scheme in Figure 3.6, for the interleaved scheme in Figure 3.7. In these graphs, execution time of the measured portion of the application is presented for one, two, four, and eight contexts per processor, normalized to the single-context time. This execution time is now divided into six categories, with the new category, *context switch*, accounting for time spent in switching overhead.

Looking first at applications in the third category, we observe the effects of the limited parallelism of Ocean, PTHOR, and Cholesky. For PTHOR and Cholesky, time spent in synchronization actually increases with larger number of contexts even though the multiple contexts are tolerating some of this synchronization latency. In addition, for all three applications, the total amount of application-specified work actually increases with the number of processes. For Cholesky the amount of work depends on the distribution of the supernodes and this changes with increasing

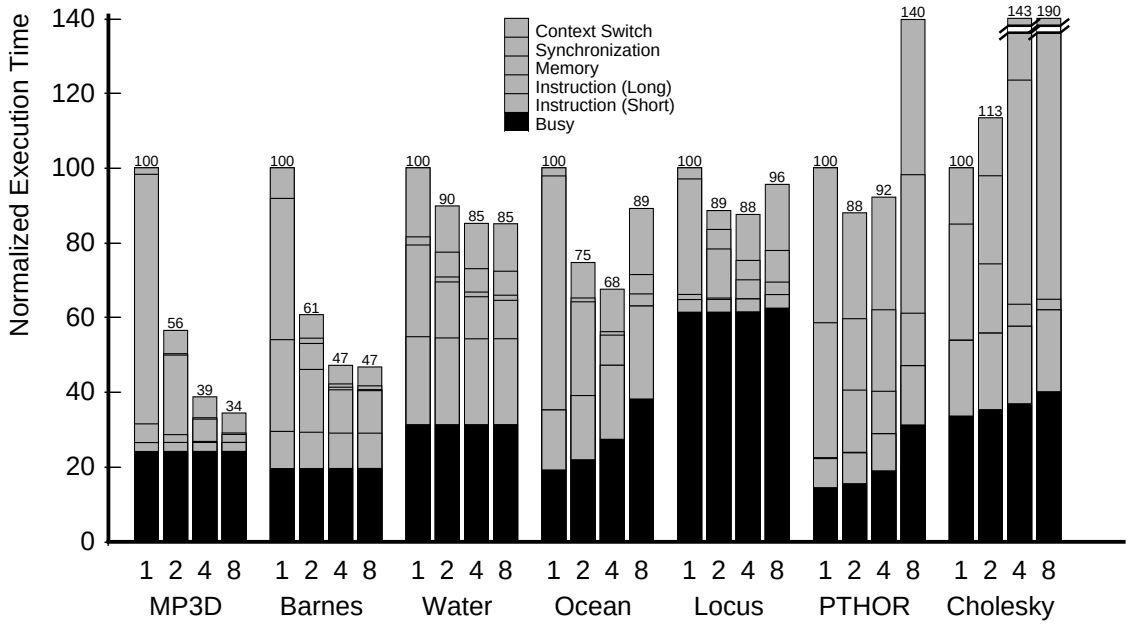


Figure 3.6: Application execution time breakdown for the blocked scheme.

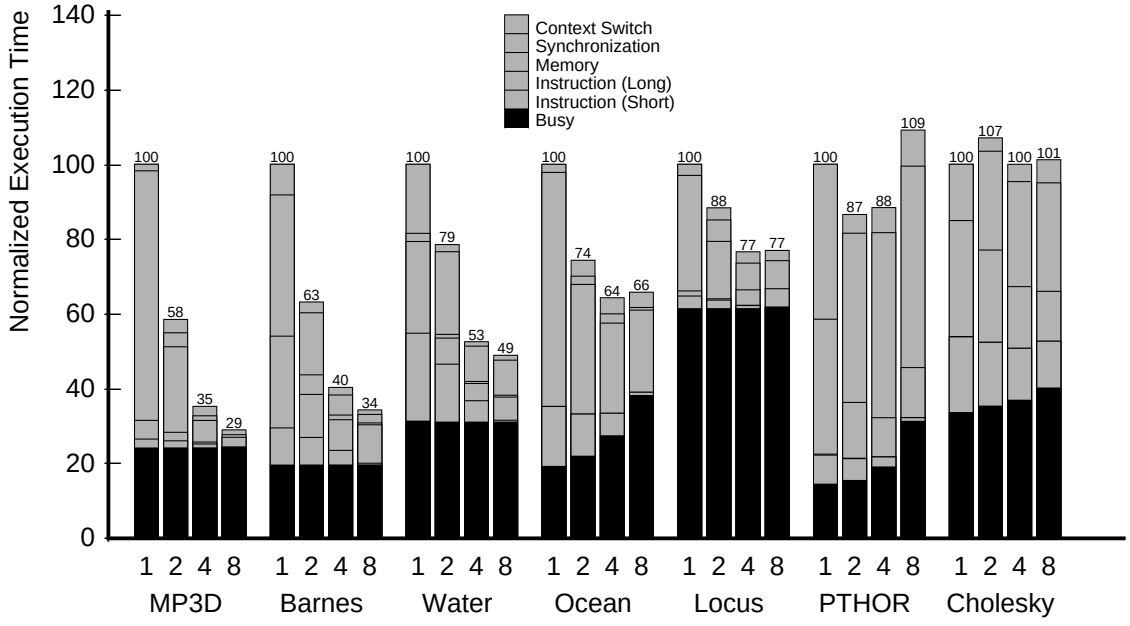


Figure 3.7: Application execution time breakdown for the interleaved scheme.

processes. Ocean statically divides its grids among the processes, and extra work is done for boundary elements. As the number of processes increases, the fraction of elements that are on the boundary also increases. PTHOR uses a set of distributed task queues, and more time is spent maintaining these queues as the number of processes increases. Not only does this extra work remove some of the gains due to multiple contexts, but some of these extra instructions are themselves memory references. Table 3.10 lists the total number of memory reads generated by the application as the number of contexts is increased. We can see that for the three applications the number of read references does increase, including a dramatic increase for Ocean and PTHOR. Since these extra memory references will result in extra cache misses, the amount of latency that must be tolerated is further increased. This combination of extra instructions executed and increased synchronization and memory latency prevents these applications from getting large speedups.

Table 3.10: Total number of reads per application (in millions).

Contexts	MP3D	Barnes	Water	Ocean	Locus	PTHOR	Cholesky
One	15 M	49 M	22 M	60 M	121 M	61 M	93 M
Two	15 M	49 M	22 M	71 M	121 M	66 M	98 M
Four	15 M	49 M	22 M	93 M	121 M	82 M	103 M
Eight	15 M	49 M	22 M	135 M	124 M	140 M	112 M

Turning next to LocusRoute, the sole application in the second category, we see that while there is very little latency to be tolerated, multiple contexts can effectively use LocusRoute's additional parallelism to tolerate this remaining latency. For both schemes, almost all latency is removed using four contexts per processor, and increasing to eight contexts per processor actually reduces performance due to increased synchronization overhead and interference between the contexts. We also see that the interleaved scheme outperforms the blocked for LocusRoute due to its much lower switch cost and ability to tolerate short pipeline stalls. However, because there was little latency to begin with, these advantages do not result in a large performance difference between the two schemes.

Finally, MP3D, Barnes, and Water fall into the first category. To examine these three applications in more detail, we will show the same data from Figure 3.6 and Figure 3.7 in a different form: processor utilization graphs. Rather than normalizing versus the single-context results, processor utilization graphs show the percent of time spent in each category in a self-normalized

fashion. This allows us to more easily observe trends, however, we must be careful in interpreting these graphs. Even when the absolute time spent on a given category decreases as we add more contexts per processor, the percent of time spent on that category may actually increase (due to decreasing at a slower rate than other categories). Figures 3.6 and 3.7 can be used to determine that the absolute time decreased while the percentage of time devoted to this category increased. Finally, note that the amount of real work done by all of these applications remains constant as more processes are added, so an increased processor utilization translates directly into an increased application speedup.

Examining the application latency distributions in Table 3.9, we see that instruction latency plays a significant role for Barnes and Water, but is not as critical to the performance of MP3D. Thus, the performance advantage of the interleaved scheme for MP3D will be mainly due to the lower switch overhead, while for Barnes and Water it will be a combination of the two factors. We will examine MP3D first, concentrating on the effects of the lower switch overhead of the interleaved scheme. We then turn to Barnes and Water to examine the differences in instruction latency tolerance between the two schemes.

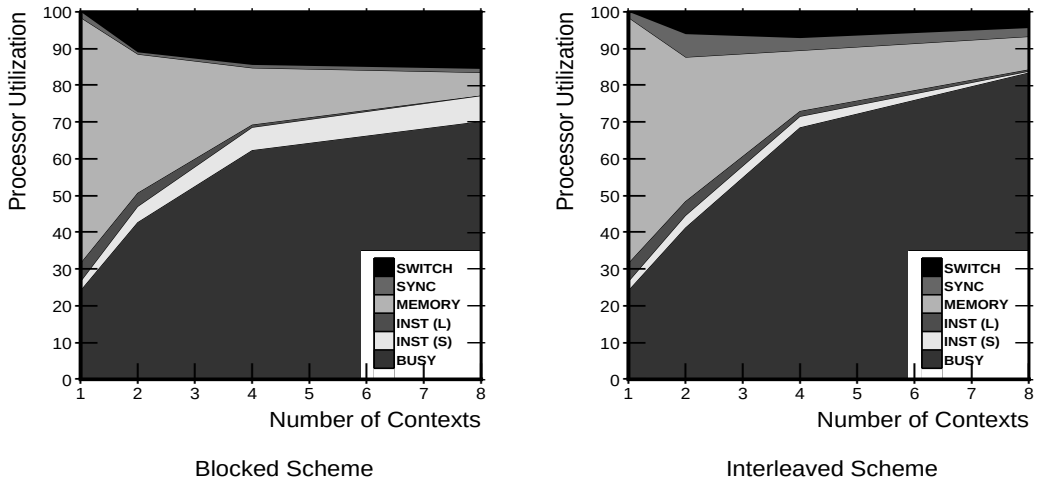


Figure 3.8: Processor utilization for MP3D.

Processor utilization for both the blocked and interleaved schemes running MP3D is shown in Figure 3.8. The single-context processor spends only about a quarter of its time busy, with the rest of the time primarily being lost in the memory system. Both schemes are able to tolerate this large memory latency of MP3D, however, the interleaved scheme is able to do so at a much lower switch cost. This allows the interleaved scheme to achieve a higher processor utilization

then the blocked scheme. At eight contexts per processor, the blocked scheme is able to reach a 70% processor utilization, while for the interleaved scheme the utilization is over 80%.

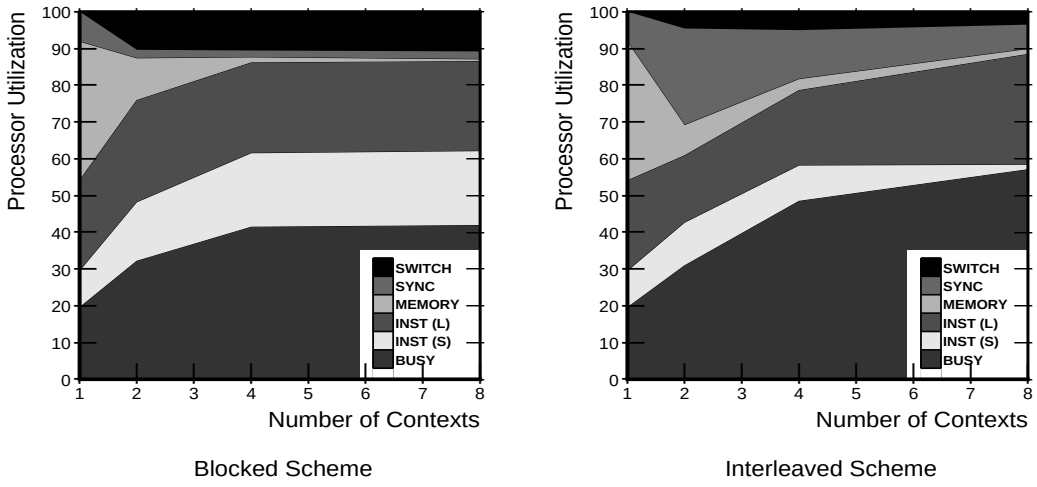


Figure 3.9: Processor utilization for Barnes.

Instruction latency is a significant fraction of the total latency for Barnes and Water. Examining the processor utilization graphs for Barnes and Water in Figures 3.9 and 3.10 respectively, we observe that both schemes are tolerating some of the long instruction latency, with the interleaved scheme tolerating slightly more than the blocked scheme. Barnes and Water have large amounts of longer instruction latency, due primarily to floating-point divides, which take 61 cycles on our pipeline. The long latency of these divides is handled by context switching for the blocked scheme and by backing-off for the interleaved scheme. Since the floating-point divide unit is not pipelined, these latency tolerance techniques will be successful only if floating-point divides are spread apart far enough to avoid saturating the divide unit. We can see this saturation occurring in Figures 3.6 and 3.7. For both Barnes and Water, the amount of time spent in floating-point divide stalls actually increases slightly for the interleaved scheme and remains constant for the blocked scheme when going from four to eight contexts per processor due to the divide unit being saturated.

Even though the blocked scheme is able to tolerate some longer pipeline stalls, it may do so fairly inefficiently due to the larger cost of explicit context switches compared to scoreboard-initiated backoff. This is indeed the case with Water as is apparent from the processor utilization graphs shown in Figure 3.10. For the blocked scheme, switching on instruction latency comes at the cost of a large context switch overhead, and this large switch cost, coupled with the short

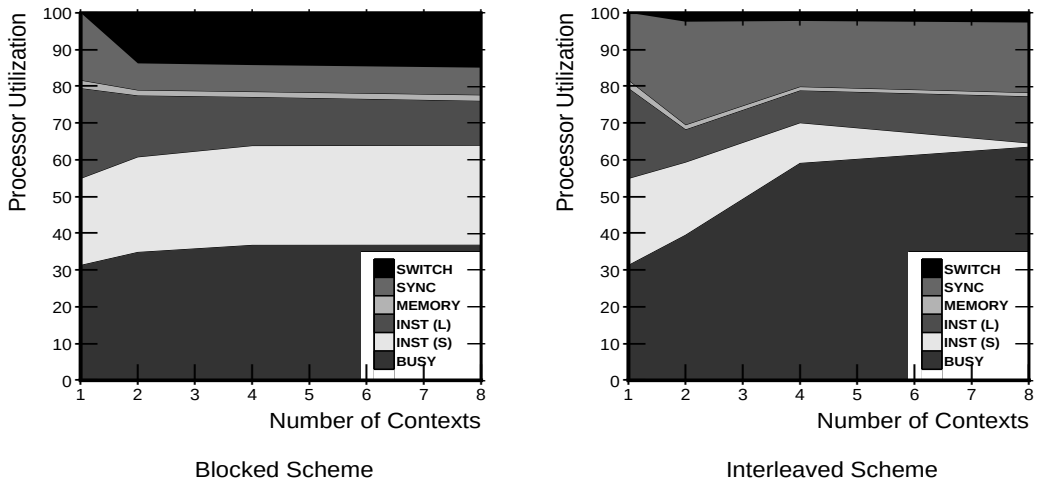


Figure 3.10: Processor utilization for Water.

latencies which cannot be tolerated, results in a small processor utilization increase, going from roughly 30% for the single-context case to under 40% with eight contexts. For the interleaved scheme, the cycle-by-cycle interleaving along with the lower switch cost for backoff instructions allow the pipeline latency to be hidden much more effectively, and a processor utilization of over 60% with eight contexts per processor results.

3.2.3 Summary

This section has shown that the interleaved scheme does indeed benefit from a lower context switch overhead and tolerance of short pipeline stalls. The performance advantage of the interleaved scheme varies depending on the application, and can be substantial for some applications. Table 3.11 summarizes the multiple-context speedups for all seven SPLASH applications. In the following section, we perform some variational analyses to confirm that these advantages of the interleaved scheme hold across a broad range of multiprocessor environments.

Table 3.11: Summary of application speedups due to multiple contexts.

	Scheme	MP3D	Barnes	Water	Ocean	Locus	Pthor	Cholesky	Mean
Four Contexts	Interleaved	2.84	2.48	1.90	1.56	1.31	1.16	1.00	1.64
	Blocked	2.58	2.12	1.17	1.48	1.14	1.14	1.00	1.43
Eight Contexts	Interleaved	3.46	2.92	2.05	1.56	1.31	1.16	1.00	1.74
	Blocked	2.91	2.14	1.18	1.48	1.14	1.14	1.00	1.46

3.3 Variational Analyses

We start our variational analyses by examining the importance of providing mechanisms allowing the multiple-context processor to tolerate synchronization and instruction latency. We then examine the effects of different memory latencies, cache sizes, and cache associativities, as would be found across a range of multiprocessors.

3.3.1 Synchronization Latency Tolerance Mechanisms

The numbers presented for the interleaved scheme in Section 3.2 assume that the interleaved scheme provides a backoff instruction upon which a switch spinning policy can be implemented. The design of the interleaved multiple-context processor could be simplified somewhat by not providing a backoff instruction and instead letting a context spinning on synchronization simply waste processor cycles. However, relying solely on spinning does have a significant performance penalty as shown in Table 3.12. The table lists the maximum speedup under a simple spinning policy, under a switch-spinning policy using a fixed backoff value of 150 cycles, and under a switch-spinning policy using an application-specific backoff value (shown in Table 3.13). In addition to the maximum speedups, the table shows in parenthesis the number of contexts required to achieve that speedup.

Table 3.12: Performance of different synchronization tolerance policies for the interleaved multiple context processor.

Policy	MP3D	Barnes	Water	Ocean	Locus	PTHOR	Cholesky	Mean
Spinning	2.72 (8)	2.51 (8)	1.88 (8)	1.51 (4)	1.10 (2)	1.02 (2)	1.00 (1)	1.55
Switch spinning	3.44 (8)	2.92 (8)	2.05 (8)	1.56 (4)	1.31 (4)	1.14 (2)	1.00 (1)	1.74
- tuned	3.46 (8)	2.92 (8)	2.05 (8)	1.56 (4)	1.31 (4)	1.16 (2)	1.00 (1)	1.74

Table 3.12 shows that switch spinning is very effective for the interleaved scheme, even though both switch spinning algorithms employed were quite primitive, using a fixed backoff value for all synchronization events within a given application. The tuned values were selected by trial and error, however since the applications did not benefit much from tuning, any fixed value roughly equal to a remote memory latency would likely work well. The performance of the switch spinning policy could have been better if the backoff scheme were tailored to the specific type and/or instance of each synchronization event. However, even with our simple approach, the mean performance of the applications improved by 11% by employing switch spinning. As

Chapter 7 will show, the extra complexity required to implement the backoff instruction is small enough to be justified by this large performance improvement.

Table 3.13: Backoff values (in processor cycles) employed for the tuned switch spinning policy.

Barnes	Cholesky	Locus	MP3D	Ocean	PTHOR	Water
100	500	150	300	50	50	150

3.3.2 Instruction Latency Tolerance Mechanisms

The results in Section 3.2 assumed that the blocked processor switched contexts to tolerate longer instruction latencies, and the interleaved processor issued a scoreboard-triggered backoff for all untolerated instruction latencies it encountered. We now examine the performance advantages of being able to tolerate these instruction latencies by comparing our base blocked processor to one which is unable to use its explicit switch instruction to tolerate pipeline dependencies and our base interleaved processor to one which is unable to use backoff to hide instruction latencies not tolerated by the context interleaving. In addition, for the blocked scheme we also examine a processor with a slower explicit switch cost than the base (seven cycles, the same as for the switch on a cache miss.)

Table 3.14 shows the application speedups for the processors with the better instruction latency tolerance abilities over their more limited counterparts. The interleaved scheme exhibited a 9% mean improvement in performance, while for the blocked scheme the mean of the execution time improvement was 7% for the fast switch, and 6% for the slower switch. As we will show in Chapter 6, providing a fast switch results in a fair amount of additional complexity, and our numbers here show that the performance to be gained by this fast switch are very small.

For four of the applications, no improvement or a slight decrease in performance resulted from switching contexts to tolerate instruction latency. The slight performance degradation seen in some instances occurred because instruction latency tolerance used up application parallelism which would have been better employed to tolerate the longer memory or synchronization latencies present. Barnes, Water, and MP3D, on the other hand, showed significant gains due to tolerating instruction latency. Because the context switch and backoff instructions are required to effectively deal with synchronization latency, the mechanism for instruction latency tolerance will already be included in the multiple-context processor, and the performance improvements from utilizing this mechanism are large enough to warrant an investigation of its additional implementation costs in

Chapters 6 and 7.

Table 3.14: Speedup due to tolerating longer instruction latency.

Scheme	MP3D	Barnes	Water	Ocean	Locus	PTHOR	Cholesky	Mean
Blocked - Fast	1.14 (8)	1.25 (8)	1.17 (8)	1.00 (4)	0.96 (4)	0.98 (2)	1.00 (1)	1.07
Blocked - Slow	1.11 (8)	1.24 (4)	1.16 (4)	1.00 (4)	1.01 (4)	0.96 (2)	1.00 (1)	1.06
Interleaved	1.19 (8)	1.23 (8)	1.33 (8)	1.00 (4)	0.99 (4)	0.97 (2)	0.95 (1)	1.09

3.3.3 Effects of Memory Latency

To examine the effectiveness of multiple contexts in a broad range of multiprocessors, we varied the memory latency to be one-half, twice, and four times as the base simulator latencies. Results for six of the seven SPLASH applications are given in Table 3.15. Simulating Ocean for the longer memory latencies required more memory than our workstations had available, so we have excluded it from the table. For the different memory latencies, numbers are normalized to the single-context execution times with that memory latency. Again, the number of contexts required to gain the listed speedup are given in parenthesis.

Table 3.15: Effects of memory latency.

Latency	Scheme	MP3D	Barnes	Water	Locus	PTHOR	Cholesky	Mean
$\frac{1}{2}$x	Interleaved	2.80 (8)	2.69 (8)	2.04 (8)	1.28 (4)	1.18 (2)	1.02 (4)	1.69
base	Blocked	2.55 (8)	1.96 (8)	1.18 (8)	1.23 (4)	1.17 (2)	1.00 (1)	1.43
1x	Interleaved	3.46 (8)	2.92 (8)	2.05 (8)	1.31 (4)	1.16 (2)	1.00 (1)	1.78
base	Blocked	2.91 (8)	2.14 (8)	1.18 (8)	1.14 (4)	1.14 (2)	1.00 (1)	1.46
2x	Interleaved	3.55 (8)	3.32 (8)	2.03 (8)	1.44 (4)	1.29 (4)	1.00 (1)	1.88
base	Blocked	3.57 (8)	2.49 (8)	1.18 (8)	1.34 (4)	1.21 (4)	1.00 (1)	1.60
4x	Interleaved	5.04 (8)	3.93 (8)	2.13 (8)	1.56 (4)	1.35 (4)	1.00 (1)	2.11
base	Blocked	4.56 (8)	3.09 (8)	1.26 (8)	1.37 (4)	1.29 (4)	1.00 (1)	1.78

The performance difference between the blocked and interleaved schemes remains roughly constant around 20% with increasing memory latency. Even at very long memory latencies, the applications still have significant segments of execution where the memory latency can be completely tolerated by the contexts, and during these sections the interleaved scheme still benefits from the lower switch cost and instruction latency tolerance.

3.3.4 Effects of Differing Cache Organizations

Finally, to examine the effects of differing cache sizes on multiple context performance, we gathered results for 4, 16, 64, and 256 Kbyte primary data caches. For the 4 Kbyte and 16 Kbyte caches, the simulations included a 1 Mbyte secondary cache. Speedups with the differing cache sizes are listed in Table 3.16. We were unable to run Ocean for cache sizes other than 64 Kbytes due to the memory limitations of our workstations. Multiple-context speedups were fairly constant across the different cache sizes, however there is a trend for speedups to increase slightly from the smaller to larger cache sizes. The slightly lower speedup for small caches is due to their susceptibility to interference between the contexts as will be shown in Section 3.5.3.

Table 3.16: Multiple-context speedup for differing cache sizes (all caches direct-mapped).

Size	Scheme	MP3D	Barnes	Water	Locus	PTHOR	Cholesky	Mean
4 KB	Blocked	2.24 (8)	1.97 (8)	1.15 (4)	1.06 (2)	1.02 (2)	1.00 (1)	1.33
	Interleaved	3.08 (8)	2.94 (8)	1.99 (8)	1.23 (4)	1.00 (2)	1.21 (4)	1.73
16 KB	Blocked	2.64 (8)	2.12 (8)	1.14 (4)	1.10 (2)	1.09 (2)	1.00 (2)	1.40
	Interleaved	3.42 (8)	2.96 (8)	1.97 (8)	1.26 (4)	1.04 (2)	1.24 (4)	1.79
64 KB	Blocked	2.91 (8)	2.14 (8)	1.18 (8)	1.14 (4)	1.14 (2)	1.00 (1)	1.46
	Interleaved	3.46 (8)	2.92 (8)	2.05 (8)	1.31 (4)	1.16 (2)	1.00 (1)	1.78
256 KB	Blocked	2.96 (8)	2.02 (8)	1.17 (8)	1.18 (4)	1.11 (2)	1.00 (1)	1.45
	Interleaved	3.54 (8)	2.80 (8)	2.03 (8)	1.26 (4)	1.13 (2)	1.18 (4)	1.80

To examine the effects of cache associativity on multiple-context speedups, we simulated a direct-mapped cache, a direct-mapped cache combined with a 16-entry victim cache [Jou90], and two and four-way set associative caches. The size of the caches was held constant at 64 KBytes. Again, the multiple-context speedups were fairly constant across the different cache associativities, with a slight trend towards better performance with greater cache associativity for the interleaved scheme. As with increasing the cache size, greater cache associativity reduces the susceptibility to interference between the contexts.

3.4 Combining with Other Latency Tolerance Mechanisms

Because the other latency tolerance mechanisms (prefetch, relaxed memory consistency, and nonblocking loads) exploit parallelism within a single thread, they can be used to tolerate latency when not enough parallelism exists for multiple contexts. Therefore it is important that multiple contexts be able to combine with these other approaches without a performance loss. In this

Table 3.17: Multiple-context speedup for differing cache associativities (all caches 64 KB).

Assoc.	Scheme	MP3D	Barnes	Water	Locus	PTHOR	Cholesky	Mean
Direct Mapped	Blocked	2.91 (8)	2.14 (8)	1.18 (8)	1.14 (4)	1.14 (2)	1.00 (1)	1.46
	Interleaved	3.46 (8)	2.92 (8)	2.05 (8)	1.31 (4)	1.16 (2)	1.00 (1)	1.78
Sixteen Victim	Blocked	2.93 (8)	2.07 (8)	1.17 (8)	1.23 (4)	1.18 (2)	1.00 (1)	1.48
	Interleaved	3.56 (8)	2.87 (8)	2.03 (8)	1.35 (4)	1.20 (4)	1.02 (4)	1.80
Two Set	Blocked	2.94 (8)	2.04 (8)	1.18 (8)	1.18 (4)	1.17 (2)	1.00 (1)	1.46
	Interleaved	3.52 (8)	2.81 (8)	2.03 (8)	1.31 (4)	1.22 (4)	1.15 (4)	1.82
Four Set	Blocked	2.96 (8)	2.00 (8)	1.18 (8)	1.19 (4)	1.16 (2)	1.00 (1)	1.46
	Interleaved	3.52 (8)	2.77 (8)	2.03 (8)	1.33 (4)	1.19 (4)	1.21 (4)	1.83

section, we examine combining multiple contexts with two of the three other major latency tolerance techniques, relaxed memory consistency models and nonblocking loads. We would have liked to also evaluate combining multiple contexts with prefetching, unfortunately, the compiler we are using does not support prefetching. Compiler-based prefetching has been shown to perform well on some applications that cause difficulty for multiple contexts [Mow94], while multiple contexts provides gains on some applications where prefetching shows little gains. Combining multiple contexts with prefetching will not be simple, as the use of multiple contexts will cause additional cache interference, which will complicate the prediction of which references will be cache misses done by the compiler or programmer. Despite this concern, there does seem to be promise for combining the two schemes [GHG+91], and a detailed study of this combination should provide interesting results.

3.4.1 Multiple Contexts and Release Consistency

All our previous results have been for applications running under release consistency [GLL+90]. To examine the benefits of combining multiple contexts with a relaxed memory consistency model, we now compare the performance of our release-consistent multiple-context processors to multiple-context processors employing sequential consistency [Lam79]. Before doing this, we first show the performance gains of release consistency over sequential consistency for the base single-context processor in Table 3.18. As in previous studies [GGH91, GHG+91, GGH92], our study shows release consistency to provide substantial gains over sequential consistency for the SPLASH suite, resulting in a 34% mean speedup in execution time.

Table 3.19 lists the benefits of combining relaxed memory consistency with multiple contexts. All speedups are relative to the single-context sequentially-consistent case. Since multiple

Table 3.18: Speedup due to release consistency for the single-context processor.

MP3D	Barnes	Water	Ocean	Locus	PTHOR	Cholesky	Mean
1.57	1.57	1.03	1.54	1.31	1.22	1.27	1.34

contexts are capable of tolerating some of the same write latency that can be tolerated by release consistency, adding release consistency results in only a 12% and 7% execution time speedup for the interleaved scheme and blocked scheme respectively. The larger gain occurs for the interleaved scheme because it is tolerating a larger number of long-latency operations and benefits more from relaxed consistency reducing the number of operations which result in a context switch.

Table 3.19: Combining release consistency with multiple contexts.

Scheme	MP3D	Barnes	Water	Ocean	Locus	PTHOR	Cholesky	Mean
Blocked, SC	3.99 (8)	3.29 (8)	1.21 (4)	1.93 (2)	1.58 (4)	1.32 (2)	1.06 (2)	1.83
Blocked, RC	4.57 (8)	3.37 (8)	1.21 (8)	2.28 (4)	1.49 (4)	1.38 (2)	1.27 (1)	1.96
Interleaved, SC	3.91 (8)	4.36 (8)	2.07 (8)	1.66 (2)	1.70 (4)	1.25 (4)	1.12 (4)	2.02
Interleaved, RC	5.43 (8)	4.59 (8)	2.10 (8)	2.39 (4)	1.71 (4)	1.40 (2)	1.27 (1)	2.34

Release consistency improves the performance of the single-context processor by such a substantial margin that future microprocessors are likely to incorporate support for some form of relaxed memory consistency. This implies that multiple-context processors will be supporting relaxed consistency. Our study reinforces a previous study [GHG+91] showing relaxed consistency to combine well with multiple contexts, providing better performance than either relaxed consistency or multiple contexts alone. By combining relaxed consistency and multiple contexts, the processor may be able to reach a desired performance target with a smaller number of contexts.

3.4.2 Multiple Contexts and Nonblocking Loads

We now turn to exploring the benefits arising from combining multiple contexts with nonblocking loads. Nonblocking loads tolerate memory latency by allowing a processor to continue after a cache miss until the processor actually needs the result of the load. A recent paper by Boothe and Ranade [BR92] investigated the advantages of combining nonblocking loads with multiple contexts for a system without caches, thereby allowing the processor to switch on the use of a

register rather than on each load. By grouping together loads, they found substantial gains for *switch-on-use* over *switch-on-load*. These large gains arise because the compiler can easily group several loads together, thereby pipelining the memory references. They also investigated adding caches to their switch-on-use scheme, and found this resulted in an additional performance gain by a factor at least as large as going from switch-on-load to switch-on-use. However, the one comparison not performed by Boothe and Ranade was to compare the advantages of switch-on-use versus switch-on-load for a system with coherent caches. We would expect the advantages of switching on the data use to be much smaller for this case, since multiple *misses* must be grouped together to pipeline memory accesses. We have investigated this further by adding nonblocking loads to our cache-coherent, multiple-context multiprocessor.

We developed a primitive compiler post-processor for nonblocking loads similar to that of Boothe and Ranade, which does dependency analysis within a basic block and rearranges code to group loads together. Our post-processor groups loads within each basic block by moving them up as far as their dependencies will allow. We then simulated this rearranged code on both a blocked and interleaved processor with nonblocking loads. For these multiple-context processors with nonblocking loads, the cost to switch contexts (blocked) or maximum cost of making a context unavailable (interleaved) was assumed to be three cycles, as the determination of whether a register has been loaded with its data occurs earlier in the pipeline than determining if a cache miss has occurred.

The benefits of adding nonblocking loads to the blocked scheme are shown in Figure 3.11. All execution times are normalized to the single-context case with standard blocking loads, therefore the single-context bars show the advantages of adding nonblocking loads to a conventional processor. By comparing with Figure 3.6, we see that the largest performance gains due to nonblocking loads occur for the single-context case, with the difference between blocking and nonblocking loads diminishing as more contexts are added.

The interleaved scheme, shown in Figure 3.12, exhibits similar behavior, however, the advantage due to nonblocking loads does not diminish as much with increasing contexts. The interleaved scheme encounters more untolerated memory latency than the blocked scheme, and therefore shows larger gains from the nonblocking loads tolerating some of this memory latency.

The advantages of adding nonblocking loads are summarized in Table 3.20. In contrast with the large gains shown for systems without caches, the advantages of adding nonblocking loads and switch-on-use to a system with caches is much smaller. Of course, the benefits from combining multiple contexts and nonblocking loads presented here could be increased with help

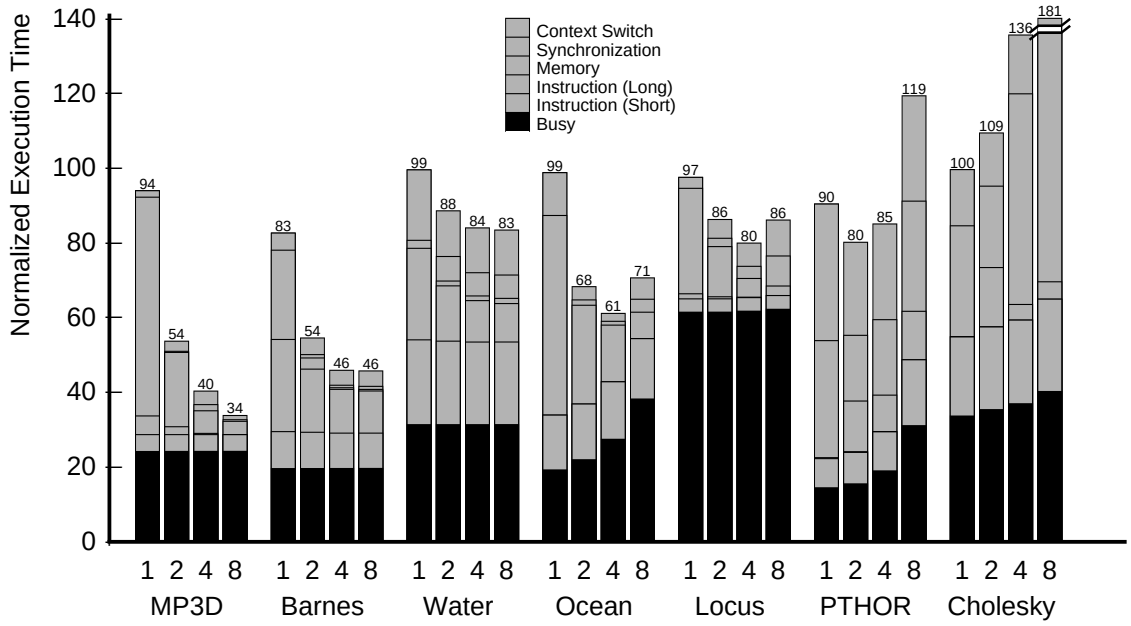


Figure 3.11: Blocked scheme with nonblocking loads.

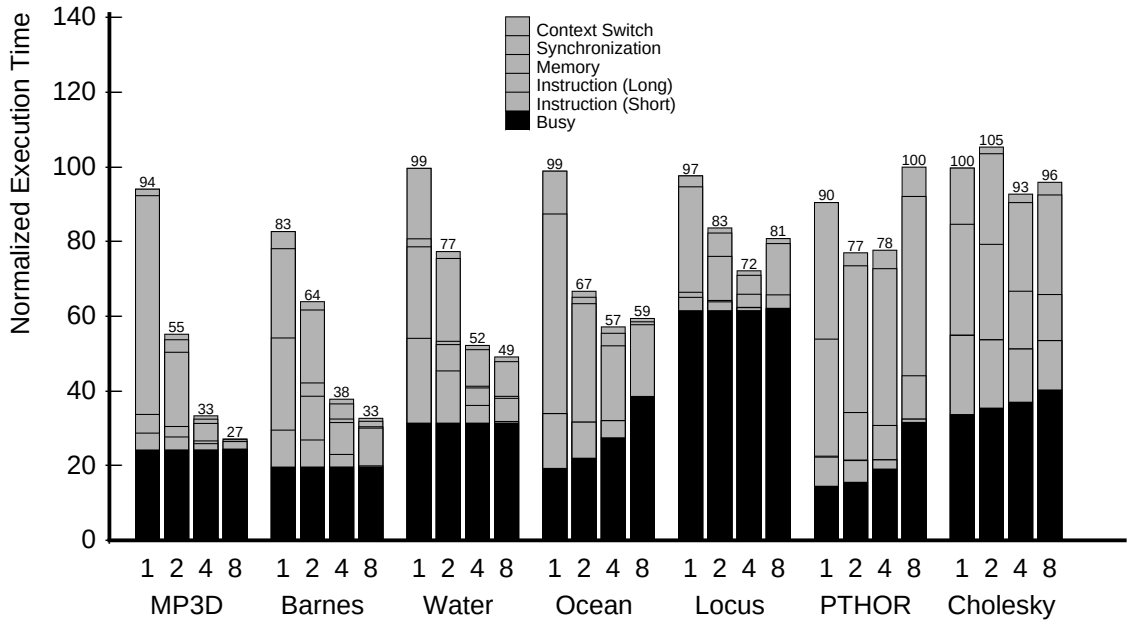


Figure 3.12: Interleaved scheme with nonblocking loads.

Table 3.20: Speedup of switch-on-use over switch-on-load.

Scheme	MP3D	Barnes	Water	Locus	Ocean	PTHOR	Cholesky	Mean
Blocked	1.02 (8)	1.02 (8)	1.02 (8)	1.10 (4)	1.11 (4)	1.10 (2)	1.00 (1)	1.05
Interleaved	1.07 (8)	1.05 (8)	1.00 (8)	1.06 (4)	1.13 (4)	1.13 (2)	1.08 (4)	1.07

from the compiler. The same compiler analyses which are used to predict which loads will miss in the cache for prefetching [Mow94] could also be used in determining which loads to group together. However, the small number of processor registers will tend to limit the amount of added performance which can be gained from improving the compiler support and it is likely that the decision to select switch-on-use over switch-on-load will be determined by whether the base architecture already supports nonblocking loads for other reasons.

3.5 Shared Resource Impact

Finally, to complete our multiprocessor analyses, we examine the impact of multiple contexts on the resources shared between the contexts. The shared resource most often focused on is the cache, since supporting multiple processes on a single-context processor is known to cause cache interference. However, there are a number of other resources important to processor performance that are shared by the contexts. These resources include the branch prediction hardware and the translation look-aside buffer. We will first show that running different threads of the same application on the multiple contexts does not significantly impact these additional shared resources. Then we will explore the cache interference issue in more detail.

3.5.1 Branch Prediction

Having multiple threads share the same 2048-entry, direct-mapped BTB has a small effect on its effectiveness as shown in Table 3.21. The table shows the miss rate of the TLB and the percent of branches mispredicted for both the single-context processor and the multiple-context processor (using the number of contexts which exhibited the largest speedup). Note that a branch can miss in the cache but still be considered predicted correctly if the branch is not taken. Cholesky is excluded from the table as multiple contexts did not improve its performance. The best-case number of contexts and its miss and misprediction statistics are shown on right side of the arrow, the single context statistics on the left side. We see that for nearly all applications, there is a

fairly small change in the percent of branches mispredicted. For some applications the miss rate actually decreases, as one context on the processor prefetches entries into the BTB for the other contexts.

Table 3.21: Effect of multiple contexts on the BTB.

	MP3D	Barnes	Water	Ocean	Locus	PTHOR
Blocked	1 → 8	1 → 8	1 → 8	1 → 4	1 → 4	1 → 2
Miss Rate (%)	6.2 → 6.2	17.1 → 15.5	11.3 → 11.4	10.3 → 7.5	3.6 → 3.7	23.6 → 18.5
Mispred. Rate (%)	7.6 → 7.5	15.7 → 15.5	11.4 → 12.2	2.3 → 3.5	3.8 → 4.0	17.0 → 14.4
Interleaved	1 → 8	1 → 8	1 → 8	1 → 4	1 → 4	1 → 2
Miss Rate (%)	6.2 → 6.3	17.1 → 15.5	11.3 → 12.4	10.3 → 7.5	3.6 → 4.9	23.6 → 19.8
Mispred. Rate (%)	7.6 → 7.5	15.7 → 16.0	11.4 → 14.2	2.3 → 4.1	3.8 → 4.0	17.0 → 15.2

3.5.2 Translation Look-aside Buffer

It is very important that the multiple-context processor does not cause TLB thrashing, as TLB misses are handled on most processors by either a software routine or by a hardware table-walker which is unlikely to be able to simultaneously process multiple TLB misses. Thus, multiple contexts will be unable to tolerate the increase in TLB miss overhead. The result of multiple threads using the same 128-entry fully associative data TLB are shown in Table 3.22 (for 4 Kbyte pages). The Shared row shows the miss rates when the TLB provides the ability for a page shared between the contexts to use a single TLB entry, while the Non-shared row shows the miss rate when each context needs its own separate mapping for shared pages. Only for MP3D does the ability to represent shared pages with a single TLB entry make a significant performance impact. In general, for the SPLASH applications, multiple contexts have a negligible effect on the TLB miss rate, and again we see that in a few cases where TLB entries can be shared the miss rate actually went down due to prefetching occurring between contexts.

While these interference numbers are very encouraging, the SPLASH applications tend to be small (by scientific application standards), and are not simulated with their largest data set sizes. Multiple-context TLB interference for applications with very large data sets needs to be studied in the future to ensure that the TLB does not become a limiting resource.

Table 3.22: Effect of multiple contexts on the data TLB miss rate (%).

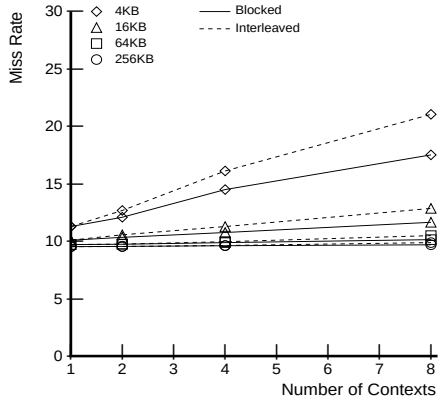
	MP3D	Barnes	Water	Ocean	Locus	PTHOR
Blocked	1 → 8	1 → 8	1 → 8	1 → 4	1 → 4	1 → 2
Shared	0.23 → 0.11	0.00 → 0.00	0.00 → 0.00	0.11 → 0.07	0.02 → 0.11	0.84 → 0.95
Non-shared	0.23 → 2.49	0.00 → 0.00	0.00 → 0.00	0.11 → 0.23	0.02 → 0.10	0.84 → 0.96
Interleaved	1 → 8	1 → 8	1 → 8	1 → 4	1 → 4	1 → 2
Shared	0.23 → 0.11	0.00 → 0.00	0.00 → 0.00	0.11 → 0.07	0.02 → 0.11	0.84 → 0.97
Non-shared	0.23 → 1.06	0.00 → 0.00	0.00 → 0.00	0.11 → 0.22	0.02 → 0.23	0.84 → 1.02

3.5.3 Data Cache

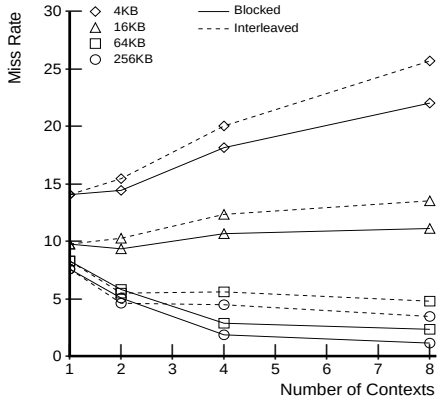
The effects of multiple contexts on the read miss rate for caches of differing sizes is first examined. We simulated cache sizes of 4, 16, 64, and 256 KBytes. Read miss rates for all applications but Ocean are shown in Figure 3.13. As we would expect, interference is largest for the smaller caches, and becomes less of an issue for the larger caches. We also see that the miss rate for the interleaved scheme tends to be slightly larger than for the blocked scheme, although there are a few cases where the interleaved scheme has a lower miss rate than the blocked scheme. Even though the miss rates are increasing due to interference, multiple contexts can tolerate a substantial portion of this increased memory latency provided the memory system has sufficient bandwidth. This was shown in Section 3.3.4, where even for the small cache sizes which exhibit a fair amount of interference, multiple contexts were still very effective. This ability to tolerate its own cache interference is an advantage multiple contexts has over prefetching, where increased cache interference due to prefetching leads to misses that are not tolerated and therefore slow down the processor.

Looking at Figure 3.13 more closely we see that for MP3D, Water, and PTHOR, the miss rate stays relatively constant as we increase the number of processors as long as the cache is 64 KB or larger. For smaller caches, inter-context interference leads to large increases in the miss rate for Water and PTHOR and more modest increases for MP3D. Both Locus and Cholesky show increases in miss rate for all cache sizes, with the increase being larger for the smaller caches. Barnes actually shows a decrease in miss rate with more contexts, provided the cache is 64 KB or larger. This reduced miss rate is due to one context prefetching data for the other contexts, and occurs in Barnes because processes near each other are working on overlapping data sets.

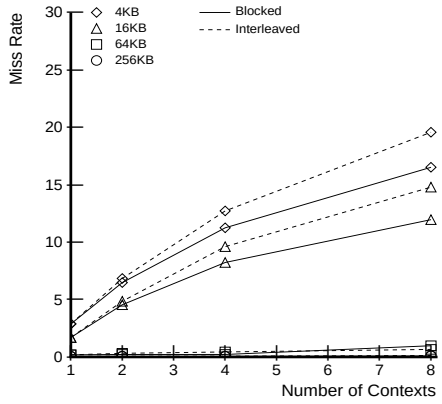
While we were unable to simulate Ocean with cache sizes other than 64 KBytes, Ocean has a rather interesting miss rate graph for the 64 KByte runs. The miss rate for Ocean depends



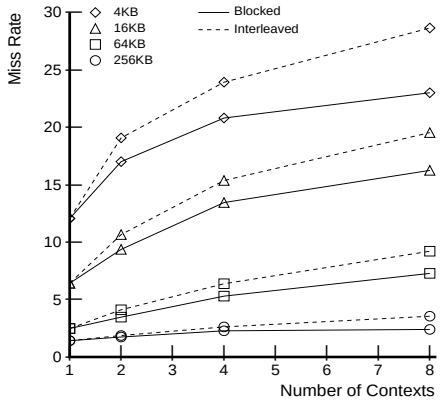
(a) MP3D



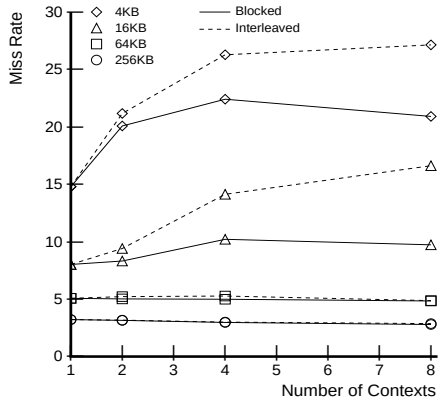
(b) Barnes



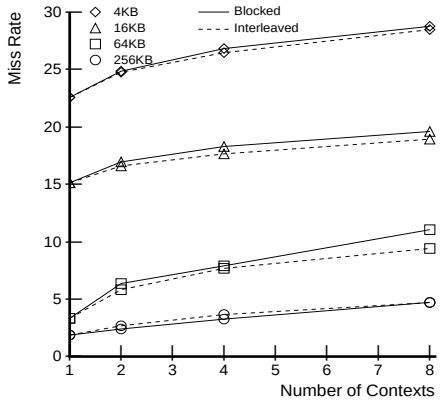
(c) Water



(d) Locus



(e) PTHOR



(f) Cholesky

Figure 3.13: Miss rates for varying cache sizes.

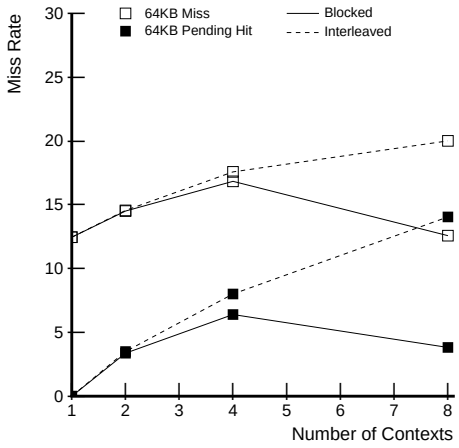
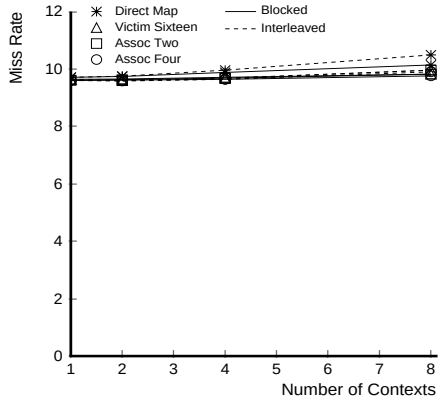


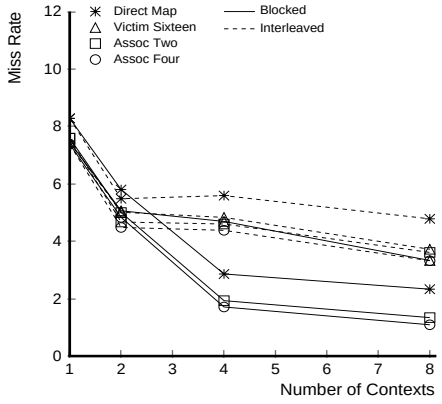
Figure 3.14: Ocean miss rates for a 64 KByte data cache.

strongly on what is labeled a miss. If every read that does not find its data in the cache is labeled as a miss, the miss rates with increasing numbers of contexts is shown by the upper set of lines in Figure 3.14. As we can see, the miss rates diverge at eight contexts per processor for the blocked and interleaved schemes. The lower set of lines explains this divergence. This set of lines plots the number of those misses which are actually hits to a pending cache line. In Ocean, a significant fraction of the misses occur right after a barrier as all contexts attempt to simultaneously access the same set of cache lines. For the interleaved scheme this results in all contexts issuing their load to the same address, followed by all contexts waiting for the data. Thus for the interleaved scheme, the number of pending hits increases as we increase the number of contexts per processor. For the blocked scheme, the first context will access the memory line, followed by a context switch and its associated delay, then the next context access the same memory line, and so on. As long as the number of contexts is small, the miss behavior for the blocked scheme is the same as for the interleaved scheme. However, when the number of contexts becomes large, the miss data actually returns before all contexts have been able to issue their request, causing contexts later in the issuing sequence to actually hit in the cache, and this results in the divergence in miss rates between the two schemes for eight contexts per processor.

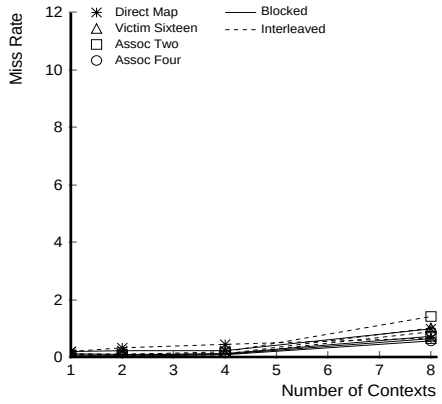
The miss rates for 64 Kbyte caches with varying associativities are shown in Figure 3.15. As would be expected, the general shapes of the miss rate curves for a given application is similar, with the miss rate decreasing as associativity is increased for many of the applications.



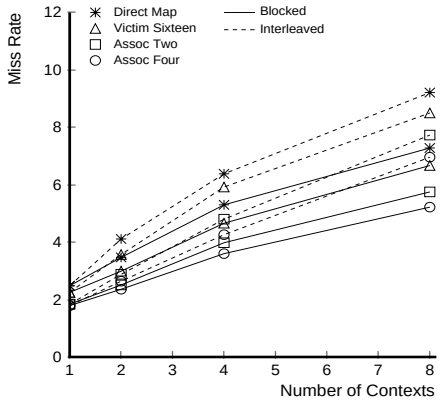
(a) MP3D



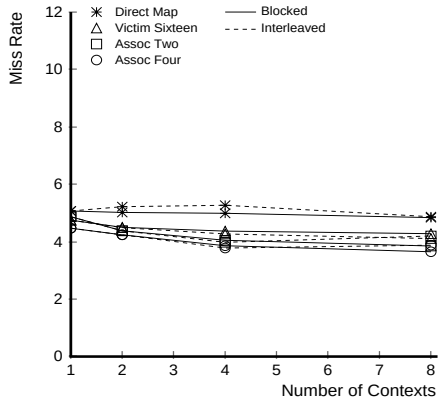
(b) Barnes



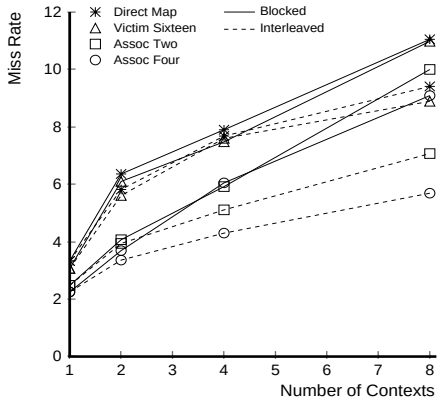
(c) Water



(d) Locus



(e) PTHOR



(f) Cholesky

Figure 3.15: Miss rates for varying cache associativities (64 KB caches).

3.6 Summary

This chapter has shown that most of the SPLASH applications show some benefit from multiple contexts. Three applications showed very large speedups due to multiple contexts (2.0 to 3.5 speedups with eight contexts per processor), three applications showed more modest gains (1.15 to 1.6 speedups), and only one application showed no gains at all. The interleaved scheme outperformed the blocked for all applications, including substantial performance differences for MP3D, Barnes, and Water.

The interleaved scheme was found to be effective across a range of multiprocessor configurations, including multiprocessors with memory latencies both smaller and larger than those of the Stanford DASH [LLJ+93], and with varying cache sizes and associativities. We also showed that combining multiple contexts with relaxed memory consistency and nonblocking loads results in relatively modest performance gains. Being able to combine multiple contexts with these other latency tolerance mechanisms without incurring in a performance loss is important, as the other latency tolerance mechanisms exploit parallelism within a single thread and work well in situations where limited parallelism prevents the use of multiple contexts.

Finally, we examined the impact of multiple contexts on the resources shared between the contexts, namely the caches, TLB, and BTB. We found the interference to be modest for caches 64 KB and larger and for our base TLB size of 128 entries and BTB size of 2048 entries. For some applications, cache interference was significant for the smaller caches (4 and 16 KB). However, because the multiple contexts were able to tolerate the additional cache misses caused by this interference, these applications still showed significant performance improvements for the smaller cache sizes.

This chapter has shown that the interleaved scheme is quite effective in speeding up parallel applications on a shared-memory multiprocessor. In the following chapter, we explore its use in a high-performance uniprocessor environment.

Chapter 4

Multiple Contexts: Utility for Uniprocessors

Since large-scale multiprocessors are generally built using off-the-shelf microprocessors whose design is targeted for uniprocessors, it is important that multiple contexts also address the needs of uniprocessors. Applying multiple-context processors to a general-purpose uniprocessor has not received much attention from the research community.

Culler et. al. [CGL92] have examined using blocked multiple-context processors to improve throughput for a multiprogrammed uniprocessor. Their study uses traces of a subset of the SPEC benchmarks [Car92] to simulate multiple contexts across a wide range of cache sizes and associativity for two, four, and eight contexts per processor. They found multiple contexts to actually reduce the cache miss rate for associative caches due to the blocked multiple-context processor giving threads with lower miss ratios a larger fraction of the processor cycles. This is in contrast with the applications timesharing a single-context processor, where all applications are given the same fraction of the total processor cycles.

Of course, the bottom line is the increase in processor throughput due to the multiple contexts. For a 50 cycle memory latency and switch cost of 5 cycles, they found a processor utilization increase of 13% for two contexts, 29% for four contexts, and 39% for eight contexts. Their results are encouraging — unfortunately, their study has several limitations. First, they ignore operating system overhead, including the operating system displacing cache state between context switches. Second, they assume a relatively aggressive switch cost of 5 cycles, but only a modestly aggressive memory system consisting of a single level of caches backed by a 50 cycle memory. Finally, the most important limitation of their study is that they model all instructions as ideal,

taking only a single cycle, and therefore do not account for any processor stall time except that due to memory latency. For modern superpipelined or superscalar processors, dependencies between instructions prevent the maximum instruction issue rate from being sustained for most applications. Pipeline effects need be modeled before any conclusions can be drawn about the advantages of multiple contexts. In this chapter we will examine the improvement in uniprocessor throughput for both the blocked and interleaved schemes while avoiding the limitations of this previous study.

Section 4.1 starts this chapter by describing our simulation environment and workloads. We simulate six workloads composed of several applications from the SPEC suite and another workload composed of uniprocessor versions of the SPLASH applications. In Section 4.2, we present simulation results for two and four contexts per processor. We show that the blocked scheme is able to increase throughput only slightly (11% increase with four contexts), whereas the interleaved scheme shows much larger increases in throughput (50% with four contexts). The higher switch cost of the blocked scheme prevents it from doing well in this environment. In Section 4.3 the shared resource interference is examined. Instruction cache, BTB, and TLB interference is larger than encountered in the previous chapter for a multiprocessor, but is still small enough to be greatly outweighed by the gains of the multiple contexts. Data cache interference turns out to be surprisingly small for most workloads. Finally, we summarize the chapter in Section 4.4.

4.1 Methodology

The base uniprocessor system employs the same microprocessor as for the multiprocessor studies. Of course, the memory system is different for the uniprocessor studies, as shown in Figure 4.1. The base cache sizes for our uniprocessor studies are 64 Kbyte direct-mapped instruction and data caches, backed by a 1 Mbyte direct-mapped, unified secondary cache. For our uniprocessor studies, we simulate the instruction cache in full detail, since different applications may cause instruction cache interference.

The memory system is four-way interleaved, connected to the processor across a high-speed, split-transaction bus. The unloaded memory access times are given in Table 4.1. Cache and memory contention are modeled, and can add to these latencies. While the data cache is lockup-free, the instruction cache is blocking, therefore no context switching will be done for instruction cache misses. We have deliberately modeled a very aggressive memory system, since this will be the worst case situation for multiple contexts.

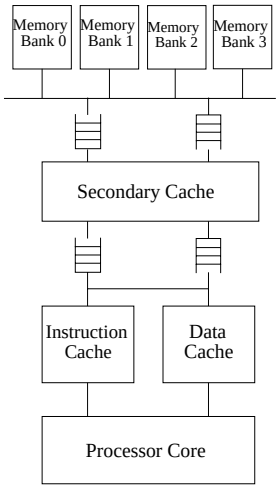


Figure 4.1: Base uniprocessor architecture.

Table 4.1: Uniprocessor memory latencies.

Hit in Primary Cache	1 cycle
Hit in Secondary Cache	9 cycles
Reply from Memory	34 cycles

The operating system model is based on measurements by Torrellas [Tor92] and Gupta, Tucker, and Stevens [GTS91] of IRIX, a UNIX System V variant, running on a Silicon Graphics 4D/340 [BJS88]. The time-slice used by the operating system is 30 ms, and assuming a 200 MHz processor this translates to a scheduler interrupt every six million processor cycles. The scheduler uses a simple affinity mechanism which keeps the same application scheduled on the processor for the equivalent of three time slices (i.e. three time slices for the single-context processor, six for the two-context processor, etc.). Because of this affinity mechanism, the actual number of processes which are switched by a scheduler call varies from none to the number of hardware contexts supported.

The simulator does not actually run the operating system scheduler, rather, the operating system is modeled as a routine with negligible latency which displaces a number of cache lines. Scheduler execution times were determined to be negligible based on measurements made of the IRIX scheduler latency [GTS91]. The cache interference caused by the scheduler depends on the number of contexts which must be swapped, as shown in Table 4.2. Cache interference is modeled by issuing the number of memory requests given in the table to addresses selected using

a random distribution. These cache interference numbers were taken from measurements of the IRIX operating system [Tor92].

Table 4.2: Operating system costs.

Processes Switched	**Instruction Cache Interference**	**Data Cache Interference**
0	4	25
N	$50N + 158$	$100N + 180$

Six workloads are generated using members of the SPEC benchmarks. The members selected include Doduc, Eqntott, Li, Matrix300, Tomcatv, and NASA7 broken into its kernels: Btrix, Cholsky, Cfft2d, Emit, Gmtry, Mxm, and Vpenta.[1] These workloads were constructed to have the following characteristics:

IC Stresses the instruction cache.

DC Stresses the data cache.

DT Stresses the data TLB.

FP Floating-point intensive.

R0 Random workload.

R1 Random workload.

A seventh workload (labeled **SP**) uses the uniprocessor versions of four SPLASH applications. These seven workloads are listed in Table 4.3.

Since many applications take several minutes of CPU time each, the workloads are not run to completion. Instead, the workload is run for 36 time-slices (roughly 1 second of CPU time). Because we are only simulating a fraction of the complete application, it is important that we are simulating the section of the application responsible for most of the execution time. To ensure that statistics are gathered for the relevant application section, references from the application are sent to the simulator only after entering this portion of the code. In addition, to remove cold-start effects, each application in the workload is run for a time slice before simulation statistics are

[1]These applications were selected from the SPEC 1 suite solely on the basis of being able to pass through our compilation system with no or minor adjustments to the application itself.

Table 4.3: Uniprocessor workloads.

Workload						
IC	**DC**	**DT**	**FP**	**R0**	**R1**	**SP**
Doduc	Cfft2d	Btrix	Emit	Emit	Mxm	MP3D
Li	Gmtry	Cholsky	Cholsky	Btrix	Li	Water
Eqntott	Tomcatv	Gmtry	Doduc	Cfft2d	Matrix300	Locus
Mxm	Vpenta	Vpenta	Matrix300	Eqntott	Tomcatv	Barnes

gathered. Thus, when simulation statistics are being gathered, the applications are in their major routines and the caches have been loaded.

Table 4.4: Characteristics of measured portion of uniprocessor applications.

	Proc. Util. Breakdown (%)				Runlength
Application	**Busy**	**Inst.**	**Mem.**	**Other**	**(proc cycles)**
Doduc	26.7	64.3	0.7	8.3	1660
Eqntott	71.3	20.5	8.0	0.2	144
Li	59.9	24.9	0.5	14.7	1610
Matrix300	36.5	54.8	8.7	0.0	167
Tomcatv	36.1	35.1	28.3	0.5	53
Btrix	45.3	40.0	14.3	0.4	85
Cholsky	36.6	41.6	21.4	0.4	75
Cfft2d	35.9	38.6	23.6	1.9	32
Emit	43.0	51.6	5.4	0.0	505
Gmtry	22.1	26.4	50.4	1.1	24
Mxm	51.9	35.5	10.5	1.6	78
Vpenta	13.1	20.8	64.8	2.3	13
Barnes	30.2	64.8	2.2	2.8	424
Locus	90.3	4.8	4.5	0.5	299
Mp3d	51.1	24.9	22.3	1.7	144
Water	36.4	60.8	0.2	2.6	18000

Table 4.4 shows the uniprocessor statistics for the base applications. *Busy* again refers to cycles spent doing useful work. *Instruction* refers to time spent stalled due to pipeline dependencies. *Memory* refers to the time spent stalled on memory due to data cache misses. Multiple contexts provide the potential to tolerate these first two forms of stall. *Other* refers to the remaining stall time that multiple contexts will not be able to tolerate. The major components of the *Other* category are stalls due to instruction misses and data TLB faults. Finally, the runlength, or number of cycles between read misses, is also given for each application. Note that the applications

are generally spending more time busy than the multiprocessor applications from the previous chapter. Instruction latency tends to be fairly large for most of the applications, with memory latency dominating instruction latency only for Gmtry and Vpenta. Thus, based on Table 4.4, the uniprocessor environment would seem to be much more difficult for effectively employing multiple contexts when compared to the multiprocessor environment.

The workloads selected represent an environment where several large jobs are being multiprogrammed on a single workstation. This is the situation found on the workstations in our research lab, however, many workstations run with one large job in the background and smaller jobs in the foreground. While we do not explicitly model this environment here, our results should be indicative of the benefits that multiple contexts can provide by allowing the smaller jobs to be loaded and run on the processor without requiring the larger job to be switched out. In addition to providing this advantage, multiple contexts allow background applications which suffer significantly from memory latency to be written as parallel programs to take advantage of the latency tolerance. Finally, there are a large number of applications which are designed to run on workstation clusters or small-scale multiprocessors that are already multithreaded and can take advantage of the multiple contexts on the processor. While the throughput improvement of several large jobs is not a metric directly applicable to these other situations, all these situations benefit from multiple contexts.

4.2 Effectiveness of Multiple Contexts

Before we examine the effects of multiple contexts on multiprogramming throughput, we first need to discuss the impact of multiple contexts on process scheduling. As was observed in [CGL92], applications with lower miss rates tend to get more cycles under blocked multiple contexts than applications with higher miss rates. This is because lower miss rates usually translate into longer runlengths, and assuming a strict round-robin scheduling, the fraction of the total processor cycles allocated to each application will depend on the size of its runlength relative to the other runlengths as shown in Equation 4.1, where R_i denote the runlength of process i and N is the total number of contexts on the processor.

$$Percent\ of\ Processor\ Cycles\ Allocated\ to\ Process\ i = \frac{R_i}{R_1 + R_2 + \cdots + R_N} \quad (4.1)$$

A similar effect also occurs for the interleaved scheme. Assuming the processor supports N contexts, an application receives $\frac{1}{N}$ of the processor cycles as long as it is not unavailable

due to an outstanding memory request. Since applications with longer runlengths spend less of their time waiting for memory, they will get a larger fraction of the processor. However, for the interleaved scheme, an additional factor is involved in determining the fraction of time spent on each application, namely the total number of cycles an application spends stalled after latency has been tolerated. Because the interleaved scheme switches between available contexts on a per-*instruction* basis, applications with larger total stalls will get a larger fraction of the processor cycles.

To illustrate the effect of untolerated stalls on the number of cycles devoted to each application on the interleaved processor, consider the following simple example. Two applications (A and B) are sharing the interleaved multiple-context processor. Both applications never cache miss. Application A has no stalls when running on the single-context processor; each instruction in application B is followed by a two-cycle pipeline dependency stall on the single-context processor. When A and B are interleaved on the multiple-context processor, one of the stall cycles of application B will be tolerated by the interleaving, leaving one untolerated cycle of stall. Because A and B issue instructions round-robin, B will get two processor cycles (one active and one stalled) for every one processor cycle of A. When the applications running on the interleaved processor have comparable miss rates, these stalling effects can have a significant impact on the percentage of total time devoted to each application.

The behavior of both schemes is in contrast with the same applications timesharing a single-context processor, where each application receives the same fraction of the processor's cycles assuming all applications are running at the same priority. Since users would like their application to get its fair share of the processor, it is desirable to provide feedback to the operating system on how much time is being devoted to each context. The operating system can then incorporate this information into its scheduling algorithm to make the processor allocation more fair.

Since we would like to compare the blocked and interleaved schemes based on how well they improve the throughput of all applications in the workload, not on whether they devote more processor time to applications with better memory or worse pipeline behavior, we will assume that the hardware provides this information to the operating system, and the operating system schedules the workload to even out the amount of processor cycles devoted to each application. Therefore, we will normalize our results (which do not include the effects of this feedback to the operating system) to the case where each application out of N is given $\frac{1}{N}$ of the processor.

The improvement in throughput for our workloads resulting from blocked multiple contexts is shown in Figure 4.2. Processor utilization is broken into five categories: *busy*, time spent doing

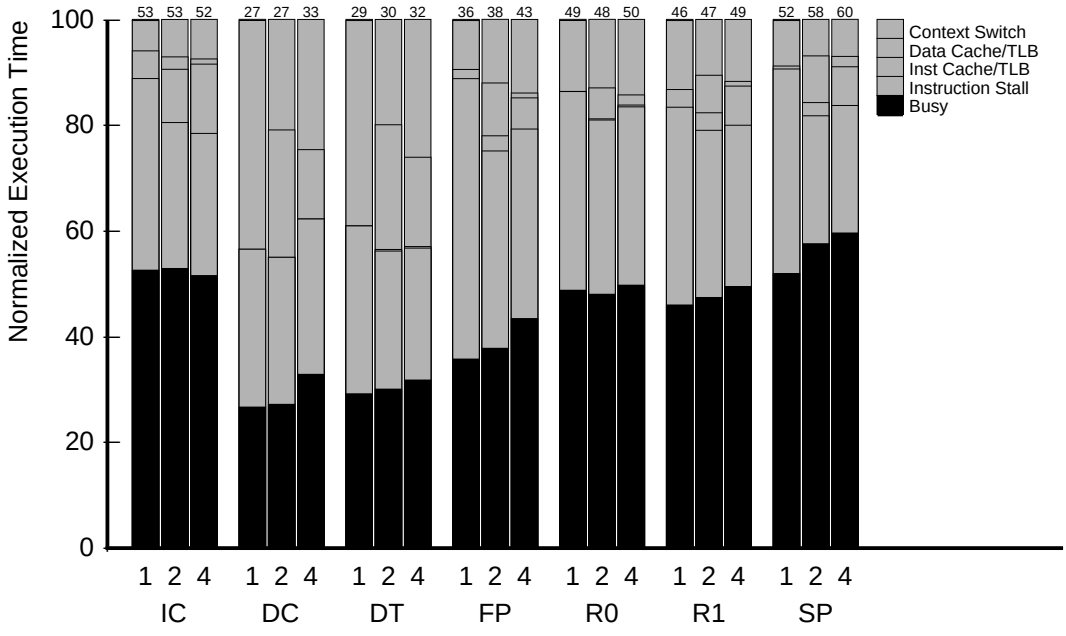


Figure 4.2: Blocked scheme processor utilization — switch on primary cache miss.

useful work, *instruction*, time stalled due to pipeline dependencies, *inst cache/TLB*, time stalled due to instruction references, *data cache/TLB*, time stalled on memory due to data references, and *context switch*, time spent context switching. The number on top of the bars show the percent of time spent busy. In general, processor utilization does not increase much with additional contexts. This is because many of the workloads simply do not contain that much memory or long-instruction latency for the multiple contexts to tolerate. For workloads where there is a fairly large amount of memory latency, such as **DC** and **DT**, the high cost of context switching removes much of the gain of tolerating the memory latency.

Selecting the proper criteria for determining when to context switch is not as straightforward for the uniprocessor environment as for the multiprocessor, since many misses in the primary data cache are likely to hit in the secondary cache. Because the cost of a primary cache miss that hits in the secondary cache is close to the cost of the context switch, there may be an advantage in waiting to context switch until a miss occurs in the secondary cache. Figure 4.3 shows that for some workloads this results in the same performance as switching on the primary miss, while for a few workloads this results in a slightly lower performance. For our memory latencies switching on the primary results in the best performance, however, with different memory latencies the choice

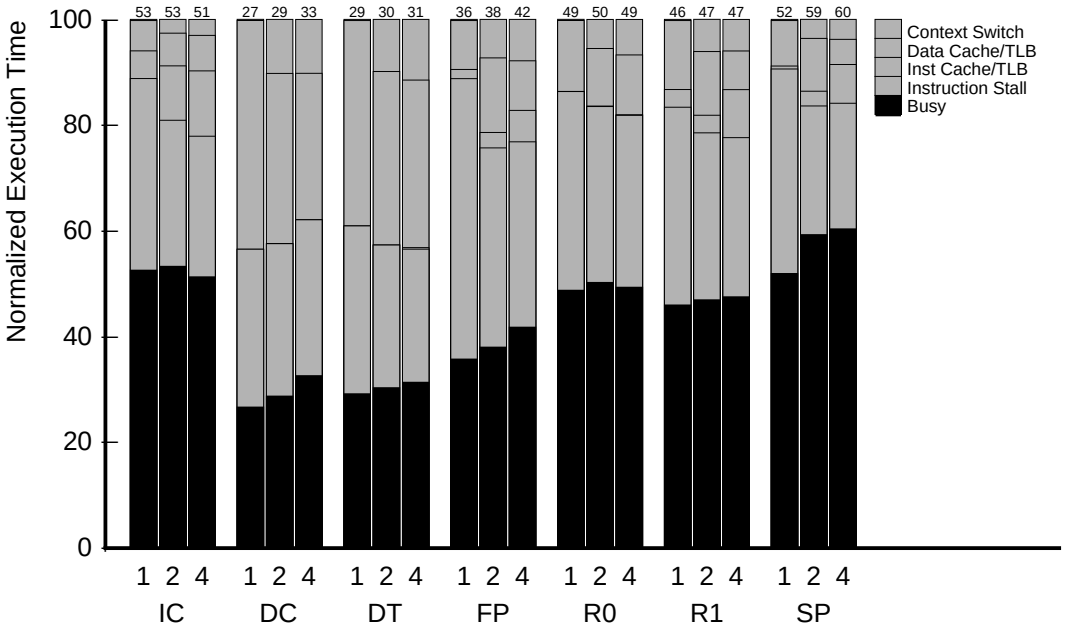


Figure 4.3: Blocked scheme processor utilization — switch on secondary cache miss.

might not be so clear. This complicates the implementation of the multiple-context processor targeted at the uniprocessor, as design simplicity argues for a fixed switch criteria.

The interleaved scheme does not encounter this dilemma, as its switch cost is small enough to tilt the equation toward always switching on primary misses. A breakdown of processor utilization for the interleaved scheme is shown in Figure 4.4. The interleaved scheme is much more successful in tolerating both memory and instruction latency. Processor utilization increases significantly under the interleaved scheme for workloads with large amounts of instruction latency, as the cycle-by-cycle interleaving tolerates these shorter instruction latencies, while the scoreboard-triggered backoff tolerates the long instruction latency. In addition, workloads with large amounts of memory latency (**DC** and **DT**) gain over the blocked scheme due to the lower switch cost of the interleaved scheme.

Table 4.5 summarizes the increase in processor utilization (and therefore throughput) for the two schemes. The blocked scheme increases throughput by only 11% with four contexts per processor, while the interleaved scheme is able to increase throughput by 50% with four contexts per processor. The 50% increase in throughput is much more compelling when making a decision concerning the cost/performance tradeoffs of adding multiple contexts to a processor.

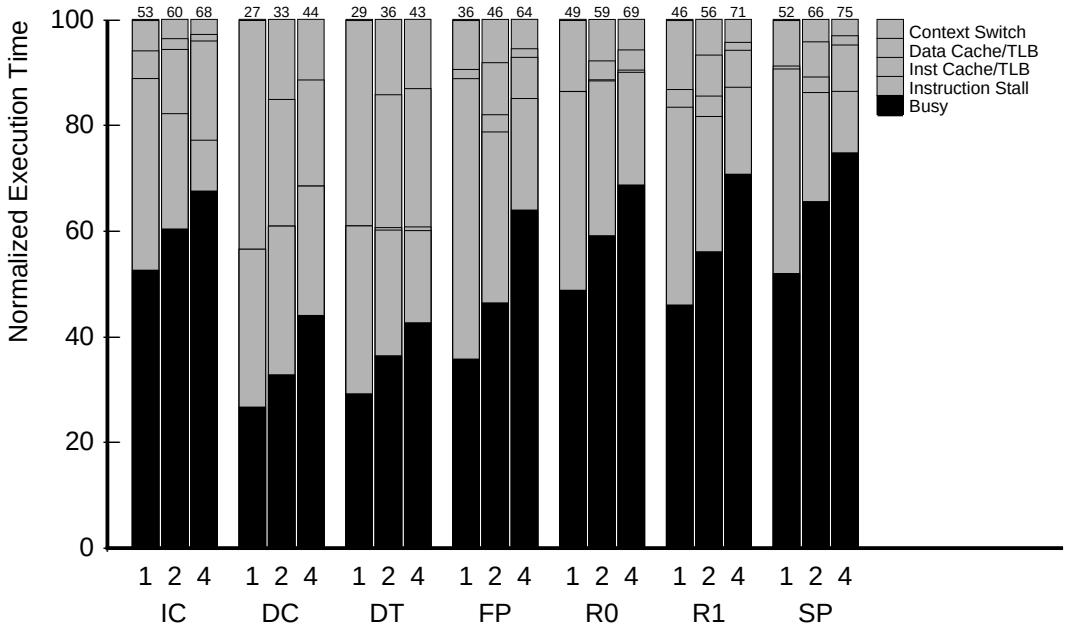


Figure 4.4: Interleaved scheme processor utilization.

Table 4.5: Increase in application throughput with multiple contexts.

	Scheme	IC	DC	DT	FP	R0	R1	SP	Mean
Two	Interleaved	1.15	1.23	1.22	1.29	1.21	1.21	1.26	1.22
Contexts	Blocked	1.01	1.02	1.00	1.04	1.00	1.01	1.10	1.03
Four	Interleaved	1.28	1.65	1.46	1.79	1.41	1.54	1.44	1.50
Contexts	Blocked	1.01	1.23	1.09	1.21	1.02	1.08	1.15	1.11

4.3 Shared Resource Impact

Sharing the caches, TLBs, and branch prediction logic between distinct applications is more challenging than sharing them between multiple threads of the same application. In this section, we examine the BTB, TLB, instruction cache, and data cache interference for our multiprogrammed workloads.

4.3.1 Branch Prediction

Sharing a BTB between multiple applications requires some additional hardware support. Because we want to avoid one context using the branch prediction information of another context, we

have assumed that the BTB is tagged with a process identifier (PID) in addition to the normal address tag. We use PID tagging instead of context identifier (CID) tagging because we still want applications with the same process identifier but different context identifiers (i.e. threads of a single parallel application) to share entires. Interference for our direct-mapped, 2048-entry BTB with PID tagging is shown in Table 4.6. This table shows the change in both BTB miss and misprediction rate when going from the single-context to the four-context processor. Applications which have a very high single-context prediction rate can be especially affected by BTB interference. For example, branch prediction in Tomcatv is primarily determined by the predictability of the inner loop branch. If this prediction information should get displaced from the BTB, a large increase in branch misprediction can occur, as occurred for workload **R1** for both schemes. The important point is that despite this increase in BTB interference, interleaved multiple contexts were still able to significantly improve uniprocessor throughput.

4.3.2 Translation Look-aside Buffers

The change in miss rates for the 128-entry, fully associative data TLB when going from one to four contexts is shown in Table 4.7 (for 16 Kbtye pages).[2] For most workloads the increase in the data TLB miss rate due to supporting four contexts was small. The exceptions were **DC** and **DT**, where the data TLB miss rates for some of the NASA7 kernels went from a few hundredths of a percent to up to nearly seven percent, worst case. These kernels were already placing pressure on the data TLB before multiple contexts. Again, despite the large increase in TLB miss rates, multiple contexts were able to improve throughput for these workloads. These results do point out the need for the operating system or miss-handling hardware to keep track of how many TLB misses are occurring to prevent the operating system from continuously scheduling contexts which are thrashing in the TLB.

Interference turned out to not be an issue for the 32-entry, fully-associative instruction TLB simulated. The largest increase occurred for Doduc in workload **IC**, where the instruction miss rate with four contexts per processor increased from under 0.01% to 0.02%.

[2] 16 KByte pages were used for the uniprocessor studies (in contrast with the 4 KByte pages of the multiprocessor studies), since the single-context data TLB miss rate was unacceptably high for several of the NASA7 kernels when using 4 KByte pages, and would have distorted the results.

Table 4.6: Effect of multiple contexts on the BTB.

Workload IC	Blocked	**doduc**	**li**	**eqntott**	**mxm**
	Miss Rate (%)	33.8 → 43.4	22.9 → 23.8	0.9 → 1.8	0.0 → 0.3
	Misprediction Rate (%)	13.1 → 20.0	15.3 → 16.0	1.0 → 1.6	0.4 → 0.7
	Interleaved	**doduc**	**li**	**eqntott**	**mxm**
	Miss Rate (%)	33.8 → 40.3	22.9 → 25.0	0.9 → 1.7	0.0 → 0.3
	Misprediction Rate (%)	13.1 → 18.0	15.3 → 16.8	1.0 → 1.6	0.4 → 0.7
Workload DC	Blocked	**cfft2d**	**gmtry**	**tomcatv**	**vpenta**
	Miss Rate (%)	0.2 → 0.4	0.3 → 0.2	0.5 → 0.5	0.0 → 0.3
	Misprediction Rate (%)	0.8 → 0.9	1.2 → 1.2	0.5 → 0.5	0.6 → 0.9
	Interleaved	**cfft2d**	**gmtry**	**tomcatv**	**vpenta**
	Miss Rate (%)	0.2 → 0.4	0.3 → 0.1	0.5 → 0.6	0.0 → 0.2
	Misprediction Rate (%)	0.8 → 0.9	1.2 → 1.2	0.5 → 0.6	0.6 → 0.8
Workload DT	Blocked	**btrix**	**cholsky**	**gmtry**	**vpenta**
	Miss Rate (%)	2.5 → 3.5	3.4 → 4.8	0.3 → 0.5	0.0 → 0.0
	Misprediction Rate (%)	3.4 → 4.4	1.2 → 2.4	1.2 → 1.5	0.6 → 0.6
	Interleaved	**btrix**	**cholsky**	**gmtry**	**vpenta**
	Miss Rate (%)	2.5 → 4.0	3.4 → 3.6	0.3 → 0.3	0.0 → 2.8
	Misprediction Rate (%)	3.4 → 4.9	1.2 → 1.2	1.2 → 1.3	0.6 → 3.4
Workload FP	Blocked	**emit**	**cholsky**	**doduc**	**matrix300**
	Miss Rate (%)	47.2 → 47.4	3.5 → 3.9	32.4 → 32.9	2.5 → 3.1
	Misprediction Rate (%)	0.3 → 0.4	1.2 → 1.7	13.1 → 13.5	1.2 → 1.9
	Interleaved	**emit**	**cholsky**	**doduc**	**matrix300**
	Miss Rate (%)	47.2 → 48.0	3.5 → 3.6	32.4 → 32.9	2.5 → 2.9
	Misprediction Rate (%)	0.3 → 0.5	1.2 → 1.6	13.1 → 13.6	1.2 → 1.7
Workload R0	Blocked	**emit**	**btrix**	**cfft2d**	**eqntott**
	Miss Rate (%)	47.2 → 48.1	2.5 → 2.7	0.2 → 0.4	0.9 → 0.6
	Misprediction Rate (%)	0.3 → 0.2	3.4 → 3.6	0.8 → 1.0	1.1 → 0.8
	Interleaved	**emit**	**btrix**	**cfft2d**	**eqntott**
	Miss Rate (%)	47.2 → 46.8	2.5 → 2.7	0.2 → 0.3	0.9 → 0.9
	Misprediction Rate (%)	0.3 → 0.3	3.4 → 3.6	0.8 → 0.9	1.1 → 1.0
Workload R1	Blocked	**mxm**	**li**	**matrix300**	**tomcatv**
	Miss Rate (%)	0.0 → 0.2	20.4 → 20.5	2.5 → 3.7	0.5 → 8.5
	Misprediction Rate (%)	0.4 → 0.6	15.3 → 15.4	1.2 → 2.5	0.5 → 8.5
	Interleaved	**mxm**	**li**	**matrix300**	**tomcatv**
	Miss Rate (%)	0.0 → 0.3	20.4 → 20.6	2.5 → 3.4	0.5 → 3.4
	Misprediction Rate (%)	0.4 → 0.7	15.3 → 15.4	1.2 → 2.2	0.5 → 3.4
Workload R2	Blocked	**mp3d**	**water**	**locus**	**barnes**
	Miss Rate (%)	1.9 → 10.9	11.7 → 15.4	2.5 → 3.0	13.8 → 26.7
	Misprediction Rate (%)	7.4 → 9.4	11.5 → 14.2	2.7 → 2.9	12.1 → 22.0
	Interleaved	**mp3d**	**water**	**locus**	**barnes**
	Miss Rate (%)	1.9 → 15.4	11.7 → 16.8	2.5 → 3.4	13.8 → 24.0
	Misprediction Rate (%)	7.4 → 11.2	11.5 → 15.1	2.7 → 3.3	12.1 → 19.5

Table 4.7: Effect of multiple contexts on the data TLB.

Workload IC	Blocked Miss Rate (%)	**doduc** 0.00 → 0.00	**li** 0.00 → 0.00	**eqntott** 0.00 → 0.00	**mxm** 0.00 → 0.00
	Interleaved Miss Rate (%)	**doduc** 0.00 → 0.00	**li** 0.00 → 0.00	**eqntott** 0.00 → 0.00	**mxm** 0.00 → 0.00
Workload DC	Blocked Miss Rate (%)	**cfft2d** 0.00 → 1.62	**gmtry** 0.03 → 4.67	**tomcatv** 0.02 → 0.33	**vpenta** 0.03 → 3.68
	Interleaved Miss Rate (%)	**cfft2d** 0.00 → 1.52	**gmtry** 0.03 → 4.83	**tomcatv** 0.02 → 0.60	**vpenta** 0.03 → 3.46
Workload DT	Blocked Miss Rate (%)	**btrix** 0.01 → 1.85	**cholsky** 0.01 → 0.69	**gmtry** 0.03 → 6.83	**vpenta** 0.03 → 4.42
	Interleaved Miss Rate (%)	**btrix** 0.01 → 2.61	**cholsky** 0.01 → 0.71	**gmtry** 0.03 → 5.98	**vpenta** 0.03 → 4.17
Workload FP	Blocked Miss Rate (%)	**emit** 0.00 → 0.00	**cholsky** 0.01 → 0.02	**doduc** 0.00 → 0.00	**matrix300** 0.01 → 0.02
	Interleaved Miss Rate (%)	**emit** 0.00 → 0.00	**cholsky** 0.01 → 0.01	**doduc** 0.00 → 0.00	**matrix300** 0.01 → 0.01
Workload R0	Blocked Miss Rate (%)	**emit** 0.00 → 0.02	**btrix** 0.01 → 0.32	**cfft2d** 0.00 → 0.30	**eqntott** 0.00 → 0.02
	Interleaved Miss Rate (%)	**emit** 0.00 → 0.03	**btrix** 0.01 → 0.32	**cfft2d** 0.00 → 0.30	**eqntott** 0.00 → 0.04
Workload R1	Blocked Miss Rate (%)	**mxm** 0.00 → 0.00	**li** 0.00 → 0.00	**matrix300** 0.01 → 0.02	**tomcatv** 0.02 → 0.02
	Interleaved Miss Rate (%)	**mxm** 0.00 → 0.00	**li** 0.00 → 0.00	**matrix300** 0.01 → 0.02	**tomcatv** 0.02 → 0.02
Workload R2	Blocked Miss Rate (%)	**mp3d** 0.01 → 0.01	**water** 0.00 → 0.00	**locus** 0.00 → 0.01	**barnes** 0.00 → 0.02
	Interleaved Miss Rate (%)	**mp3d** 0.01 → 0.01	**water** 0.00 → 0.00	**locus** 0.00 → 0.01	**barnes** 0.00 → 0.01

4.3.3 Caches

Finally, the miss rates for both instruction and data caches of differing sizes (4, 16, 64, and 256 KBytes) and differing associativities were examined. Indexing into the instruction cache is based on both the instruction address and the process identifier, with the PID hashed into the upper bits. This is done to prevent the code of the multiple applications from lying on top of each other in the cache. Indexing into the data cache is based solely on the data address. While there are advantages to also hashing the PID into the index function for the data cache, as the stacks and private data spaces of multiple applications tend to have the same virtual addresses, we performed some simulations which showed the effects to be negligible.

Data Cache

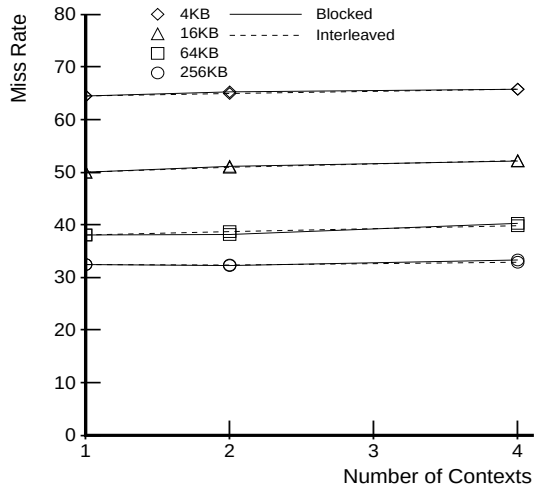
The read miss rates for the primary data cache are shown in Figure 4.5. Note that the scale on the miss rate axis is not constant for all workloads. In general, interference is not that large, even for some of the smaller caches, however, there are several workloads, such as **FP** and **SP**, that do have substantial interference between the applications. These modest increases in miss rate are quite encouraging, as one would expect the interference for a multiprogramming workload to be larger. However, the average miss rate does not show the entire picture, and even for cases where the miss rate does not increase substantially, the miss rates of certain applications in the workload may increase a fair amount. As an example, Figure 4.6 shows that the miss rate of all applications in workload **R1** increase at roughly the same rate, whereas most of the increase for workload **SP** is due to Barnes. However, even with this large increase in miss rate for Barnes, **SP** showed a good throughput improvement for interleaved multiple contexts.

Instruction Cache

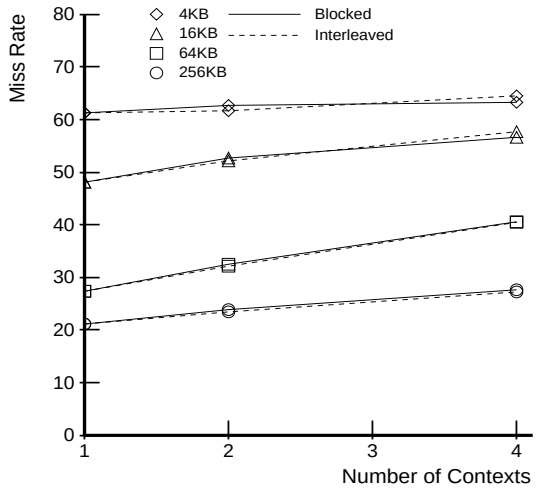
Figure 4.7 shows that instruction cache interference is fairly large for the 4 and 16 KByte caches. For the 64 and 256 KByte caches, the instruction cache interference between the applications is fairly small. Instruction cache interference is more serious than data cache interference, since we assumed the instruction cache was not lockup-free. These cache interference studies show that lockup-free instruction caches should be seriously considered if the instruction cache is 16 KBytes or smaller. In addition, while our workloads exhibited little interference for the larger caches, the operating system should be provided with some feedback on instruction cache interference to avoid scheduling applications that are thrashing.

4.4 Summary

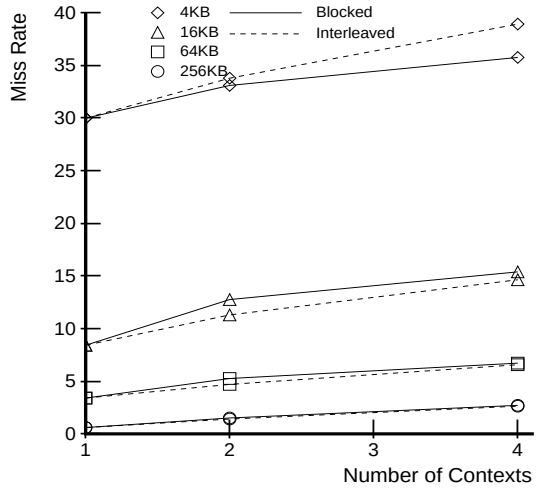
This chapter has shown that achieving large performance gains from multiple contexts in the uniprocessor environment is more difficult than for the multiprocessor environment. The lower switch cost and ability to tolerate short instruction latencies of the interleaved scheme is crucial for good performance in uniprocessors. The interleaved scheme was able to improve throughput by 50% for the workloads we examined, while the blocked scheme could only achieve a 11% increase. We also examined shared resource interference between the contexts and found it to be larger for a multiprogramming workload than for multiple threads of the same parallel application.



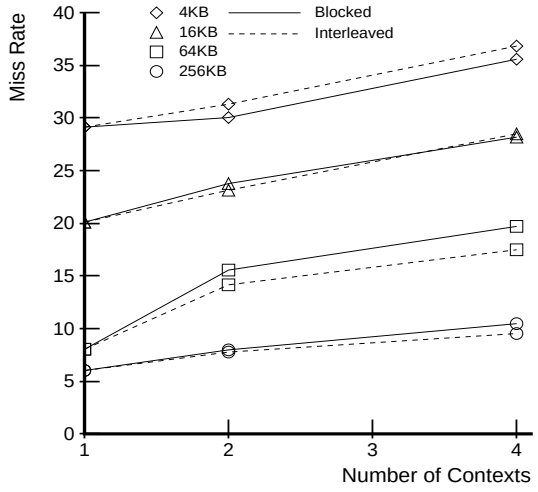
(a) DC



(b) DT

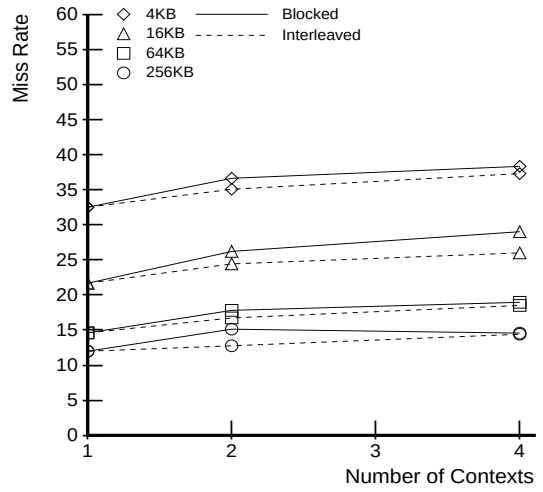


(c) IC

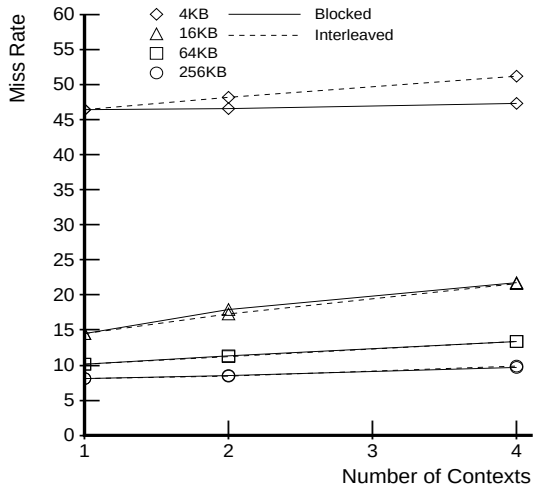


(d) FP

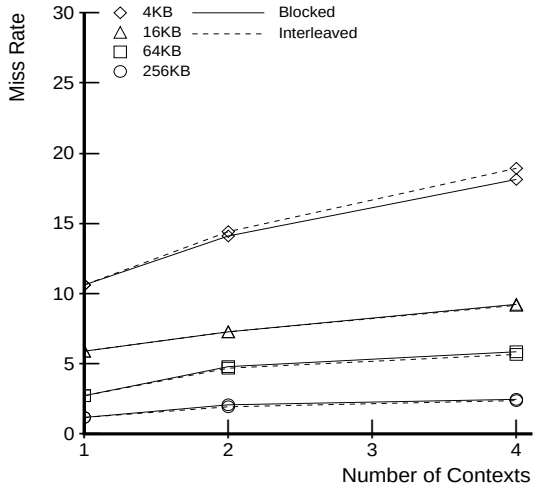
Figure 4.5: Data cache read miss rates for varying cache sizes.



(e) R0

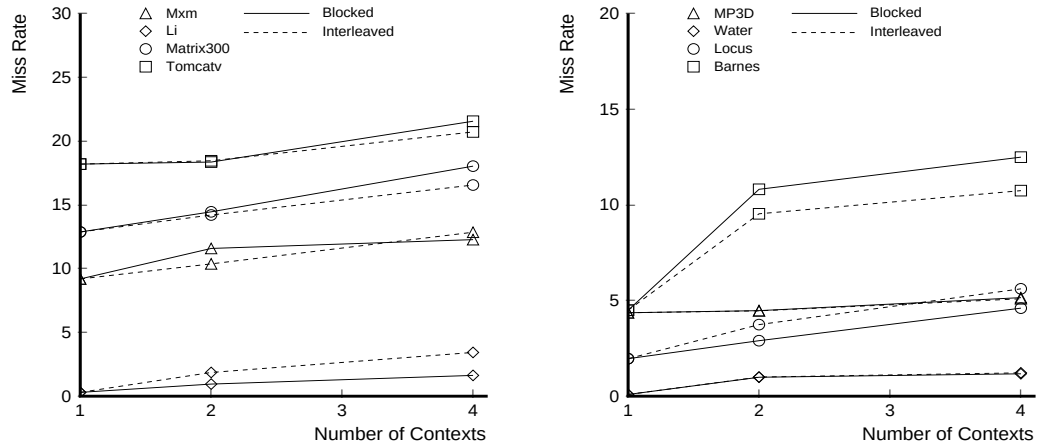


(f) R1



(g) SP

Figure 4.5: Data cache read miss rates for varying cache sizes (continued).

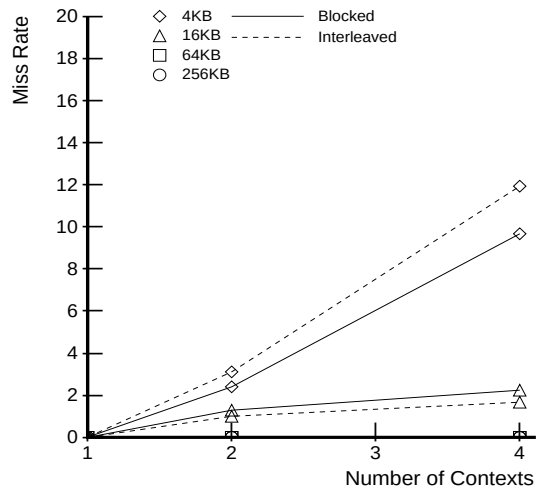


(a) R1 (b) SP

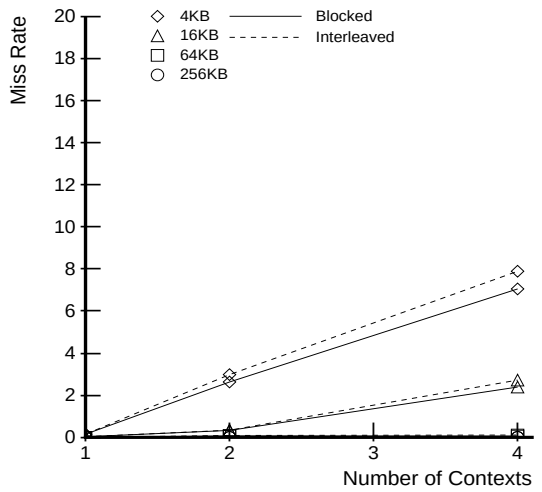
Figure 4.6: Data cache read miss rates by application for the 64 KByte cache.

However, the increase in interference did not outweigh the latency tolerating advantages of the multiple contexts.

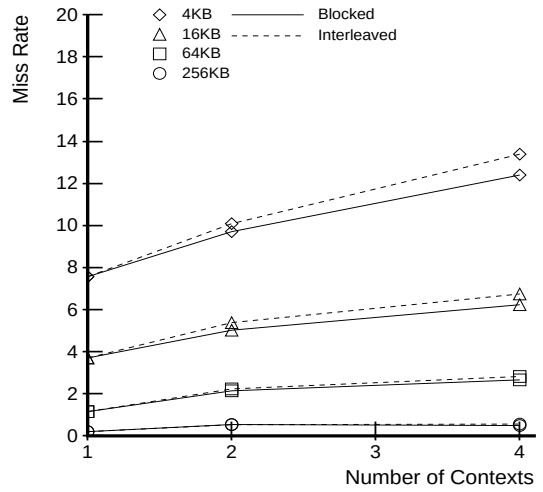
The interleaved scheme has been shown to substantially outperform the blocked for both multiprocessor and uniprocessor workloads. This is promising — we now need to examine the implementation costs of the interleaved scheme to determine if the additional costs are justified by the extra performance we have seen. The next chapter starts the implementation exploration by examining the costs of providing a cache capable of multiple outstanding requests, which is required by any multiple context scheme. Chapters 6 and 7 then examine the remaining cost for the blocked and interleaved schemes, respectively.



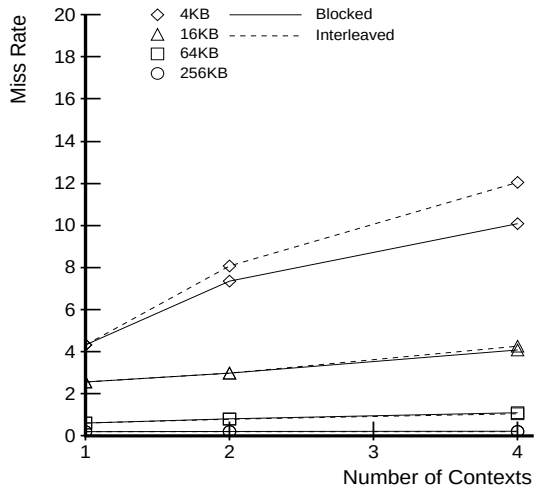
(a) DC



(b) DT

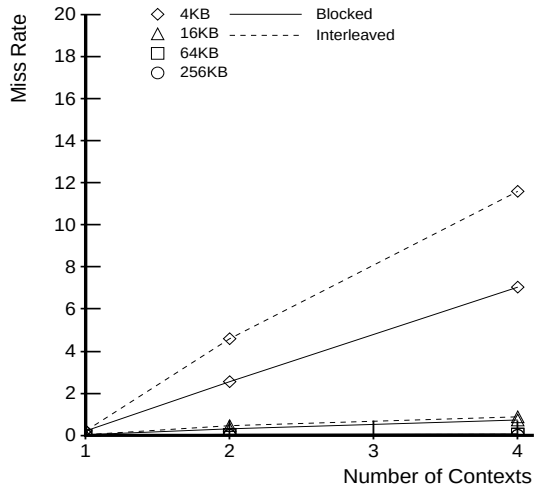


(c) IC

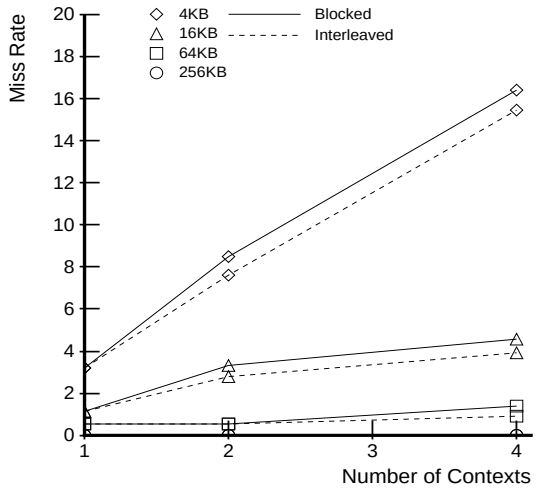


(d) FP

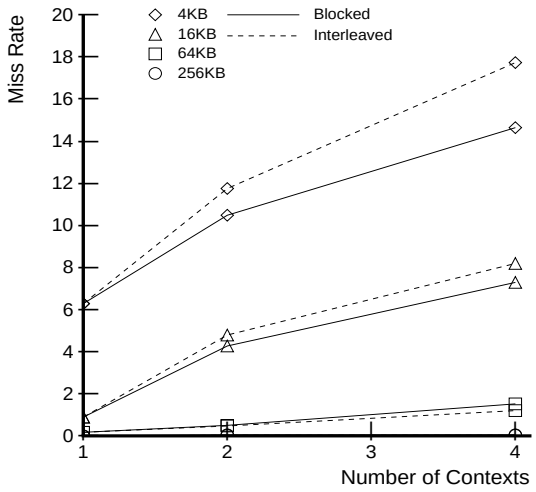
Figure 4.7: Instruction cache miss rates for varying cache sizes.



(e) R0



(f) R1



(g) SP

Figure 4.7: Instruction cache miss rates for varying cache sizes (continued).

Chapter 5

Lockup-free Cache Design

Chapters 3 and 4 have shown the performance of multiple-context processors to be promising, especially the interleaved approach. We now need to examine the implementation costs involved in building a multiple-context processor. One of the largest implementation complexity increase for multiple-context processors occurs in providing a cache capable of supporting multiple outstanding misses, or to use the prevailing terminology, a *lockup-free* cache. Lockup-free cache designs are more complex than standard *blocking* caches and it is important that this additional complexity does not translate into a large increase in implementation cost.

We start this chapter by examining previous lockup-free cache designs in Section 5.1. All previous lockup-free cache proposals track outstanding requests either in a set of transaction buffers [Kro81], or directly in the cache itself [LLG+90, Len92]. However, the tradeoffs between these two methods of tracking requests have not been explored in detail, and in Section 5.2 we examine this tradeoff for the four major methods of generating multiple outstanding requests. Based on this examination we then propose a lockup-free cache design which employs a pending state in the cache to allow requests to the same address to be merged. This pending state is also used to prevent issue of requests to the same cache line from different addresses. Section 5.2 also discusses how the deadlock issues which arise for lockup-free caches can be avoided. Deadlock can arise for a multiple-context processor due to pathological replacement patterns between the contexts, and we propose a mechanism called *adaptive stalling* to address this problem. In Section 5.3 we show that our lockup-free cache proposal provides good performance and that the impact of adaptive stalling on performance is minimal. We also show the importance of reducing the occupancy of replies for caches with larger line sizes. Finally, Section 5.4 summarizes the chapter.

5.1 Previous Work

One common feature of previous lockup-free cache designs is that they all provide some mechanism for tracking outstanding requests. Previous proposals can be split into two categories based on where outstanding requests are tracked. Most lockup-free caches have tracked outstanding requests in separate transaction buffers [Kro81, PBG+85, SD91, SDL91, KCA92]. These transaction buffers are often called MSHRs (miss information/status holding registers), as this was the term used by Kroft in the first published paper discussing lockup-free caches [Kro81]. Outstanding requests can also be tracked directly in the cache itself, as done in the remote access cache of the Stanford DASH [LLG+90, Len92].

In this section we first describe both Kroft's and the Stanford DASH lockup-free caches as representatives of the two approaches. We then discuss the limitations of the existing lockup-free cache proposals and describe how these limitations can be addressed.

5.1.1 Transaction Buffer Request Tracking

The seminal lockup-free cache design was presented by Kroft [Kro81]. This lockup-free cache was used in Control Data Corporation's Cyber 835, which was introduced in 1979. The same basic lockup-free cache design was also used in the Cyber 930 and 932 [Kro92].

To implement a lockup-free cache capable of supporting nonblocking loads and multiple outstanding writes, Kroft adds a number of miss information/status holding registers (MSHRs) and some associated logic. Each MSHR handles one or more misses to a single memory line.

Table 5.1: MSHR state entries.

State	Description
Cache Line Pointer	Pointer to cache line allocated for pending request
Request Address	Address of pending request
Unit Identification Tags (one per word)	Identifies requesting processor unit
Send-to-CPU Status (one per word)	Valid bits for Unit Identification Tags
Partial Write Codes (one per word)	Bit vector tracking partial writes to a word
Valid Indicator	Valid MSHR contents
Obsolete Indicator	Underlying cache line has been reallocated

Kroft's design allows virtually any combination of multiple outstanding requests to the same cache line, including requests with different addresses. Consequently, the MSHRs consist of a fair amount of state to handle these multiple outstanding requests, as shown in Table 5.1. First,

the MSHRs are designed for a set associative cache, where several cache lines can be allocation candidates for a given address. Therefore, the MSHR has a cache line pointer, which is used to keep track of the cache line to store the returning data. The request address is also stored to allow merging of subsequent requests to the same memory line. Each word (a word is the smallest unit of request) has a requesting unit identification tag and a send-to-CPU status bit. This is used to forward data to the CPU functional unit which made the request. Partial write codes are associated with each word to handle partial-word writes. There is also a valid indicator (to signify valid MSHR contents) and an obsolete indicator, whose purpose will be explained later. Kroft also has some extra fields to control the loading of return data from a buffer into the cache, and handle requests while loading the cache from this buffer. We will not discuss this buffer control further, as it is a side issue detailed in [Kro81].

The MSHRs are accessed in parallel with the cache. If the access hits in the cache, the normal cache hit actions take place. If the access misses in the cache, the operations that occur depend on the MSHR status. We first detail the operations that occur on a MSHR miss. For a miss, a free MSHR is allocated and initialized as follows:

1. Valid indicator is set.
2. Obsolete indicator is cleared.
3. Cache line pointer is loaded with cache index.
4. Request address is loaded.
5. Requested word has its send-to-CPU status bit set and requesting unit identifier loaded with the functional unit designator.
6. All other send-to-CPU status bits cleared.
7. The block of words is requested from the memory system.
8. If request was a store the data is written into the cache and the appropriate partial write bits are set.
9. All MSHRs with the same cache line pointer are purged.

The valid indicator is set and the obsolete indicator cleared to indicate that the MSHR now contains valid information. The cache line pointer is loaded in order to detect later requests

which map to the same cache line. The request address is loaded to be able to merge requests to the same address. The CPU requesting unit identifier is loaded and send-to-CPU status bit is set to allow the appropriate data from the eventual reply to be forwarded to the proper unit. Finally, the block of words is requested from the memory system and all MSHRs with the same cache line pointer are purged, as the cache line has been reallocated for this new request.

For evolutionary reasons, this MSHR purging was implemented using two methods, depending on whether the line is being reallocated for an address that has a previous request still outstanding [Kro92]. If a previous request for the same address is still outstanding, the obsolete bit is set to prevent multiple hits in the MSHRs (both the valid and obsolete bits are used in the MSHR hit detection determination). Otherwise, the MSHR was purged by simply setting all the partial write codes to ones (ensuring that the data would not be put into the cache upon return). The write bits can safely be set to all ones in this case because the cache line has been reallocated for a new address.

When the reply returns, the cache line pointer determines the cache line to load with the returning data. The partial write codes are used to avoid writing stale data into the cache. The send-to-CPU status bits and requesting unit identifiers are used to forward the replies to the waiting functional units.

A request for a cache line that is outstanding (a MSHR hit) can be merged with a previous outstanding request (thereby using the same MSHR) under several circumstances. First, all store operations to an outstanding request can be merged by writing the data into the cache and setting the appropriate partial write bits. Second, loads of a word that has not already been requested can use the same MSHR, as the CPU requesting unit identifier and send-to-CPU status bit of that word are available. Loads of a word already requested need to allocate a new MSHR, as there is only a single CPU requesting identifier per word. Finally, loads of a word completely written in the cache can simply read the data from the cache buffer.

A number of recent lockup-free cache designs have been built based on transaction buffers. Scheurich and Dubois [SD88, SD91] and Senstrom, Dahlgren, and Lundberg [SDL91] both present transaction buffer-based lockup-free caches for processors which support a relaxed memory consistency model. Kubiatowicz, Chaiken, and Agarwal [KCA92] present a transaction buffer-based lockup-free cache for a multiple-context processor, APRIL.

5.1.2 In-cache Request Tracking

Instead of requiring a separate set of transaction buffers, it is is possible to provide lockup-free support in the cache itself, as we did for the Remote Access Cache (RAC) of the DASH multiprocessor [LLG+90, Len92]. The RAC provides lockup-free cache support for a cluster of four processors in the DASH prototype. The state required to track the outstanding transactions is a part of each cache line. The RAC is a direct-mapped cache, with each line containing the information shown in Figure 5.1.

Figure 5.1: Format of each line in the DASH Remote Access Cache.

The tag contains the portion of the address needed to uniquely identify the address allocated to a particular RAC line. The cache state tracks the progress of a RAC transaction. The wait and conflict field are bit vectors to track the processors from a cluster waiting for the results of a RAC entry. The wait field contains the processors waiting for an outstanding request. The conflict field contains the processors with a request for a conflicting address which has mapped to the same line as a previous outstanding address. The count field tracks invalidation acknowledgements, allowing DASH to support both a processor and a release memory consistency model. The start and end bytes are used by the RAC for repeat of operations on partial cache-lines. The byte information was stored in the RAC because negative acknowledgements did not retain this information.

The RAC employs a large number of states to track the status of the many different flavors of outstanding requests available in DASH [Len92]. The basic flow of state transitions for a RAC entry is the following. The RAC snoops all processor requests which issue to the cluster bus. If the RAC entry for the address is free, it is allocated in a pending state, the request is issued to the memory system, and the processor has its bit set in the wait vector to indicate it is now waiting for the reply. If the RAC entry is already allocated for a compatible request[1], the processor's bit in the wait vector is simply set to indicate that it is also waiting for the reply. If the RAC entry is allocated for an incompatible request, the bit for the processor is set in the conflict field. Once the reply returns, the RAC entry transitions into a state that indicates that data has been received, but has not been given to the requesting processor(s). The processors in the wait field are then

[1]Compatible requests correspond to a read that encounters a read or read-exclusive pending, and a read-exclusive that encounters another read-exclusive pending.

freed and told to repeat their requests. Once the first waiting processor receives its response data from the RAC, the RAC frees any conflicting processors so they can repeat their request. The RAC entry then transitions to either an invalid state if the processor took ownership of the line or into a "soft" shared state if the processor did not take ownership. The "soft" shared state means that the RAC entry can be reallocated by a processor which needs the entry to process a remote request.

5.1.3 Discussion

Designing the RAC led to insights into the limitations of existing lockup-free cache proposals. First, previous designs have not evaluated the tradeoff between providing separate transaction buffers and tracking requests directly in the cache. Second, some of the transaction-buffer based proposals have a significant amount of complexity due to allowing multiple outstanding requests with different addresses to the same cache line. For example, because Kroft's MSHRs allowed multiple requests to the same cache line, the obsolete bit had to be added later in the design to handle the case where a cache line was allocated for one request, reallocated for a request to a different memory line, and then reallocated again for the first request, all before the reply to the the first request had returned [Kro92]. Simplifications resulting from placing restrictions on the allowable outstanding requests have not been explored in detail. Finally, existing proposals have generally targeted one specific method of generating multiple-outstanding requests. Kroft targeted a processor with nonblocking loads, Scheurich and Dubois a system with relaxed memory consistency, Kubiatowicz et. al. a multiple-context processor.

We start the following section by examining the requirements placed on a lockup-free cache by all four major methods of generating multiple outstanding requests: prefetch, relaxed memory consistency, non-blocking loads, and multiple contexts. This examination discusses where the outstanding requests generated by each method are best tracked: in separate transaction buffers, within the cache itself, or by a combination of the two. The following section also explores how limits can be placed on the outstanding requests to simplify the design.

5.2 Lockup-free Cache Design

In order to provide a framework for our lockup-free cache discussion, we assume a shared-memory multiprocessor where each node has the data cache hierarchy given in Figure 5.2. Since we are concentrating on high-performance processors, the cache hierarchy at each node is assumed to

consist of multiple levels of writeback caches. Caches closer to the processor are referred to as upper or nearer; caches closer to the network interface are referred to as lower or outer. No assumptions are made about the associativity of the caches at any level. To simplify the discussion, the same line size is used for all levels throughout the hierarchy.[2]

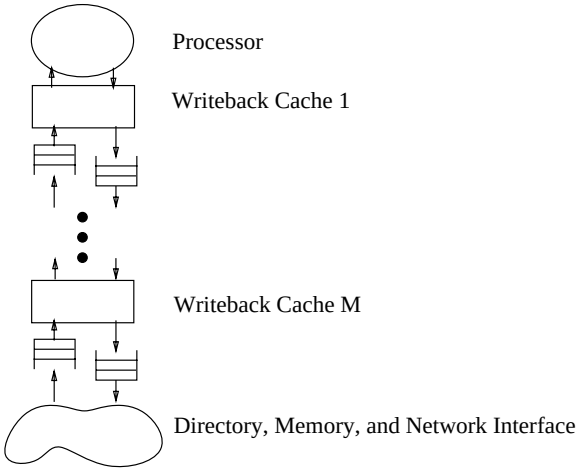


Figure 5.2: Data cache hierarchy.

Cache coherence is maintained between nodes through the use of a distributed, directory-based protocol [LLG+90, JLGS90, CKA91, Sim92, Web93]. To simplify our discussion, the hardware necessary to support both the directory-based protocol and track the completion of memory operations (for use in enforcing a relaxed memory consistency model) is assumed to exist outside the cache hierarchy.

While the directory maintains coherence between nodes, a separate coherence protocol needs to be developed to maintain coherence within the cache hierarchy. Appendix C describes an intra-hierarchy coherence protocol which we now summarize here. The protocol is based on the four cache states given in Table 5.2. The protocol employs two variants of the exclusive cache state because of the multiple levels of writeback caches. An exclusive copy is only up-to-date in one of the caches, and therefore the dirty state is used to tag that copy, while the stale state tags the other exclusive copies in the hierarchy.

The messages listed in Table 5.3 are sent between the caches to satisfy both processor requests and external coherence operations, and to maintain *inclusion* [BW88] between the cache levels. The function of most messages is self-explanatory, with the exception of flush and copyback.

[2]The common scenario where cache line size monotonically increases from inner to outer caches is a straightforward extension of our discussion covered in Appendix C.

Table 5.2: Cache states used to support the intra-hierarchy coherence protocol.

State	Description
D	exclusive, up-to-date copy (Dirty)
T	exclusive, out-of-date copy (sTale)
S	shared copy (Shared)
I	invalid (Invalid)

The flush request asks for exclusive data from the cache, requiring that the cache also invalidate its copy. The copyback request also asks for exclusive data from the cache, but allows the cache to keep a shared copy. The flush and copyback replies then return the exclusive data.

Table 5.3: Coherence messages.

Message	Description	Direction
$ExcRq$	Exclusive Request	From upper cache to lower cache
$ShdRq$	Shared Request	
$FshRp$	Flush Reply	
$CbkRp$	Copyback Reply	
$FshRq$	Flush Request	From lower cache to upper cache
$CbkRq$	Copyback Request	
$InvRq$	Invalidate Request	
$ExcRp$	Exclusive Reply	
$ShdRp$	Shared Reply	

The intra-hierarchy protocol handles processor requests by percolating them downward until they either encounter a cache which has the requested line in the proper state (shared or dirty for the shared request, dirty for the exclusive request) or are sent to the external interface. Replies to these processor requests are sent up the cache hierarchy, updating each cache as they progress towards the processor. Shared replies are loaded into each cache in the hierarchy in a shared state, while exclusive replies are loaded into all levels of the cache hierarchy except the innermost in the stale state. In the innermost cache exclusive data is loaded in the dirty state, as this will be the most up-to-date copy.

External flush and copyback requests percolate upward from the external interface, modifying the state of the cache lines at each level (changing the line to invalid for the flush, shared for the copyback request) until they encounter the dirty copy, at which point a reply is generated for the external interface. Invalidates simply traverse up the cache hierarchy, invalidating all shared lines

they encounter. Note that handling these coherence requests requires the caches to be dual-ported, as they will need to accept external requests while a processor request is outstanding.

Table 5.4: General cache coherence protocol.

Trans-action	**Cache State** D_H	D_M	T_H	T_M	S_H	S_M	I
$ShdRq$	$\rightarrow T$ $\perp ExcRp\uparrow$	$ShdRq\downarrow$	$\emptyset$	$ShdRq\downarrow$	$\perp ShdRp\uparrow$	$ShdRq\downarrow$	$ShdRq\downarrow$
$ShdRp$	$\emptyset$	$\rightarrow S$ $\perp FshRp\downarrow$ $\top ShdRp\uparrow$	$\emptyset$	$\rightarrow S$ $FshRq\uparrow$ $\top ShdRp\uparrow$	$\emptyset$	$\rightarrow S$ $InvRq\uparrow$ $\top ShdRp\uparrow$	$\rightarrow S$ $\top ShdRp\uparrow$
$ExcRq$	$\rightarrow T$ $\perp ExcRp\uparrow$	$ExcRq\downarrow$	$\emptyset$	$ExcRq\downarrow$	$ExcRq\downarrow$	$ExcRq\downarrow$	$ExcRq\downarrow$
$ExcRp$	$\emptyset$	$\rightarrow T$ $\perp FshRp\downarrow$ $\top ExcRp\uparrow$	$\emptyset$	$\rightarrow T$ $FshRq\uparrow$ $\top ExcRp\uparrow$	$\rightarrow T$ $\top ExcRp\uparrow$	$\rightarrow T$ $InvRq\uparrow$ $\top ExcRp\uparrow$	$\rightarrow T$ $\top ExcRp\uparrow$
$FshRq$	$\rightarrow I$ $\perp FshRp\downarrow$		$\rightarrow I$ $FshRq\uparrow$		$\emptyset$		
$FshRp$	$\emptyset$	$FshRp\downarrow$	$\rightarrow D$ $\top$	$FshRp\downarrow$	$\top CbkRp\downarrow$	$FshRp\downarrow$	$FshRp\downarrow$
$CbkRq$	$\rightarrow S$ $\perp CbkRp\downarrow$		$\rightarrow S$ $CbkRq\uparrow$		$\emptyset$		
$CbkRp$	$\emptyset$	$FshRp\downarrow$	$\emptyset$	$FshRp\downarrow$	$\top CbkRp\downarrow$	$FshRp\downarrow$	$FshRp\downarrow$
$InvRq$	$\emptyset$		$\emptyset$		$\rightarrow I$ $InvRq\uparrow$		

$\rightarrow$ state transition
$\downarrow$ message to lower-level cache
$\uparrow$ message to upper-level cache
$\perp$ read data from cache into message
$\top$ write data from message into cache
$\emptyset$ error (should not occur)

The protocol for all caches except the innermost is given in Table 5.4. The protocol for the primary cache is given in Appendix C and differs from that of the other caches in that it loads exclusive data in the dirty state and never sends requests to upper level caches.

Messages are listed along the left side, the cache states along the top. The cache state subscripts H and M are used to signify a hit or miss respectively. A detailed discussion of all the state transitions of Table 5.4 can be found in Appendix C; we will discuss one transition (an exclusive reply that encounters the cache line allocated in the stale state for a different address)

here as an example. This transition is located under the T_M column, in the $Exc\ Rp$ row. The table specifies that (a) the cache transition to the stale state (now with a new address tag), (b) the new data be written into the cache line, and (c) two messages be generated for the upper-level cache. First, a flush request needs to be sent for the current cache address to maintain inclusion. Second, the exclusive reply is sent to the upper-level cache to update that cache level and eventually be forwarded to the processor.

5.2.1 Lockup-free Requirements

Now that we have established our base coherent cache hierarchy, we can examine in detail the requirements placed on the cache hierarchy to support the multiple outstanding requests resulting from prefetch, relaxed memory consistency, nonblocking loads, and multiple contexts. All methods of generating multiple requests require the lockup-free cache to handle requests from the processor simultaneous with replies from outside the processor. Since coherent caches must already handle memory requests from the processor side and coherence requests from the network side, the logic to arbitrate between messages from both sides is already in place, and can be easily extended to handle these multiple replies. Beyond this basic requirement, each latency tolerance approach presents additional demands on the lockup-free cache.

Requirements for Prefetch

Prefetching requires the cache hierarchy to handle multiple outstanding read and/or read-exclusive requests. Prefetching has several properties which help to keep the design of the lockup-free cache simple. First, since prefetch requests are only performance hints, they can be dropped as needed. Second, prefetch operations only specify data movement, they do require any data to be modified. Third, the only correctness requirement for prefetching is that its addition does not cause any violation of either coherence or memory consistency. Finally, the impact on the cache hardware of the basic prefetching performance constraints is rather modest. Ideally, prefetches should be dropped when the data is already in the cache hierarchy or when the memory system becomes overloaded. In addition, combinations of prefetch and normal requests or multiple prefetches to the same cache line should be merged whenever possible.

With these loose constraints the support required for handling multiple prefetch requests is rather small. In the simplest case, prefetches can simply be sent down the cache hierarchy, checking each cache as they pass. If the prefetch hits in any cache on the way out, it is dropped.

Otherwise the prefetch is issued to the external interface and will either return with data or be dropped externally.

However, this simple approach does have a few drawbacks. First, multiple prefetches/memory requests to the same line within the cache hierarchy are not merged. Since prefetches are likely to be generated either by performance-hungry programmers or sophisticated compilers, multiple prefetches to the same cache line should be infrequent, however merging of a prefetch and a regular memory reference is desirable, since this situation will arise whenever the prefetch could not be issued far enough in advance to completely tolerate the memory latency. If the two requests are not merged, extra cache bandwidth is occupied by the prefetch/request combination. In addition, allowing multiple prefetches and/or normal requests to the same line to be simultaneously in flight in the network does cause some additional races for the external coherence protocol, which it must handle correctly. Second, the "unsolicited" replies generated by the prefetches may return to find all possible cache lines which can hold the data are allocated to pending requests for other addresses. The prefetch data must then either be dropped (for a shared copy) or be treated as if it was just replaced from the cache (for an exclusive copy).

By tracking outstanding prefetches, either in a separate prefetch buffer [GHG+91] or in the cache itself [Len92], multiple prefetch/memory requests to the same cache line can be merged within the cache hierarchy and a space is always reserved for the prefetch reply. Prefetches can be tracked very simply in the cache itself by adding a pending cache state to the primary cache. When a prefetch misses in the primary cache it allocates a cache line in the pending state and sends the prefetch to the next cache level. If all possible allocation targets for the prefetch are already pending for other address, the prefetch can either be dropped or can cause the processor to stall until one of the allocation targets frees up. Multiple prefetches to the same cache line or a prefetch followed by a normal memory request are detected as pending hits, allowing the second request to be merged with the first by simply dropping the second request. Note that prefetches do not need to allocate pending cache lines in the cache levels beyond the primary, because allocation in the primary cache (where the smallest number of possible allocation targets for any given address exists) implies that an allocation target will exist for the returning prefetch data in all outer caches.

Tracking prefetch requests does have the disadvantage that prefetches can no longer be silently dropped. Dropping a prefetch requires a negative acknowledgement to be returned so the cache line reserved for the prefetch can be freed. In addition, if a normal request cannot allocate a cache line due to all possible allocation targets being used for prefetching, the processor is

forced to stall. These disadvantages seem relatively small and are most likely outweighed by the advantages of request merging and guaranteed line allocation for the returning data.

Requirements for Relaxed Memory Consistency

The requirements on the lockup-free cache to support relaxed memory consistency are very different from those for prefetching. Relaxed memory consistency gains most of its performance benefits by overlapping multiple outstanding writes.[3] Allowing multiple outstanding writes poses a larger problem than multiple outstanding prefetches because each write consists not only of an exclusive request for the cache line, but also specifies new data which needs to be merged into the reply. Supporting multiple outstanding writes entails the following:

- Merging multiple writes to the same line.
- Handling writes to a cache line which has a read outstanding.
- Detecting a read request to already written data, and potentially forwarding the written data.
- Buffering the write data until it can be merged with the reply.

One common approach to supporting relaxed memory consistency for a system consisting of a writethrough cache backed by a writeback cache is to add a write buffer between the writethrough and the writeback cache [BJS88, SD91, DBCÖ92]. This buffer holds the write data for a pending write miss in addition to the standard service of providing buffering between the caches. This solution allows multiple writes to be outstanding with respect to the processor without actually supporting multiple requests outside the cache hierarchy, however, it does not take full advantage of the relaxed consistency model, since the first write miss stalls all subsequent memory requests to the secondary, writeback cache. Read bypassing can be added to the write buffer to address this problem [GGH91]. While read bypassing does increase performance, it greatly complicates the once simple buffer, since now each buffer entry must have address compare logic to be able to compare the address of all writes in the buffer against any read. Providing a lockup-free cache which supports relaxed memory consistency has been found to result in much better performance than a write buffer with read bypassing [GGH91], and in the following paragraphs we argue that it is also a lower cost solution.

[3] Relaxed memory consistency is also necessary to get maximum benefits from nonblocking loads [GGH92]. A relaxed memory consistency model is assumed when discussing the cache requirements for nonblocking loads.

Adding read bypassing to the write buffer increases the implementation cost for a number of reasons. First, as already mentioned, each buffer entry must have an associated address comparator to be able to compare the address of the buffered write against any incoming reads. Second, the datapath for allowing the read to bypass the write buffer must exist, including some means of buffering the read in case the cache is busy. Finally, the write buffer itself must be made relatively deep, to hold all the writes which will backup while the first write miss is being serviced. Summed up, this is a fair amount of additional complexity.

The naive approach to supporting multiple writes directly in the lockup-free cache is to add subblock valid bits to each cache line, and store the write data in the cache until the reply returns. Upon a write miss, the cache goes to a pending state, while also setting the valid bit(s) of the word(s) being stored. Merging of multiple writes is done by simply writing the data into the cache and setting the appropriate valid bits. Forwarding written data to subsequent reads occurs by signaling a cache hit for reads to valid data in a pending cache line. Unfortunately, providing subblock valid bits in the cache is quite expensive, and most of the valid bits will not contain useful information, as they are only needed while a write miss is outstanding.

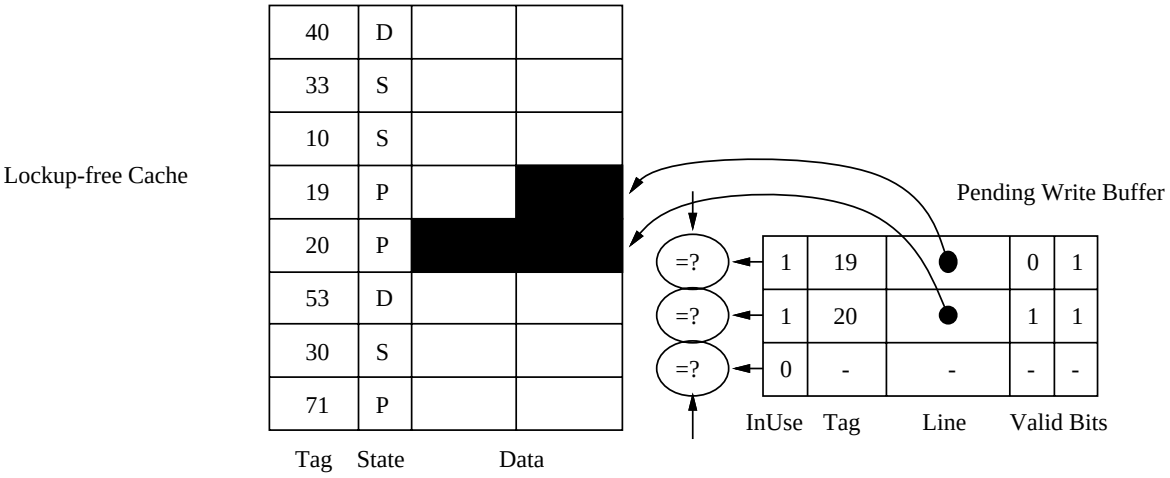


Figure 5.3: Lockup-free cache supporting relaxed memory consistency using a pending-write buffer.

A better approach is to keep only the data from the write in the cache, maintaining the valid bits in a separate buffer, as was done in Kroft's design [Kro81]. This *pending-write* buffer can be built as a small fully-associative cache, with the tag pointing to the cache line for which pending-write entry is allocated. Figure 5.3 shows a lockup-free cache with two-word cache lines and its pending-write buffer. The figure shows the cache handling outstanding requests for

three cache lines (marked as pending by a P in the state field). Two of the outstanding requests (to the lines tagged 19 and 20) were caused by three stores, as signified by the darkened data blocks. The lockup-free cache tracks which of these pending words contains valid data through the pending-write buffer. Multiple writes to the same cache line are merged by simply writing the data into the cache and setting the appropriate valid bits in the pending-write buffer. Forwarding of written data to subsequent reads is done by including the pending-write buffer status when making the cache hit determination. Finally, the pending-write buffer is used to prevent the store data in the cache from being overwritten by the exclusive reply data (the cache will need the ability to only fill in the invalid portions of the line). Once the reply has been merged into the cache, the corresponding pending-write entry can be freed for further use.

Comparing the pending-write buffer to a write buffer with bypassing, the pending-write buffer will likely be smaller and simpler to implement. Both the write buffer and pending-write buffer require address comparator logic for each entry, however, the pending-write buffer will be smaller than the write buffer since it only holds state for write misses, whereas the bypassing write buffer holds the address and data for the original write miss, and all subsequent writes, regardless of whether they would be hits or misses. Not only is the pending-write buffer likely to be lower cost, but it has higher performance potential, since it truly supports multiple outstanding writes. The one advantage of the conventional write buffer is that it can be integrated with the outgoing request buffer. However, this integration comes at the expensive of making each buffer entry much more expensive and requiring more entries than would be needed for simple buffering.

Requirements for Nonblocking Loads

Nonblocking loads again have differing requirements than either prefetching or relaxed memory consistency. Nonblocking loads require: (a) the cache to be capable of handling multiple outstanding loads, (b) the loads to be merged with any other outstanding requests as appropriate, and (c) some mechanism for forwarding the word requested by the nonblocking load to the appropriate register when the load reply returns.

Handling this forwarding of load data is best done by separate transaction buffers next to the primary cache which hold the address/register pairs for outstanding requests. When a reply returns, it checks these buffers and forwards the data to the appropriate registers. A pending state in the primary cache is also needed to merge requests and prevent a request from being issued if it cannot allocate a cache line.

Requirements for Multiple Contexts

Finally, supporting multiple contexts requires that each context be allowed a single outstanding request. Context switches are likely to be performed on both read and write misses for a processor operating under sequential consistency, while a processor operating under relaxed memory consistency is likely to switch only on read misses. Since the processor will repeat its load or store after the reply has returned, handling multiple stores generated by a sequentially-consistent multiple-context processor is simpler than handling the multiple stores of relaxed consistency, as the store data does not need to be kept in the cache. As with the other latency tolerance mechanisms, the lockup-free cache will achieve the best performance when compatible requests are merged. A pending state in the primary cache can handle this merging and also prevent a request from being issued until it can allocate a cache line.

The interleaved scheme places an additional requirement on the lockup-free cache. A reply to a request which caused a context switch needs to be detected in order to reenable the context. Detecting this can be done in a fashion similar to the forwarding of replies for the nonblocking loads by providing a set of transaction buffers next to the primary cache holding the address/context number pairs.

5.2.2 Lockup-free Cache Proposal

These differing methods of generating multiple requests have placed several requirements on the primary cache:

1. Outstanding prefetches, loads, and stores need to be tracked in the cache to support request merging and to provide a cache line to place returning data.

2. Write data can be stored in the cache, allowing the write to complete immediately under a relaxed consistency model, by providing separate transaction buffers (the pending-write buffer) that maintain the subblock valid bits.

3. Separate transaction buffers are needed to track nonblocking loads to allow the data to be forwarded to the proper processor register.

4. Separate transaction buffers are needed for an interleaved multiple-context processor to reenable a context when its reply returns.

Table 5.5: Lockup-free cache coherence protocol (uppermost writeback cache).

Trans-action	**Cache State** W_H	D_H	D_M	P_H	P_M	S_H	S_M	I
$Shd\ Rq$	$\perp\ Ld\ Rp\ \uparrow$	$\perp\ Ld\ Rp\ \uparrow$	$\rightarrow P$ $\perp\ Fsh\ Rp\ \downarrow$ $Shd\ Rq\ \downarrow$ $\dagger$	$\downarrow\ \dagger$	$\dagger$	$\perp\ Ld\ Rp\ \uparrow$	$\rightarrow P$ $Shd\ Rq\ \downarrow$ $\dagger$	$\rightarrow P$ $Shd\ Rq\ \downarrow$ $\dagger$
$Psh\ Rq$			$\rightarrow P$ $\perp\ Fsh\ Rp\ \downarrow$ $Shd\ Rq\ \downarrow$	$\downarrow$			$\rightarrow P$ $Shd\ Rq\ \downarrow$	$\rightarrow P$ $Shd\ Rq\ \downarrow$
$Shd\ Rp$	$Exc\ Rq\ \downarrow$	$\emptyset$	$\emptyset$	$\rightarrow S$ $\top$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$Exc\ Rq$		$\top$	$\rightarrow P, \Rightarrow V$ $Fsh\ Rp\ \downarrow$ $Exc\ Rq\ \downarrow$ $\top$	$\Rightarrow V$ $\top$	$\dagger$	$\rightarrow P,\ \Rightarrow V$ $Exc\ Rq\ \downarrow$ $\top$	$\rightarrow P,\ \Rightarrow V$ $Exc\ Rq\ \downarrow$ $\top$	$\rightarrow P,\ \Rightarrow V$ $Exc\ Rq\ \downarrow$ $\top$
$Pex\ Rq$			$\rightarrow P$ $\perp\ Fsh\ Rp\ \downarrow$ $Exc\ Rq\ \downarrow$	$\downarrow$		$\rightarrow P$ $Exc\ Rq\ \downarrow$	$\rightarrow P$ $Exc\ Rq\ \downarrow$	$\rightarrow P$ $Exc\ Rq\ \downarrow$
$Exc\ Rp$	$\Rightarrow I$	$\emptyset$	$\emptyset$	$\rightarrow D$ $\top$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$Fsh\ Rq$		$\rightarrow I$ $\perp\ Fsh\ Rp\ \downarrow$				$\emptyset$		
$Cbk\ Rq$		$\rightarrow S$ $\perp\ Cbk\ Rp\ \downarrow$				$\emptyset$		
$Inv\ Rq$		$\emptyset$				$\rightarrow I$		

$\rightarrow$ cache state transition
$\Rightarrow$ pending-write buffer state transition
$\downarrow$ message to lower-level cache
$\uparrow$ message to processor
$\perp$ read data from cache into message
$\top$ write data from message into cache
$\dagger$ context switch / stall
$\emptyset$ error (should not occur)

In this section, we use these requirements to extend our base intra-hierarchy coherence protocol to support prefetch, relaxed memory consistency, and multiple contexts. The protocol we present here was used for the performance simulations of Chapters 3 and 4. Extending this protocol to support nonblocking loads is straightforward (and was done to simulate the effects of combining multiple contexts and nonblocking loads in Section 3.4.2). The protocol uses a pending state in the primary cache to allowing merging of compatible requests and to prevent a

request from being issued which cannot allocate a cache line. Since the pending state guarantees that a single request is outstanding per cache line, the protocol for all caches except the primary cache remains the same as the blocking cache protocol given in Table 5.4. The protocol for the primary cache is given in Table 5.5. P signifies the pending state, W_H signifies a hit in the pending-write buffer, $PexRq$ and $PshRq$ are prefetch exclusive and prefetch shared requests, and $LdRp$ is the data returned from the cache for the load (ShdRq) operation.

Since the primary cache needs to allocate the cache line in a pending state upon a cache miss, the old line is replaced immediately, not when the reply returns as for the blocking protocol. When a processor makes a request which conflicts with a pending line, the processor either context switches without issuing the request or stalls until the pending line is freed (the conditions under which the processor stalls will be given in Section 5.2.4). These conflicts are expected to be infrequent, so the performance loss due to the extra switching or stalling should be small, and will be explored in Section 5.3. Note that the protocol supports merging requests to the same address, including merging of a read followed by a write request. For the read followed by a write, the write request actually stores its data into the cache and allocates a pending-write buffer entry. If a shared reply is returned for the read request, the cache then generates an exclusive request. Beyond these conflicts and merges, the protocol behaves as would be expected. Context switching occurs on read misses, and the pending-write buffer is used to hold the valid bits for writes while their requests are outstanding.

In addition to impacting the protocol for the primary cache, supporting multiple outstanding requests also introduces two situations where forward progress can be violated. The first involves the finite buffering between the caches. Deadlock arising due to limited buffering can occur for any lockup-free cache design which has multiple levels of writeback caches. In contrast, the second deadlock issue arises solely due to multiple contexts sharing the cache. We briefly discuss buffering deadlock in the following section. Then, in Section 5.2.4, deadlock issues specific to multiple-context processors are examined.

5.2.3 Buffer Deadlock

Deadlock can arise due to the finite buffering between writeback caches. For example, in Figure 5.4, the processor has issued a shared request for line A which missed in the primary (writeback) cache, but has hit in the secondary cache. However, there is no incoming buffer space to hold the shared reply, so the shared request cannot be processed. The flush requests which are preventing the shared request from being processed are waiting for buffer space in the

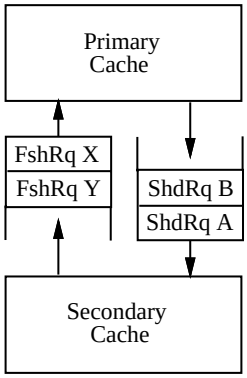


Figure 5.4: Buffer deadlock example.

outgoing buffer. Unfortunately, space in the outgoing buffer can only be generated by processing the original shared request. This circular dependency has resulted in deadlock.

There are a large number of solutions to this problem, several of which are discussed in Appendix D. The simplest solution is to only use a single level of writeback caches, as deadlocks only arise due to circular dependencies between writeback caches. If multiple levels of writeback caches are desired, a range of solutions exist — from statically limiting the total number of requests in the cache hierarchy from both the processor and external interface to dynamically limiting the requests between any two levels of the cache hierarchy using the Banker's algorithm [Tan86]. Since most systems with multiple writeback caches employ only two levels, placing static limits on the total number of requests from both the processor and the external interface is very straightforward and will give good performance as long as the limits are not too small.

Buffer deadlock can occur for any cache hierarchy with multiple levels of writeback caches. For lockup-free caches supporting multiple contexts, an additional source of potential deadlock arises due to the use of split-phase memory transactions.

5.2.4 Multiple-Context Deadlock

Many multiple-context processors introduce two-phase memory transactions to the memory system design. A two-phase memory operation consists of a *pending* phase followed by a *waiting* phase. The phases are bounded by three memory operations: request, response, and completion. The request operation occurs when a memory request is issued from the cache, and marks the

start of the pending phase. The response operation occurs when the reply returns from the memory system and is put in the cache. The response signals the end of the pending phase and the start of waiting phase. Finally, the completion operation occurs when the processor reissues the original request and receives the data from the cache, ending the waiting phase. However, since the cache may be satisfying other requests during the waiting phase, it is possible that the cache line could be replaced before the completion operation can be issued. For example, two requests could map to the same cache line, and if the second request occurs between the response and completion phases, the first request will be replaced from the cache.

Since operations are vulnerable to cache replacement during the waiting phase, this phase has also been referred to as the *window of vulnerability* [KCA92]. Under pathological conditions, replacement during the window of vulnerability can occur indefinitely, leading to system deadlock. A lockup-free cache supporting multiple contexts must be designed to address this deadlock. Since this deadlock arises due to multi-phase memory transactions, we will start by discussing support of multiple contexts with only single-phase memory transactions. We will then address the deadlock issues for multiple context schemes with multi-phase memory transactions.

Avoiding Multi-phase Memory Transactions

There are several ways to implement multiple-context processors without using two-phase memory transactions. The first method removes the window of vulnerability by collapsing the response and completion phases. This can be done by updating the requesting context's registers when a reply returns (to do this may require the use of valid bits on the registers, or a distinct load buffer per context). Since the register is loaded when the reply returns, the response and completion operations have been merged into a single operation, removing the waiting phase.

A second solution is to implement multiple contexts by using a combination of prefetch operations, standard blocking loads and stores, and an explicit context switch instruction. Multiple contexts are supported on such a processor by replacing all loads and stores (or only loads and stores that are expected to miss) with a prefetch followed by an explicit context switch and then finally the actual load or store.[4] The performance of a multiple context processor using this compiler-controlled context switching was evaluated in [BR92], and showed promise.

[4]To reduce the number of explicit context switches required, the compiler can group together several loads and stores and insert a single switch instruction between them and instructions which depend upon their completion.

However, the jury is still out on the completely compiler-controlled multiple-context processor. Since many multiple-context designs (including ours) assume two-phase memory transactions [WG89b, ALKK90, LGH92], we need to explore the issues involved in preventing deadlock in the presence of the window of vulnerability.

Handling Multi-phase Memory Transactions

Work by the Alewife group at MIT [KCA92] presents four distinct types of thrashing during the window of vulnerability which can result in deadlock. These four types are: (a) *invalidation thrashing*, (b) *replacement thrashing*, (c) *instruction-data thrashing*, and (d) *high-availability interrupt (HAI) thrashing*. We now discuss each of these four deadlock situations in more detail. We start with instruction-data and HAI thrashing, as they cannot occur under our assumptions, and then turn to invalidation and replacement thrashing.

Instruction-data thrashing occurs only for processors with a combined instruction and data cache. If an instruction and its data map to the same cache line, it is possible to deadlock, as the data and instruction lines repeatedly replace each other in the cache, as shown in Figure 5.5. Time advances from left to right in the figure. The lower bar represents the different contexts executing on the processor. The arrows above this bar show outgoing (up arrow) and incoming (down arrow) messages. The window of vulnerability is shown in gray. The processor starts with context A executing instruction Y, which performs a load of memory location X. X is not in the cache, so the processor issues a read request for X and performs a context switch. Eventually, the read for X completes, knocking instruction Y out of the cache. When the processor restarts context A, the processor fetches Y (IF), and stalls until the instruction fetch is complete (IR). However, the instruction has knocked X out of the cache, so the processor again issues a load for X and performs a context switch. This cycle will continue indefinitely. Luckily, modern processors have separate instruction and data caches, in order to support their bandwidth requirements of one to four instructions and one data reference per cycle. With separate instruction and data caches, instruction-data thrashing cannot arise.[5]

High-availability interrupt thrashing occurs when a processor supports high priority interrupts to handle asynchronous events, some of which may be necessary for an outstanding memory operation to complete. For example, in Alewife, HAI are used to deal with buffers filling during an outstanding request [KCA92]. When the interrupt code for the HAI is run during the window

[5]Instruction-data thrashing is possible if the separate instruction and data caches are backed by a unified cache which enforces subset property before returning data to the upper level caches.

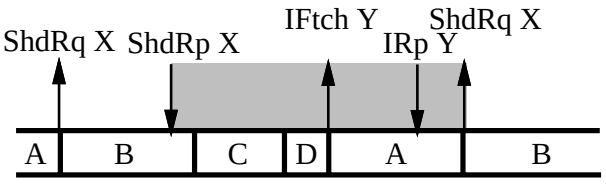


Figure 5.5: Instruction-data thrashing scenario.

of vulnerability the cache line may be removed due to cache conflicts with code or data used by the HAI. Under pathological conditions this can occur indefinitely, leading to deadlock. Note that the HAI problem is not confined to multiple-context processors. Since HAIs can interrupt a processor while it is stalled on a memory operation, the addition of HAIs to a single-context processor turns its memory accesses into two-phase operations. The simplest way to avoid HAI deadlock is to not implement high-availability interrupts, as we have assumed with our architecture. However, assuming HAIs need to be supported, HAI deadlock must be addressed, and we will discuss a few possible methods for handling HAIs at the end of this section. These methods tend to be expensive and therefore the decision to support HAIs should be weighed carefully.

Invalidation thrashing occurs when two or more processors contend for the same memory line, and at least one of the processors wishes to perform a write. Figure 5.6 illustrates one possible invalidation thrashing scenario. Processors 1 and 2 are contending over the same memory line X. Context A on processor 1 wishes to read the line, while context A on processor 2 wishes to write the line. First, processor 1 makes a read request (ShdRq), and performs a context switch. At a later point, the shared reply (ShdRp) returns, but context A no longer is executing on the processor. We have now entered the window of vulnerability for the read request on processor 1. While in the window of vulnerability, context A on processor 2 does a write (ExcRq) to the same memory line and context switches. This write invalidates the line X from processor 1 and eventually returns an exclusive reply (ExcRp) to processor 2. However, context A is not executing on processor 2, and the window of vulnerability starts on processor 2. Before context A can restart on processor 2, context A on processor 1 restarts and repeats its read request, which removes the exclusive copy from the cache in processor 2. Under pathological conditions, this sequence of events could continue indefinitely, leading to deadlock.

There are several simple solutions to this problem, all of which involve closing the window of vulnerability for write operations. The first solution is to stall on write misses. This would

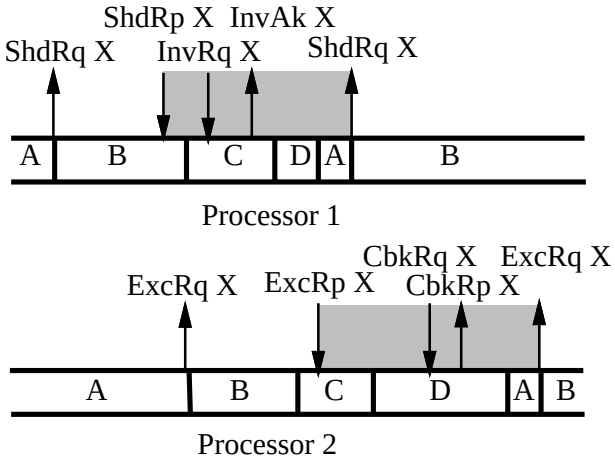


Figure 5.6: Invalidation thrashing scenario.

obviously remove the window of vulnerability. However, for a multiple-context processor operating under the strong consistency model, context switching can be an important mechanism for tolerating write miss latency and stalling on write misses can have a significant impact on performance.

A better solution is to remove the window of vulnerability by ensuring that the write completes atomically with the exclusive reply being received. Our lockup-free proposal guarantees this by storing the write data in the cache line and atomically merging the written data with the exclusive reply. This solution has a higher performance potential, because the processor does not have to switch on write misses for which a cache line can be allocated (it still has to switch or stall if the line cannot be allocated). However, this solution does have a hardware cost, as a pending-write buffer must store the valid status of the cache line until it can be merged in with the exclusive reply. Due to this hardware cost, we will not rule out cache designs for which invalidation thrashing is possible when developing our deadlock handling solution.

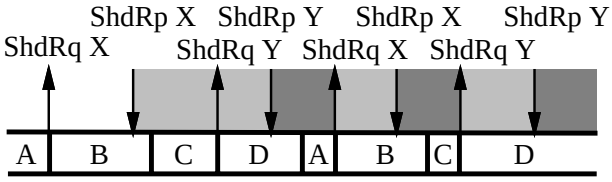


Figure 5.7: Replacement thrashing scenario.

Replacement thrashing is the remaining cause of deadlock for multiple-context processors.

An example of replacement thrashing resulting in deadlock is given in Figure 5.7. Due to limited associativity in the cache, lines X and Y map to the same cache line. Contexts A and C are requesting copies of X and Y, respectively. Figure 5.7 shows a scenario where they repeatedly knock each other out of the cache, resulting in livelock. This livelock will turn into deadlock when the other processes in the multiprocessor cannot make forward progress without the results from contexts A and C.

Adaptive Stalling

To address both invalidate and replacement thrashing, we have developed a deadlock detection and breaking scheme called *adaptive stalling*. Under adaptive stalling, the processor detects a thrashing situation, and backs off to a stalling method of memory access when thrashing is encountered. Since stalling on a memory access closes the window of vulnerability, the deadlock cycle is broken. The key to adaptive stalling is that the processor backs off to a single-phase memory transaction whenever it detects the potential for deadlock.

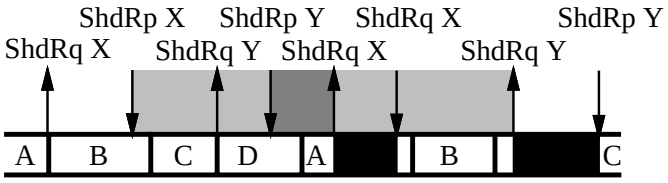


Figure 5.8: Adaptive stalling handling replacement thrashing.

Handling the replacement thrashing scenario using adaptive stalling is shown in Figure 5.8. The black areas on the context portion of the time line show the processor stalling in order to break the potential deadlock. When context A retries its request for X and misses in the cache, it reissues the request for X, but stalls rather than switches. Thus A will complete its request for X when the reply returns. When context C reissues its request for Y, it also stalls until the reply for Y returns.

Figure 5.9 shows invalidation thrashing being handled by adaptive stalling. When context A on processor 1 reissues its shared request, it stalls until the shared reply returns. Context A on processor 2 behaves similarly when it repeats its write request.

Adaptive stalling also handles instruction-data thrashing deadlock, as shown in Figure 5.10. Adaptive stalling can also handle HAIs, if a HAI can never occur when a processor is stalled on memory. However, if a HAI can interrupt a stalled processor, the adaptive stalling scheme cannot

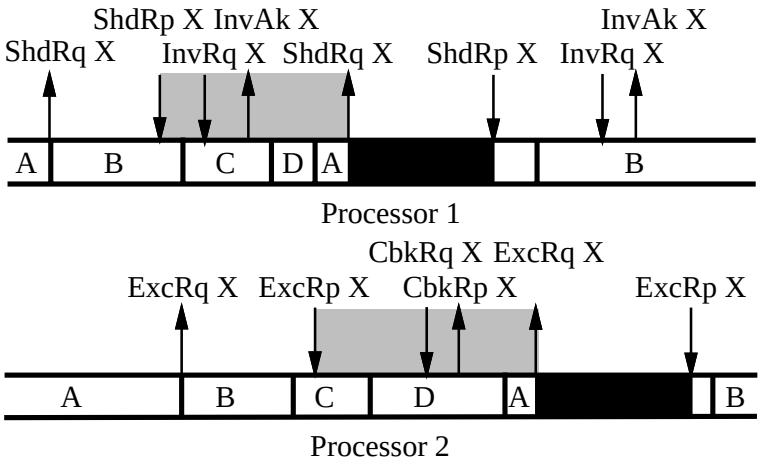


Figure 5.9: Adaptive stalling handling invalidation thrashing.

guarantee forward progress as it no longer can close the window of vulnerability by stalling.

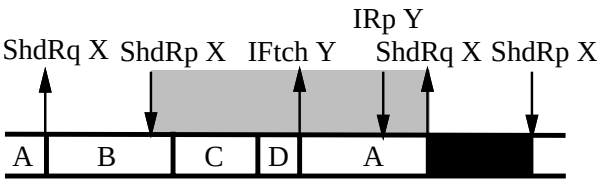


Figure 5.10: Adaptive stalling handling instruction-data thrashing.

Detection of potential deadlock must be simple for adaptive stalling to be easily implementable. Luckily an easy detection method exists. When a context that had been switched out due to a cache miss restarts, the first instruction it will issue is the memory reference which caused the cache miss. If the data item is not in the cache for this memory reference repeat, instead of switching again, the processor stalls until the memory operation completes. Implementation of this scheme only requires that the retry of a memory operation be identifiable and that the cache controller recognize this retry so that it can stall instead of switch. As we will see in Chapters 6 and 7, identifying this second memory access is very straightforward.

The adaptive stalling scheme has the advantage that most of the time, when the processor is not thrashing, there are no limitations placed on the outstanding requests (either in number of requests or ordering). Only when there is thrashing does the processor pay a performance penalty. As we will see in Section 5.3, this thrashing is infrequent, making the performance cost of adaptive stalling minimal.

Related Work

The *thrashwait* scheme [KCA92], developed at MIT simultaneously with our development of adaptive stalling, employs mechanisms very similar to adaptive stalling. The main difference is that they keep track of whether a request is being repeated in the cache, as implementing thrashwait by tagging repeated instructions would have required additional modifications to the Alewife processor, APRIL [ALKK90]. Their in-cache tracking is not quite as powerful as adaptive stalling, and thrashwait is unable to prevent invalidation thrashing.

Like adaptive stalling, thrashwait is unable to prevent HAI thrashing, and because APRIL supports high-availability interrupts, the thrashwait algorithm is not used in Alewife. Instead, a rather complex collection of deadlock handling mechanisms called *associative thrashlock* is employed. First, a set of associative transaction buffers is supported. These buffers not only hold state to track outstanding requests, they also hold the cache line data, providing extra associativity, much like a victim cache [Jou90]. Each context has its own set of two transaction buffers (one for an instruction and one for a data request) and two tried-once bits. The same thrashing detection scheme as for thrashwait is used, however, livelock is no longer possible, since each context has its own transaction buffers. The additional transaction buffers are provided for the use of the HAI handler and for prefetching. HAIs are handled by providing two buffer lock bits, one for instruction thrashing, and one for data thrashing. These buffer lock bits are set when thrashing is detected (and therefore the processor is stalled). When the bit is set, the cache controller prevents data in the transaction buffers from being invalidated during the HAI.

Transaction buffers add a fair amount of complexity to Alewife, not only because the functionality of the buffers is complex, but also because valid data is kept in the buffers. This adds extra capacitive loads to the cache datapath, and also requires that the buffers be able to override data being read from that cache. However, given the requirements of supporting HAIs on a sequentially consistent processor with a single combined instruction and data cache, the associative thrashlock solution adopted by APRIL seems a reasonable alternative.

Supporting HAIs under our constraints would also be difficult (although easier than for APRIL). We briefly outline how this would be done. Supporting HAIs requires the addition of a single extra buffer, the *HAI victim line*. This line contains two state bits: a *thrash* bit and a *valid* bit. The line also contains space for a cache tag, a pointer to a line in the cache, and line of data. When thrashing is detected, the HAI victim line thrash bit is set, valid bit cleared, and pointer loaded with the address of the stalled cache line. If the processor finishes stalling without a HAI occurring, the HAI victim line thrash bit is reset (clearing the HAI victim line). If a HAI

occurs and the stalled line is replaced or invalidated during the HAI, its tag and data are loaded into the HAI victim line and the victim line valid bit is set.

We will now outline how HAI deadlock is prevented. If a high-availability interrupt occurs while a context is adaptively stalling, the processor is either (a) stalled due to a read whose reply has not returned or (b) stalled due to a read or write which cannot allocate a line (due to the line being pending). The processor cannot be stalled due to a write whose reply has not returned, since writes complete upon allocating the cache line.

We will start by addressing the first case. During the HAI, two scenarios hold. First, the cache line for which the context is stalled will have avoided replacement during the HAI, in which case the processor will complete its request after the HAI completes. Otherwise, the line will have been replaced during the HAI, and will be occupying the HAI victim line. When the HAI completes, the context retry can get its data from the HAI victim line. The HAI victim line valid and thrash bits are reset upon the processor making its first request after the HAI, as the data may have been invalidated during the HAI by an actual coherence request, not mapping conflicts, and therefore the data use must be one-shot only. Thus, for a context stalled for a read reply, the processor will be successful upon its first repeat from the HAI.

We now address the second case, where the processor is stalled on allocating a cache line. We will assume that the processor reissues the full request to the cache for a write retry (i.e. the processor will regenerate the write data). By constraining the HAI to return to the same context that was stalled (a constraint used in APRIL), and forcing the HAI handler to do a fence[6] before returning, we can guarantee that the context will be able to allocate its cache line upon reissue. This is because no other contexts are allowed to issue requests during the HAI, and all lines requested by the HAI will have completed. Thus upon retry, the processor will reissue its request, missing in the HAI victim line. However, the cache line can be allocated at this point. For a write, since the line can be allocated, the write is considered complete. For a read, the cache line is allocated and the HAI victim line reinitialized. We are now at the starting state of the first case, and the read is therefore guaranteed to complete.

The complexity required to handle HAIs is nontrivial. For this reason, architects must carefully weigh all the options before making a decision on requiring HAI support.

[6] A fence is an operation which blocks until all previous outstanding requests have completed [GLL+90].

5.3 Performance Issues

In this section we explore the performance of our lockup-free cache proposal. One of the outstanding questions was how many writes should be supported by the pending-write buffer and Section 5.3.1 answers this question. This number is larger than one, showing that there are performance benefits to supporting multiple outstanding writes. However, the number of outstanding write misses is still small enough that the implementation cost of the pending-write buffer will be reasonable. In the previous section we claimed that adaptive stalling should have a minimal impact on performance, and Section 5.3.2 shows that the percentage of cycles the processor spends stalled as a result of using adaptive stalling to prevent deadlock is small. Finally, because contention for the cache will occur between the requests of the active contexts and the replies of the inactive contexts, we examine this in Section 5.3.3. Contention is not a major problem for our base 32 byte lines, but does become a problem if the line size is increased to 128 bytes. Wide-ending the data helps to address the contention problem.

The base cache configuration used for these studies assumes the following cache parameters:

1. Adaptive stalling is used to prevent multiple context deadlock.

2. Buffer deadlock is handled by providing large buffers between the caches and limiting the number of requests from the processor and external interfaces (modeled as infinite buffers between caches).

3. Writeback of dirty data is buffered.

4. Reply data requires the cache to be busy while the cache is filled (four cycles for the primary cache, based on 32 byte lines and a 16 byte, half-speed memory interface to the processor).

5. Invalidations occupy the cache for one or two cache cycles (one for the tag lookup, and potentially one for the invalidation).

5.3.1 Multiple Outstanding Writes

We start by examining the benefits of allowing multiple outstanding writes by presenting histograms showing the number of pending-write buffer entries required to service each write miss in Figure 5.11. Results from Cholesky were not included since it did not see any benefits from

multiple contexts. The most common situation was that no other write misses were outstanding, accounting for over 50% of all write misses for most applications. However, there was a significant number of writes which encountered one, two, or for some applications three or four outstanding write misses. Since these histograms only count write misses to different cache lines, they only show the benefits of pipelining multiple write requests. Allowing write hits to also access the cache while misses are outstanding is another benefit of the pending-write buffer.

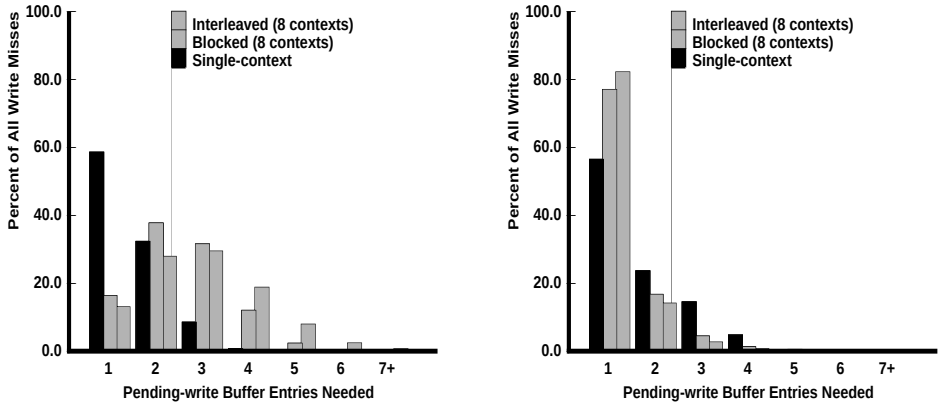
Table 5.6: Speedup due to supporting multiple outstanding writes for the single-context processor.

Number of writes	MP3D	Barnes	Water	Ocean	Locus	PTHOR	Mean
1	1.34	1.30	1.01	1.35	1.23	1.18	1.23
2	1.50	1.45	1.02	1.52	1.29	1.21	1.32
4	1.57	1.56	1.02	1.53	1.29	1.21	1.35
Infinite	1.57	1.57	1.03	1.54	1.31	1.22	1.36

The performance gains for various sizes of pending-write buffers are listed in Table 5.6 for the single-context processor, in Table 5.7 for the multiple-context processor. These tables show that the pending-write buffer does indeed provide significant performance advantages over a standard write buffer with read-bypassing. The figure also shows that the pending-write buffer can be built at a reasonable cost, as a four-entry buffer would be sufficient to realize nearly all the performance gains for both the single-context and multiple-context processor. The only significant performance gains from a pending-write buffer with more than four entries occurred for the eight-context MP3D runs.

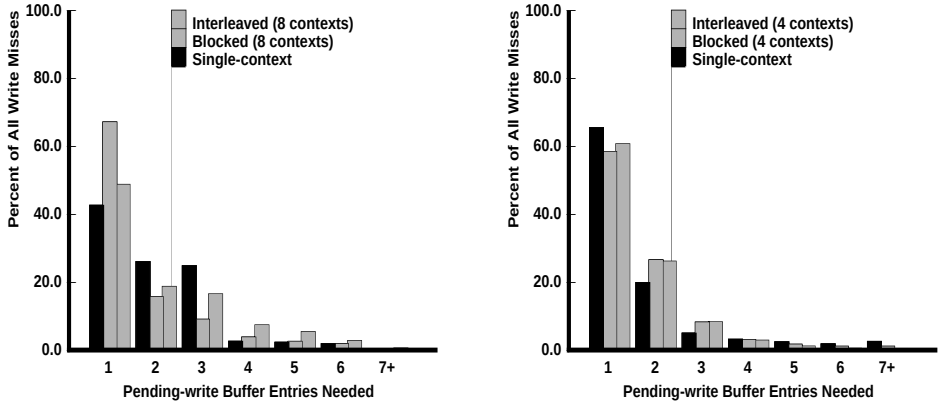
Table 5.7: Speedup due to supporting multiple outstanding writes for the multiple-context processor.

Num. writes	Scheme	MP3D	Barnes	Water	Ocean	Locus	PTHOR	Mean
0	Blocked	3.99 (8)	3.29 (8)	1.21 (4)	1.93 (2)	1.58 (4)	1.32 (2)	2.00
	Interleaved	3.91 (8)	4.36 (8)	2.07 (8)	1.66 (2)	1.70 (4)	1.25 (4)	2.23
1	Blocked	4.05 (8)	3.37 (8)	1.21 (4)	2.03 (2)	1.56 (4)	1.41 (2)	2.05
	Interleaved	4.21 (8)	4.63 (8)	2.09 (8)	1.88 (2)	1.68 (4)	1.37 (4)	2.37
2	Blocked	4.24 (8)	3.38 (4)	1.21 (8)	2.19 (4)	1.57 (4)	1.40 (2)	2.09
	Interleaved	4.47 (8)	4.67 (8)	2.08 (8)	2.35 (4)	1.74 (4)	1.41 (2)	2.51
4	Blocked	4.50 (8)	3.39 (4)	1.21 (8)	2.26 (4)	1.55 (4)	1.41 (2)	2.12
	Interleaved	4.82 (8)	4.70 (8)	2.09 (8)	2.37 (4)	1.78 (4)	1.41 (2)	2.56
Infinite	Blocked	4.57 (8)	3.37 (8)	1.21 (8)	2.28 (4)	1.62 (4)	1.31 (2)	2.12
	Interleaved	5.43 (8)	4.69 (8)	2.10 (8)	2.39 (4)	1.74 (4)	1.40 (4)	2.60



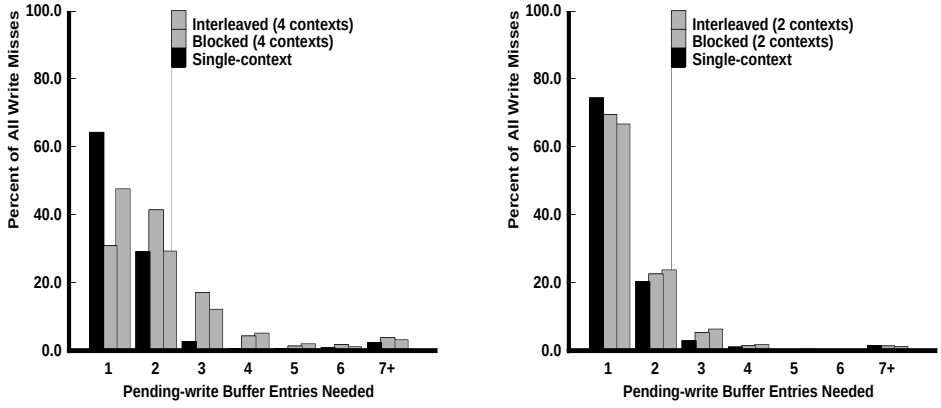
(a) MP3D

(b) Barnes



(c) Water

(d) Locus



(e) Ocean

(f) PTHOR

Figure 5.11: Histogram of number of outstanding write misses encountered by each write miss (includes the miss itself).

5.3.2 Adaptive Stalling Performance

Adaptive stalling attempts to detect deadlock situations on the basis of the processor repeating a memory request. Sometimes these repeats may simply be due to conflicts in the cache which are not causing a deadlock situation, and the processor would perform better if it allowed the retry of the memory request to cause a context switch for these situations. By increasing the threshold on the number of times a memory request must be repeated before adaptive stalling kicks in, the latency of these temporary conflicts can be tolerated.

Table 5.8 shows the percent of total cycles lost due to adaptive stalling for thresholds of one, two, four, and eight retries. With a single retry, the impact of conflicts for the same cache line are small for all applications except Locus, which spends 3% of its cycles stalled due to conflicts, and MP3D, which spends nearly 10% of its cycles stalled due to conflicts for the blocked scheme, 6% for the interleaved scheme. By making the threshold only slightly larger, such as two or four retries, most of the stalling due to conflicts is removed. The lower number of cycles stalled by making the retry value larger must be traded off with the lost performance due to switching a larger number of times before stalling in true deadlock situations. The best performance was achieved with four retries, however the performance difference between two, four, and eight retries was quite small, so selecting a value in the two to four range seems reasonable.

Table 5.8: Percent of total time stalled due to adaptive stalling.

Threshold	MP3D	Barnes	Water	Ocean	Locus	PTHOR
Blocked						
One retry	9.8% (8)	1.7% (8)	0.4% (8)	0.6% (4)	3.0% (4)	1.6% (2)
Two retries	2.6% (8)	0.9% (8)	0.2% (8)	0.3% (4)	1.8% (4)	1.1% (2)
Four retries	0.8% (8)	0.2% (8)	0.1% (8)	0.2% (4)	0.8% (4)	1.0% (2)
Eight retries	0.2% (8)	0.0% (8)	0.0% (8)	0.1% (4)	0.3% (4)	0.7% (2)
Interleaved						
One retry	5.8% (8)	1.6% (8)	0.7% (8)	0.6% (4)	3.0% (4)	0.8% (2)
Two retries	3.7% (8)	0.9% (8)	0.4% (8)	0.3% (4)	2.0% (4)	0.1% (2)
Four retries	1.5% (8)	0.3% (8)	0.2% (8)	0.1% (4)	0.9% (4)	0.0% (2)
Eight retries	0.6% (8)	0.1% (8)	0.0% (8)	0.0% (4)	0.1% (4)	0.0% (4)

5.3.3 Cache Occupancy

As the processor tolerates more and more memory latency, the cache becomes a hotly contended resource. It is therefore important to keep the occupancy of cache operations as low as possible.

Cache occupancy for hits is a single cycle, however, the occupancy for a cache miss may be much larger, as it includes the occupancy of both the original request which caused the miss and of the reply data returning. Because the memory interface of the processor tends to be narrower than the cache line size, and is often running at a lower rate than the internal logic of the processor [Dig92b, Hei93], reply data is generally returned to the processor over a number of cycles. With a standard processor which blocks on loads, the data can be written into the cache as it streams in with minimal performance penalty.[7] For the multiple context processor, this returning data may lock an executing context out of the cache for several cycles, and we would like to reduce the occupancy of the data response. This can be done by gathering the response in a buffer and then writing the entire cache line into the cache in a small number of cycles. This is referred to as *wide-ending* the data into the cache.

We examined the performance of the cache both with and without wide-ending of data. The external interface of the processor is assumed to be 16 bytes wide, running at one-half the internal processor frequency. Because the cost of the cache fill depends on the line size, we ran experiments for both our standard 32 byte lines and for larger 128 byte lines. The reply data occupancy is a single cycle with wide-ending and four cycles without wide-ending for the 32 byte lines, four cycles with wide-ending and sixteen cycles without wide-ending for the 128 byte lines.

Table 5.9 lists the fraction of the total processor cycles lost due to the processor stalling because the external interface had control of the cache. The external interface can busy the cache from external invalidation, flush, or copyback requests, or with reply data. With 32 byte lines, the percent of time spent in these cycles is fairly small for most applications, with the exception of MP3D, where over 8% of the processor's cycles are lost to these stalls for the blocked scheme, over 12% for the interleaved scheme. For 128 byte lines, the overhead due to cache contention is much more serious, and four of the SPLASH applications lose more than 5% of their cycles to cache contention. MP3D is again particularly bad, with cache contention consuming 19% of the cycles for the blocked scheme, 25% for the interleaved scheme. For both line sizes the interleaved scheme keeps the processor busier, and therefore loses a larger percent of its cycles to these cache contention stalls. Adding wide-ending reduces the stalls due to cache contention significantly.

Of course, some of the extra cycles gained by reducing the cache contention may be lost

[7] If the processor supports early restart, a small performance penalty arises due to the remaining reply data consuming cache bandwidth.

Table 5.9: Percent of total time stalled due to cache contention.

Line Size, Reply Policy	MP3D	Barnes	Water	Ocean	Locus	PTHOR
Blocked						
32B, No wide-ending	8.4% (8)	0.5% (8)	1.5% (4)	4.7% (4)	2.0% (4)	1.2% (2)
32B, Wide-ending	3.1% (8)	0.2% (8)	1.5% (8)	1.4% (4)	0.5% (4)	0.9% (2)
128B, No wide-ending	19.3% (8)	7.9% (8)	1.7% (4)	7.4% (2)	8.9% (4)	4.0% (2)
128B, Wide-ending	4.7% (8)	1.5% (8)	1.4% (4)	1.4% (2)	1.7% (4)	1.1% (2)
Interleaved						
32B, No wide-ending	12.3% (8)	2.1% (8)	1.9% (8)	3.2% (4)	2.4% (4)	1.2% (2)
32B, Wide-ending	4.7% (8)	0.9% (8)	0.8% (8)	1.6% (4)	0.7% (4)	0.8% (2)
128B, No wide-ending	24.9% (8)	12.4% (8)	1.4% (4)	10.5% (4)	11.7% (4)	3.2% (2)
128B, Wide-ending	7.0% (8)	3.1% (8)	2.1% (8)	2.9% (4)	2.2% (4)	1.1% (2)

in the memory system or other stalls. We show the performance improvements due to wide-ending in Table 5.10. For 32 byte lines only MP3D and Ocean, the applications with the largest cache contention, showed significant gains. The performance improvements for the other applications were small enough to be overwhelmed by small perturbations in application sharing and synchronization behavior between simulation runs. Cache contention was much more serious for 128 byte lines, and nearly all applications showed significant performance gains from wide-ending.

Table 5.10: Speedup due to wide-ending.

Line Size, Reply Policy	MP3D	Barnes	Water	Ocean	Locus	PTHOR	Mean
Blocked							
32B, No wide-ending	2.13 (8)	1.21 (4)	2.73 (8)	1.46 (4)	1.13 (2)	1.17 (4)	1.54
32B, Wide-ending	2.14 (8)	1.14 (4)	2.91 (8)	1.48 (4)	1.14 (2)	1.18 (8)	1.55
128B, No wide-ending	2.02 (8)	1.06 (4)	1.59 (8)	1.08 (2)	1.06 (2)	1.17 (4)	1.29
128B, Wide-ending	2.20 (8)	1.15 (4)	1.84 (8)	1.15 (2)	1.00 (2)	1.18 (4)	1.36
Interleaved							
32B, No wide-ending	2.89 (8)	1.28 (4)	3.21 (8)	1.53 (4)	1.17 (2)	2.01 (8)	1.87
32B, Wide-ending	2.92 (8)	1.31 (4)	3.46 (8)	1.56 (4)	1.16 (2)	2.05 (8)	1.91
128B, No wide-ending	2.64 (8)	1.11 (4)	1.61 (8)	1.22 (4)	1.00 (2)	1.86 (4)	1.48
128B, Wide-ending	3.12 (8)	1.21 (4)	1.89 (8)	1.32 (4)	1.07 (2)	1.95 (8)	1.64

We were able to reduce the number of cycles lost due to cache contention down to a small fraction of the processor cycles by adding wide-ending. We could take wide-ending one step further and provide enough reply buffers to handle the requests from all the contexts. By being

more intelligent about when the reply data was placed in the cache, the effects of cache contention could be nearly eliminated. For example, under the blocked scheme, these replies could be written into the cache only during the pipeline flush cycles of a context switch. Or, by pipelining the arbitration for the cache, the replies could be slipped in during cycles when the cache would otherwise be idle (due to the instruction in the data fetch stage not being a load or store). Our experiments have shown that wide-ending removes most of the cycles lost due to cache contention and these more expensive options would probably result in very little performance gain.

5.4 Summary

Previous lockup-free cache designs did not explore in detail the tradeoff between tracking outstanding requests in separate transaction buffers or within the cache itself. In addition, most lockup-free cache designs targeted a specific method of generating multiple outstanding requests. Finally, the designs were sometimes complex because they allowed multiple outstanding requests to the same cache line.

We have examined the requirements placed on lockup-free caches by the four major mechanisms for generating multiple outstanding requests: prefetch, relaxed memory consistency, nonblocking loads, and multiple contexts. We proposed a lockup-free cache that tracks outstanding prefetches, loads, and stores using a pending state in the cache. This pending state supports request merging and prevents multiple conflicting requests to the same cache line, which guarantees that a reply will always have a cache line allocated to place its data in. Write data is stored directly in the cache, allowing the store to complete immediately under a relaxed consistency model, by providing separate transaction buffers (the pending-write buffer) that maintain the subblock valid bits. Separate transaction buffers are needed to track the destination register for nonblocking loads and for reenabling a context when a memory operation completes for the interleaved multiple-context processor.

We also addressed deadlock issues for our lockup-free cache hierarchy. We showed that buffer deadlock can arise for any cache hierarchy containing multiple writeback caches, and this deadlock can be solved by a variety of techniques. We then focused on deadlock and livelock issues specific to multiple contexts. Under pathological conditions, it is possible for multiple-context processors to never complete their request due to multiple requests contending for the same line in the cache or to multiple processors simultaneously reading and writing the same cache line. To address these deadlock situations, we proposed a novel technique called adaptive

stalling. Adaptive stalling works by detecting thrashing conditions and backing off to a stalling mode of memory access under these thrashing conditions.

We evaluated our cache proposal, and found benefits from the pending-write buffer allowing multiple outstanding write misses. For the SPLASH applications, a pending-write buffer with four entries was enough to capture nearly all of the benefits, allowing an effective pending-write buffer to be built at a reasonable implementation cost. We also examined the performance impact of adaptive stalling and found this penalty to be small as long as a threshold larger than one was employed. Finally, we examined the effects of contention for the cache between processor requests and external requests and replies. We showed that by wide-ending reply data into the cache, this contention could be reduced to a small percentage of the total application cycles, even for caches with large line sizes.

Chapter 6

Blocked Scheme Implementation

Chapters 3 and 4 have shown the potential benefits of multiple-context processors, and Chapter 5 addressed the largest complexity for these processors — building a lockup-free cache. The issues that remain revolve around the hardware complexity required to support multiple contexts on a single processor. In this chapter we explore these issues for the blocked scheme, while the next chapter explores these issues for the interleaved scheme.

The hardware requirements for the blocked scheme can be broken into two categories. First, process-specific state needs to be saved across context switches. Second, scheduling logic and state is needed for switching between the contexts. Section 6.1 examines the state replication issues, while the scheduling logic and state are explored in Section 6.2. Section 6.3 summarizes the chapter.

6.1 State Replication

In order to drive the context switch time down to that of a pipeline flush, a reasonable amount of process-specific state needs to be replicated. Each context needs its own program counter, register file, and copy of any process-specific state that exists in the processor status word. We will examine each of these components in turn.

6.1.1 Program Counter

Before discussing how the program counter will be replicated, we first present the PC unit of the base single-context processor and discuss its operation to provide a basis for our discussion on how the program counter will be replicated.

Single-context Program Counter Unit

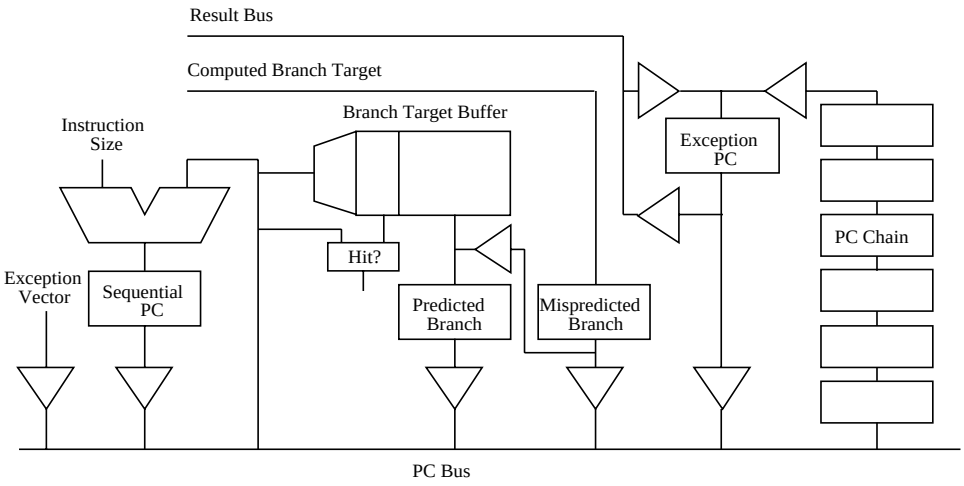


Figure 6.1: Single-context processor PC unit.

The PC unit for our single-context processor is given in Figure 6.1. The rectangles in the diagram represent registers; all registers have clock enable capability. The clock enable and tristate control are not shown. On any given cycle, one of several sources drives the PC bus. The possible PC sources are: (a) old PC value plus the instruction size (normal sequential flow), (b) Branch Target Buffer (predicted branch), (c) computed branch target (mis- or unpredicted branch), (d) exception vector, or (e) EPC register (restore from an exception).

The unit which determines whether a branch was predicted correctly and drives the computed branch target bus is shown in Figure 6.2. The branch target is calculated during the RF phase. For a PC relative branch, the branch target is the result of the sign-extended branch offset added to the address of the branch (which is taken from the PC chain). For a jump absolute, the target is taken directly from the instruction; for a jump register, the branch target is read from the appropriate register. During the EX phase, both the target and the fall-through address are compared to the predicted address and the miscomparison signal is selected from these two comparisons based on the result of the branch condition. If the branch was incorrectly predicted, during the next cycle the computed branch is driven onto the PC Bus, the BTB is updated, and the incorrectly fetched instructions in the pipeline are squashed.

The exception vector and EPC register provide the ability to take and recover from exceptions. During normal execution, as each instruction retires the address of that instruction is loaded into the EPC register from the PC chain. When an exception occurs, the loading of the EPC register

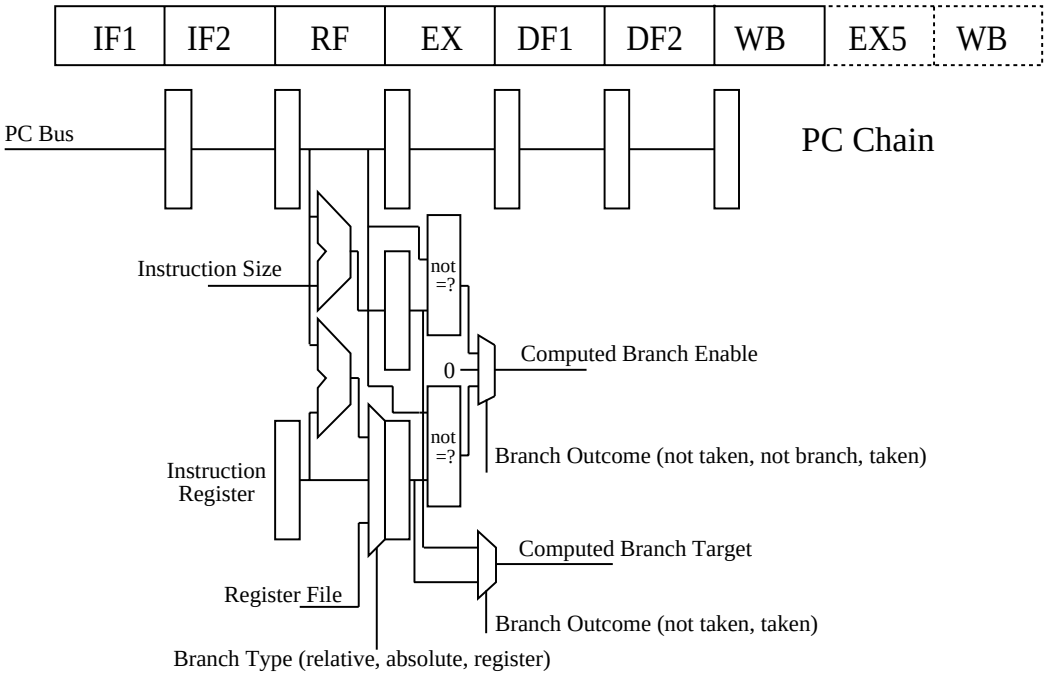


Figure 6.2: Branch target generation and prediction verification for the base single-context processor.

is stopped with the guilty instruction, and it and the following instructions in the pipeline are squashed (marked to not update any state). The exception vector is then placed on the PC bus, and the handler starts executing. The EPC is connected to the result bus to allow the exception handler to save and restore the EPC manually. When the exception has been handled, the EPC is forced onto the PC Bus via the use of an ERET (exception return) instruction, and execution continues from the point at which it left off. Note that because we have removed the MIPS branch delay slot, only a single EPC is needed. Supporting multiple contexts on a processor with branch delay slots is discussed in [LGH92].

Multiple-context Program Counter Unit

The ability of the EPC to save an instruction address for later repeat is exactly the same functionality needed to correctly save the program counter after a context switch. To be able to support multiple contexts, we simply need to replicate this functionality for each context. A PC unit capable of supporting two contexts is shown in Figure 6.3. This PC unit is very similar to that of a single-context processor, with the only difference being a modification to the EPC register

in order to support the two contexts. This modification adds an EPC register per context, which doubles as both the exception PC register and the context restart register (which contains the saved PC for that context). We now explain the multiple-context PC unit operation by discussing how it handles exceptions and context switches.

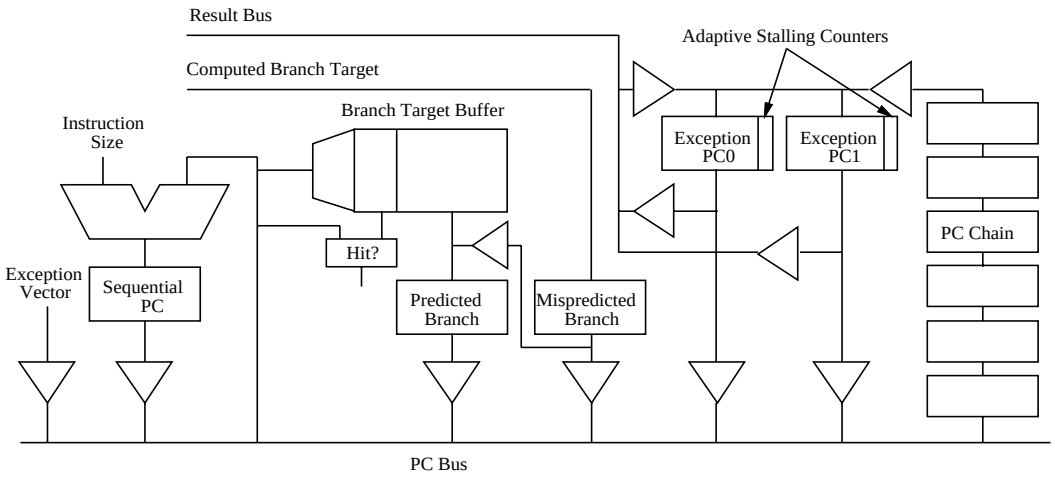


Figure 6.3: Two-context processor PC unit.

Exceptions continue to use the EPC register in the same manner as the single-context processor. The EPC register for the active context is continually being updated during normal operation, while the EPC for the idle context remains unchanged. When an exception occurs, the EPC stops being updated and the exception vector is driven onto the bus. The behavior of the PC unit for a context switch is very similar to that for an exception. When a context needs to be switched, the context switch is delayed until the normal exception point, at which time the EPC register for the blocked context stops loading as if an exception had occurred. The partially completed instructions in the pipeline are squashed and the next context is selected. The EPC register of the next context is driven onto the PC bus, and the new context starts executing at the instruction where it left off at its previous context switch. The value of the adaptive stalling counter is also sent down the pipeline with the PC, and the counter incremented. If the instruction completes (by finding data in the cache), the counter is cleared, otherwise the processor either switches or stalls depending on whether the terminal count has been reached.

Sharing the same EPC register for both context switches and exceptions causes a problem when the exception handler takes a data cache miss. Under the above scheme, the EPC value would be destroyed by the context switch. In addition, since the processor is in kernel mode,

without providing separate mode bits per context, switching to another context could allow user code to run in kernel mode, which is definitely a protection violation. For these reasons, context switching needs to be disabled upon taking an exception, much like interrupts are disabled. Before returning from the exception, the handler would be responsible for reenabling context switching.

This disabling of context-switching would seem to preclude the operating system from taking advantage of using multiple kernel threads, however, this is not the case. Provided the kernel can access the EPC registers of contexts other than the active context, it is possible for the kernel to manually save and restore the EPC registers. Therefore, if the kernel wishes to run in multithreaded mode, it simply needs to save the appropriate state (including the EPC) for as many contexts as it wishes to use, update the context control register (described in the next section) to indicate only the kernel contexts as being loaded on the processor, and then load up the multiple kernel threads.

To summarize, the additional hardware necessary to replicate the program counter turns out to be very modest — one 64-bit EPC register per additional context. We now turn to the register file to see if its replication requirements are also this manageable.

6.1.2 Register File

The simple approach to replicating the register file is to design a register file with *N* times as many registers, where *N* is the number of contexts per processor supported. This simple replication has the advantage that the registers of any context can be accessed during a given cycle. The disadvantages are an increase in the register file size (an increase of *N* for the registers themselves, and an increase of $log N$ for the decoder), and access time. We will call this scheme the *replicated* register file.

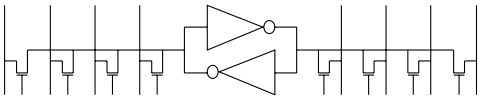
We have designed a replicated register file for a multiple-context processor supporting four contexts [McF93]. The design was for a register file with 4 ports (supporting 4 reads or 2 writes per each processor minor cycle) using a 0.8μ double-metal CMOS technology. The timing results are shown in Table 6.1. As we can see, there is a 50% increase in the four-context register file access time due to an increase in both the decode time and in the time required to drive a bit line which is four times as long as for the single-context case.

However, being able to access the registers of any context on the same cycle is not very useful for a blocked multiple-context processor, as only one context is executing on the processor at any given time. Therefore, the registers could be replicated in such a manner to reduce the space requirements by taking advantage of having only a single context active. We will call this

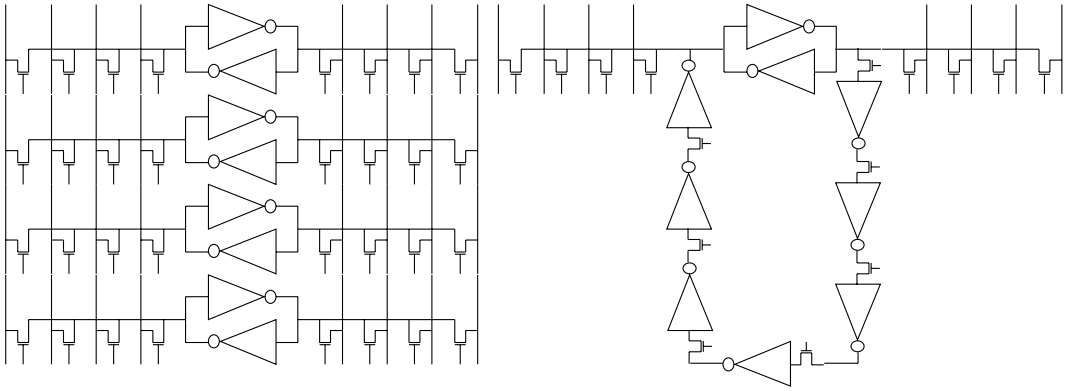
Table 6.1: Timing information for the replicated register file.

Delay Source	Single Context	Four Contexts
Decode and Word Line	1.53 ns	1.94 ns
Bit Line	0.57 ns	1.74 ns
Sense	1.07 ns	1.07 ns
Total Delay	3.17 ns	4.75 ns

scheme the *apportioned* register file. After considering several variants, our implementation of a single cell of the apportioned register file is shown in Figure 6.4, along with a cell from the both the replicated and single-context register file.



Single-context register file cell



Four-context replicated register file cell Four-context apportioned register file cell

Figure 6.4: Cells for both the replicated and apportioned register files.

The apportioned register file uses a dynamic shift register to store the state of all the contexts except the currently active context. When a context switch occurs, state from the current context is shifted into the tail of the shift-register, while the next context is loaded into the master cell from the head of the shift register. By using two-phase clocking to control the shifting, the shift register can be implemented using seven inverters. In addition, all of these inverters except the last which has to overdrive the master cell can be minimum-sized. By taking advantage of the single active context of the blocked scheme, we are able to reduce the size of each register cell roughly in half, and we remove the extra decoder delay. This results in the much more acceptable

18% increase in register file access time shown in Table 6.2.

Table 6.2: Statistics for the replicated and apportioned four-context register file.

Register File Size	Single Context	Replicated	Apportioned
Array Width	2610μ	2610μ	2610μ
Array Height	900μ	3600μ	2430μ
Delay Source			
Decode and Word Line	1.45 ns	1.94 ns	1.57 ns
Bit Line	0.57 ns	1.74 ns	1.10 ns
Sense	1.07 ns	1.07 ns	1.07 ns
Total Delay	3.17 ns	4.75 ns	3.74 ns

Of course, the apportioned register file does require some additional logic to insure that either a context switch or shift register refresh occurs with a minimum frequency to avoid losing information in the dynamic shift chain. The overhead for this extra refresh logic should be minimal. The major limitation of the apportioned register file is that tolerating instruction latency becomes more difficult, as the instruction whose latency is being tolerated would like to write its result to the register file when it completes. For instruction latency to be tolerated with an apportioned register file, the result of such instruction must be placed in a holding register until the context starts executing again. Implementing these holding registers can be expensive, as one is required per context per functional unit that can produce a long-latency result, and they need to be able to bypass their result on context restart.

6.1.3 Processor Status Word

The final state that needs to be replicated is the process-specific portion of the processor status word. The processor status word (PSW) is a catchall phrase for the architecture- and implementation-specific process and kernel state residing on the processor. In Appendix E, we explore the processor status word of the MIPS R4000. We show that while modern processors have a large number of PSW registers, only a few of these registers contain process-specific state. In particular, adding multiple contexts to the R4000 PSW requires that the floating-point control/status register and the LL bit and LL address registers (used for implementing the load-linked/store-conditional synchronization primitive) be replicated to provide each context with its own copy. In addition, a single new Context Status register needs to be added. The Context Status register (a) keeps track of the context identifier of the currently executing context, (b) maintains information concerning which contexts on the processor contain valid processes, and

(c) holds the context-switch enable bit. Finally, a set of TLB control registers: EntryHi, EntryLo0, and EntryLo1, need to be modified to support the multiple address-space identifiers of the different contexts, and to allow a single TLB entry to be used for pages shared between contexts if so desired. Appendix E describes how these registers can be replicated for the blocked multiple-context processor. The extra complexity to replicate these registers is modest. The number and type of process-specific MIPS R4000 PSW registers is also compared to that of the DEC Alpha [Dig92b] in Appendix E and found to be similar.

6.1.4 Summary

Modification of the PC unit to support multiple contexts required very modest changes. While modern processors have a large number of processor status registers, for the architectures we examined, the MIPS R4000 and the DEC Alpha, the number of process-specific registers requiring modification for multiple contexts was small, and the modifications required were straightforward to implement.

This leaves the register file as the major replication cost. The percentage of chip area devoted to the register file is small for recent processors, so increase in chip area is not a large concern. More important is the increase in register file access time caused by increasing the register file size. We examined the straightforward approach to providing register replication for a four-context processor. This approach resulted in a four-fold increase in register file size and a 50% access time penalty. By designing a register file which stored the state of the nonexecuting contexts in a dynamic shift register, we were able to reduce the register file to a two-fold size increase and a 18% access time penalty. For many current processors, a 50% increase in register file access time will not impact the critical path, however, for future processors where the timing of the register file may be more critical, making the register file apportioned will help to keep the register from impacting the processor cycle time.

6.2 Context Schedule and Control

The blocked multiple-context processor also needs the ability to schedule and control its contexts. This breaks down to three requirements, that the processor be able to: (a) decide when to context switch, (b) determine the next context to run, and (c) perform the context switch. We will start our discussion with the requirements for detection of the context switch.

6.2.1 Context Switch Detection

Under the blocked scheme the decision to context switch is fairly straightforward. Three events can cause a context switch: a cache miss, an explicit context switch instruction, and a context timeout.

The decision to switch for a cache miss is generated whenever there is a miss and three conditions are satisfied: context switching is enabled (CE), the repeat value of the instruction is not greater than the adaptive stalling threshold, and another context is ready to run.[1] The enabling conditions are shown in Equation 6.1, where OneCount is a function that returns the number of ones in a bitvector.

$$IdleCS = CE \wedge (RepeatCount < Threshold) \wedge (OneCount(CIDV ector) > 1) \quad (6.1)$$

The explicit context switch instruction can be implemented in a straightforward fashion by having it mimic a cache miss by the subsequent instruction. However, by switching immediately upon decoding the explicit switch instruction, better performance can be gained and shorter latencies can be hidden. Implementation of this lower latency switch operation results in a large increase in pipeline control complexity as will be discussed in Section 6.2.3. Chapter 3 showed the performance gains for the quick explicit switch to be small and therefore we assume that an explicit switch control signal with the proper timing (appears as if a cache miss occurred by the following instruction) is run from the cache to the context switch controller.

Finally, adding a watchdog timer mechanism to the blocked multiple-context processor prevents one application which is either running for a large number of cycles without a miss or is deadlocked due to improper synchronization behavior from hogging all the processor cycles. This watchdog timer can be easily implemented by a counter which advances every cycle but is reset by a context switch. If the counter overflows, it sends a timeout signal to the switch logic to force a context switch. It is important to store the address of the first noncompleted instruction in the EPC, as the instruction being written back when the timeout occurs has completed (these are the same requirements as for an external interrupt).

Both the explicit switch instruction and watchdog timer should only cause a context switch if context switching is enabled and another active context is loaded. Thus, the complete context switch signal depends on the cache miss (Miss) and cache miss switch enable, timeout (TO),

[1] We found no performance advantage in using context availability when selecting the next context, and for a blocked processor which always switches round-robin between loaded contexts this final condition can be guaranteed by having the operating system clear the CE bit for all single-context execution.

explicit context switch instruction (ES), context switch enable (CE), and CIDValid vector has shown in Equation 6.2. The cache stall signal (CStall), generated when there is a cache miss and the switch enabling conditions are not met, is given in Equation 6.3.

$$Switch = (Miss \wedge EnableCS) \vee (CE \wedge (OneCount(CIDValid) > 1) \wedge (TO \vee ES)) \quad (6.2)$$

$$CStall = Miss \wedge \overline{EnableCS} \quad (6.3)$$

6.2.2 Next Context Selection

We have performed experiments that found a strict round-robin interleaving of contexts to perform as well as a scheme that includes context availability information, therefore the next context can be selected round-robin, modified by the number of loaded contexts (stored in the CIDValid vector). Thus, three signals are needed to determine the next context. These signals are: (a) the current context, (b) the CIDValid vector, and (c) the context switch signal from the previous section. This next context selection can be implemented using a priority encoder where the lowest priority is always given to the context which is surrendering the processor.

6.2.3 Context Switch Mechanics

Now that we have replicated the necessary state and provided a mechanism for determining when to context switch and whom to switch to, we need to examine the actions required to perform the context switch itself. The operations occurring on a context switch are:

- Save the first uncompleted instruction address from the current context in the EPC.
- Squash all incomplete instructions in the pipeline.
- Start executing from the EPC of next thread.
- Load the new ASID identifier in EntryHi.
- Load the new information into the Floating-point Control/Status register.
- Load the new CID field in the Context Status register.
- Switch the register file control to the new context.

These operations all need to be performed at different points in the pipeline, as shown in Figure 6.5. The bars give the window for completion of each operation. Saving the current context's PC and starting the new context occur as part of the normal operation of the multiple-context PC unit described in Section 6.1.1. The saving and restoring takes place immediately upon detecting the context switch. Loading the new ASID identifier can occur anytime between the context switch and the time the next context will first access the TLB. For our processor, the TLB hit detection occurs in IF2 for the ITLB and DF2 for the DTLB, and the new ASID will need to be ready for these comparisons.

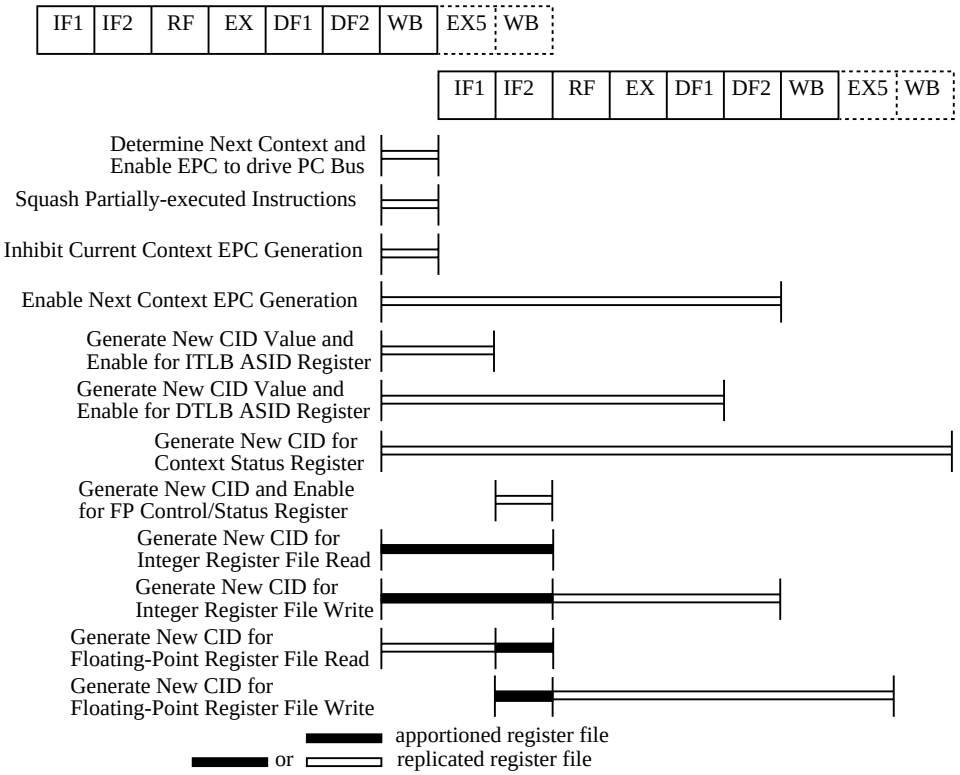


Figure 6.5: Timeline for a context switch.

The Context Status Register can only be accessed by the processor in kernel mode, so the earliest it would need to be updated is for a kernel read of the register. The worse-case scenario would be for the first instruction of the next context to cause an exception, and the first instruction of the handler to access one of these registers. This results in a large number of cycles in which to update this register.

The floating-point control/status register cannot be changed until after the floating-point write-back stage for the previous context (due to exception status) and before the rounding mode and flush denormalized to zero status is needed for the next context (they are needed by EX4). As an additional constraint, the next context could immediately read the FPCSR, requiring the register to be switched to the new context by the RF stage.

The switching of the CID values for the register file addressing is more involved. For our processor, we have two separate pipelines, one for the integer operations, and a longer pipeline for the floating-point operations. In addition, there are several operations which complete outside of the pipeline (floating-point divide, integer multiply and divide). This complicates the register file switch, as the longer operations will need to write their results to the register file when they complete. If the register file has been switched over too early, they will write their results to the wrong context's register.

However, the next context does not need to access the register file until the RF cycle. This gives a window of a few instructions before the register file has to be switched. This window is large enough for the standard floating-point operations to complete, as can be seen in Figure 6.5. For longer operations (such as floating-point divide), the action to be taken depends on whether the processor is using a replicated or apportioned register file. Recall that a replicated register file allows the registers of all contexts to be accessed while the apportioned register file only has the ability to access the registers of the current context. With a replicated register file, these longer operations can continue in the background, and by tagging their results with both the context and result register simply update the appropriate register. For an apportioned register file, the longer operations must complete before the context switch can take place (assuming holding registers are not provided for the results). This can be done by having the processor scoreboard stall the pipeline at the RF stage until the longer latency operations complete. Upon completion, the register files can be switched to the new context and the pipeline unfrozen.

This dependency on the register file between the context being switched out and the next context also affects the design of a fast explicit context switch instruction. The context switch instruction can be decoded early in the RF stage of the pipeline, and it should be possible to make the explicit context switch instruction start the switch by the end of the RF stage. Of course, for this fast context switch instruction to be able to allow pipeline latency to be hidden, the replicated register file must be used. Otherwise the apportioned file would stall at the point of the context switch for the outstanding operations from the previous context to finish, defeating the purpose of speeding up the switch.

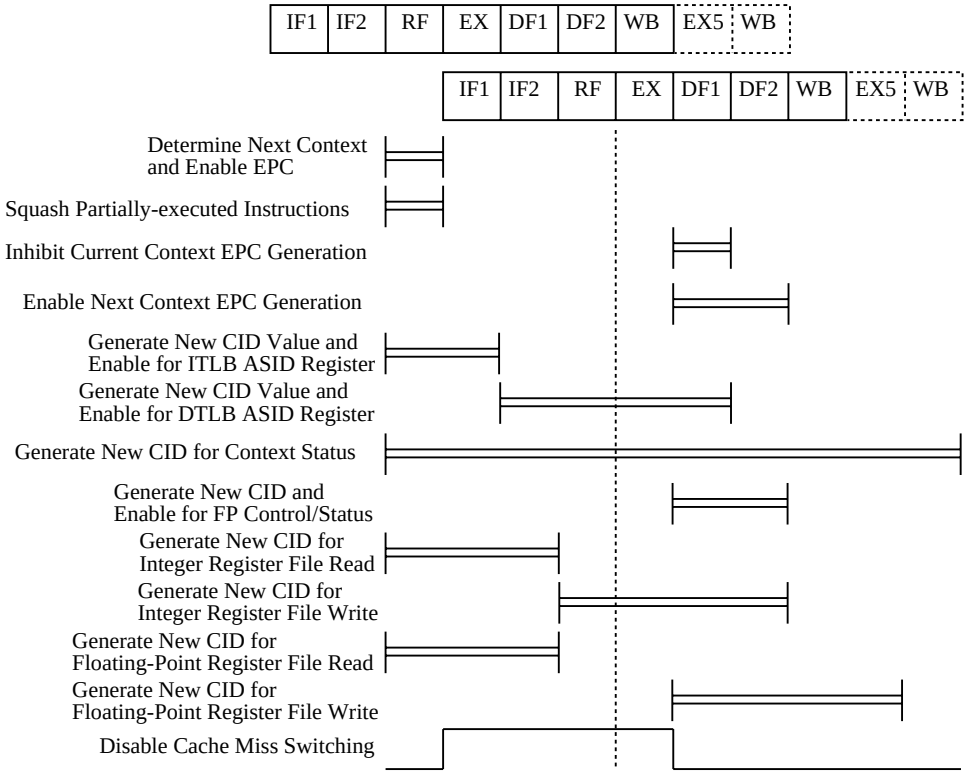


Figure 6.6: Timeline for a fast explicit context switch with a replicated register file.

Even assuming the replicated file, this fast context switch instruction causes some complications for instructions in progress from the switching context. Figure 6.6 shows the timing required to implement the fast explicit switch instruction. The next context starts executing after RF, causing the timing to get tighter for several of the operations. In addition to tightening the timing of the register file switch, the pipeline forwarding logic must now include the context identifiers of instructions when making its forwarding decisions to prevent forwarding results between different contexts.

The processor must also be more careful when enabling the next context's EPC. With the slower switch, we could enable the next context's EPC immediately, as all the instructions of the previous context were guaranteed to no longer be able to cause exceptions. However, this is no longer guaranteed, so the EPC of the next context must be preserved until no instructions from the previous context may except. The last instruction which could except is the one before the switch instruction itself. The broken line in Figure 6.6 shows the point at which the exception occurs for this instruction. Note that by this point several control registers have already been

switched to the next context. To properly handle the exception, these control registers must be switched back to the original context. Therefore, at an exception both the integer and floating-point register file read CIDs need to be loaded from their write CIDs. The ITLB ASID value needs to be loaded from the DTLB ASID value.

Not only can instructions in progress from the previous context except, but they may also cause a cache miss. Since the context switch has already been started, this cache miss must not cause a second context switch. Therefore, for the cycles shown on the graph, context switching due to cache misses must be disabled. The cache miss will still either result in a stall (due to adaptive stalling) or cause the instructions following the miss to be squashed and the EPC loading to be disabled, with the net result that the explicit switch simply started the switch for the cache miss a little early.

Finally, switching the floating-point control/status register (FPCSR) has become more complicated for the fast explicit context switch. The register cannot be switched until the floating-point operations from the previous context have updated the exception flags. The latest this could occur is the floating-point WB stage of the instruction immediately preceding the explicit switch. Since the FPCSR rounding mode and flush denormalized to zero fields are not needed until EX4, this leaves a few cycles in which to change the FPCSR, as shown in Figure 6.6. However, this late changing of the FPCSR does not allow the next context to immediately read the FPCSR register. Therefore, the interlock for the FPCSR read needs to be modified to also stall the pipeline if the FPCSR is read before the switching of its contexts can occur. This is in addition to the interlock ensuring that all outstanding floating-point operations are completed.

Providing a fast explicit switch requires a large amount of additional complexity and Chapter 3 showed the performance benefits of providing this fast switch to be small. This extra complexity and small performance improvement is a strong argument for implementing the slower explicit switch instruction.

6.3 Summary

In this chapter we have explored the extra hardware and design complexity required to implement a blocked multiple-context processor, using the MIPS R4000 architecture as a base for our exploration. We showed that it is possible to extend the standard single-context PC unit to multiple contexts by adding an extra exception PC register per context. We also found the number of process-specific registers in the processor status word to be small and extensible to multiple

contexts in a straightforward fashion. The largest increase in chip area due to multiple contexts occurs in the replication of the register file. Replication of the register file also slows its access time. Since register files tend to be a small portion of the total area of modern microprocessors, this second factor is of larger importance. We presented an optimization to fully replicating the register file which takes advantage of the blocked scheme having only a single context active at any given time. Our optimization, the *apportioned* register file, provides a fully-ported master cell only for the active context. The state of the inactive contexts is stored in a compact dynamic shift register whose head and tail are connected to the master cell. At a context switch, the master cell state is saved by shifting its contents into the tail of the shift register, while the next context loads the master cell from the head of the shift register.

Finally, we examined the implementation complexity required to schedule and control the multiple contexts. We showed that the decisions on when to context switch and which context to run next can be made by taking into account only a small number of input conditions. We also showed that the timing for switching the processor state between the two contexts should be easily met, unless a fast explicit context switch instruction is desired. Supporting a context switch instruction which causes a switch immediately after decode greatly increased the processor complexity and did not show enough performance benefits over the slower switch to justify this increase in cost.

Chapter 7

Interleaved Scheme Implementation

In the previous chapter we examined the implementation costs for the blocked scheme, and found the costs to be relatively modest. This chapter explores the implementation costs for the interleaved scheme, in order that the complexity of the two schemes may be compared.

Because the interleaved scheme is switching contexts on a cycle-by-cycle basis, the instruction issue unit of the processor needs to be capable of issuing from multiple active streams. In addition, tolerating long-latency operations requires the ability to make contexts active and inactive. This active status feeds into the instruction issue unit and prevents instructions from being issued from an inactive context. The instruction issue unit is discussed in Section 7.1, while the details of the context availability control is discussed in Section 7.2.

The multiple instruction streams also impact the processor pipeline. The PC chain of the processor needs to be augmented to associate a context identifier with each instruction which then flows down the pipeline along with the other control signals. This chain of CIDs is used in the pipeline forwarding decisions and to select the correct context state needed by the various functional units. Control of the interleaved pipeline in covered in Section 7.3.

Finally, the multiple-context processor requires that each context have its own copy of the program counter, register file, and process-specific processor status registers. This state needs to be available constantly, as the next instruction could be from any given context. State replication issues are discussed in Section 7.4. We end by comparing the blocked and interleaved implementation requirements and summarizing the chapter in Section 7.5.

7.1 Instruction Issue

A large portion of the complexity in issuing instructions from multiple contexts resides in the program counter unit. The PC unit needs to be modified to provide a separate next PC register per context to allow issuing from multiple streams. Saving the program counter when a context becomes unavailable can be accomplished by using a separate EPC register per context, as was done for the blocked processor. Taking an exception in the interleaved multiple-context processor is complicated because the EPCs of *all* contexts need to be loaded with the last uncompleted instruction. Because context availability can change at any point, the last uncompleted instruction for a given context can be in any stage in the pipeline or may not have even been issued to the pipeline.

We start this section by discussing the normal operation of the PC in Section 7.1.1. We then explore exception and interrupt handling in Section 7.1.2 and finally discuss context availability changes in Section 7.1.3.

7.1.1 PC Unit

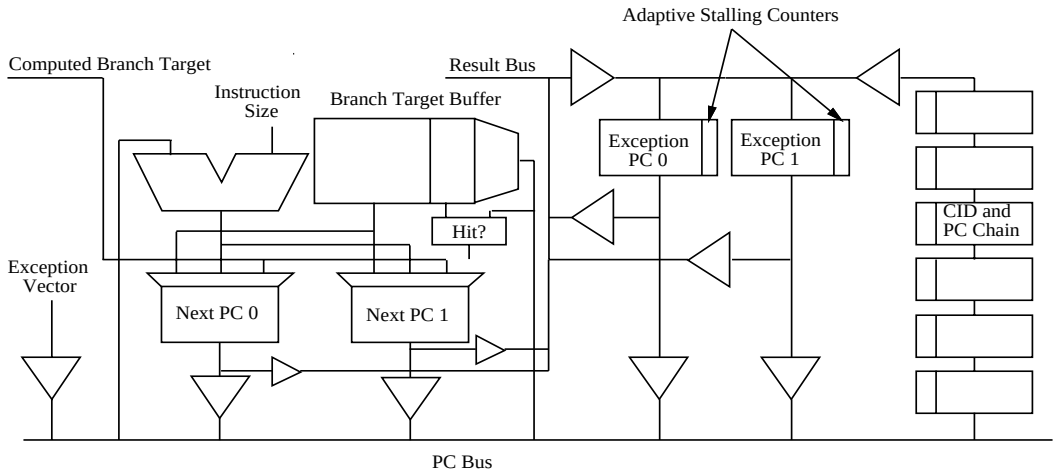


Figure 7.1: Two-context processor PC unit.

The PC unit must be able to determine the next instruction to be issued from each context. Determining this next instruction has become somewhat complicated under the interleaved scheme because the new PC value for a context becomes available a specific number of pipeline stages after issue and must be held until the context becomes active again. The delay between when

an instruction issues and when the new PC value becomes available depends on the type of instruction that was issued. The new PC value is not known until the end of the EX cycle for a branch which was not predicted correctly, while for nonbranch and correctly predicted branches the new PC value is available at the end of IF1. Because of the context interleaving, the context may not be able to drive its new PC value onto the PC bus immediately when it becomes available, and holding registers must be provided until the context is selected to drive the PC Bus. The three PC sources (sequential, predicted branch, computed branch) can be multiplexed into a single holding register per context, as shown in Figure 7.1. A bit is associated with each holding register to signify whether the register holds a computed branch which was loaded as a result of a previous branch being mispredicted. This bit is used to signal that the BTB needs to be updated to reflect the new branch prediction when the holding register is driving the PC Bus. This next PC register (NPC) is loaded by one of the following sources (in order of decreasing priority):

- The computed branch target if the instruction in EX is a mispredicted branch from this context.
- The predicted branch target if the current PC is from this context and hit in the BTB.
- The sequential address if the current PC is from this context.
- The NPC, causing the register to maintain its current value.

The computed branch has priority over all other sources if a previous branch was mispredicted, because this is guaranteed to be the correct path of instruction flow. Note that the determination of the mispredicted branch can actually occur before the predicted branch address has been issued to the PC Bus due to the context interleaving. If this occurs, the branch will only cost a single cycle, even though it was mispredicted. The unit which verifies branch prediction and computes the branch target is shown in Figure 7.2. In the single-context unit, the predicted target could be taken directly from the PC chain. However, for the interleaved processor the cycle-by-cycle switching implies that the predicted branch address may reside in the NPC or may already be somewhere in the PC chain when the prediction verification needs to be made. Hardware to locate this predicted address in the PC chain would be fairly complicated. This hardware can be avoided because the actual branch prediction is made immediately, and the predicted address can be loaded into a *predicted PC chain*, which keeps a copy of the predicted address in step with the branch instruction for comparison.

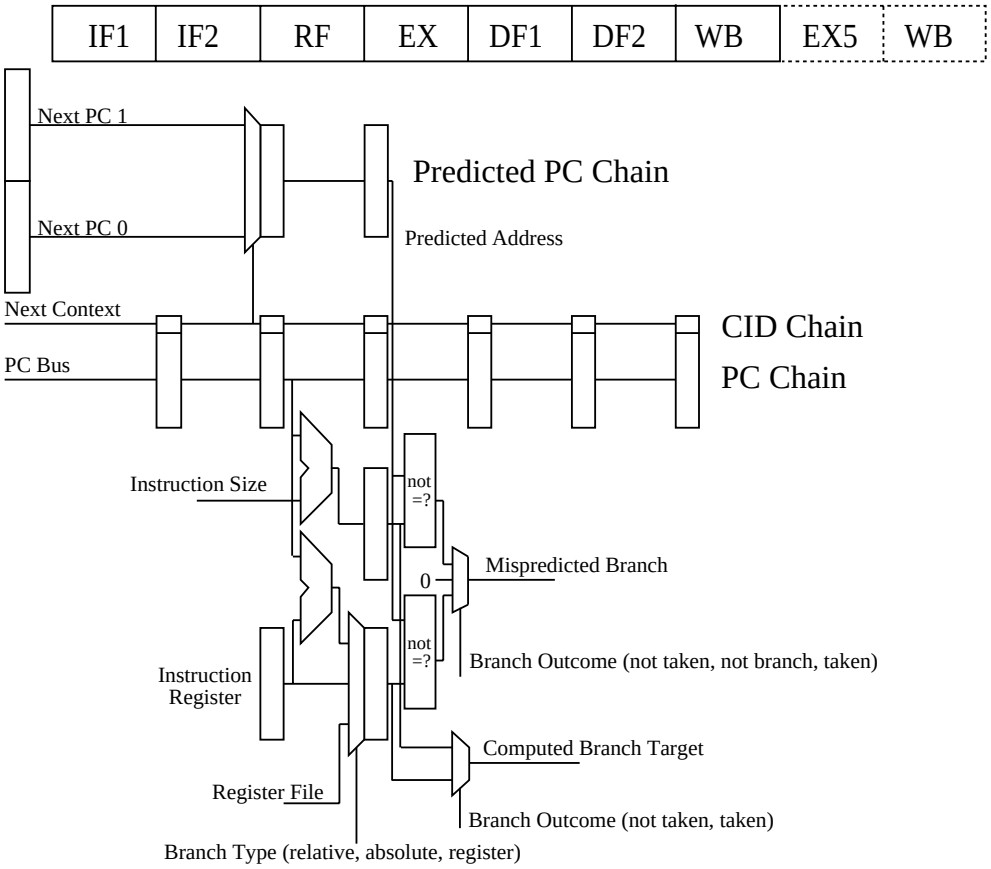


Figure 7.2: Branch target generation and prediction verification for the interleaved multiple-context processor.

In addition to affecting the PC unit design, the processor pipeline control also needs to handle mispredicted branches. Upon determining that the branch is mispredicted, the incorrectly fetched instructions in the pipeline must be squashed. This can be done by sending a branch squash signal coupled with a branch squash CID. Instructions following the branch in the pipeline with the same CID as the squash CID are then marked to not update any state. In addition to squashing the incorrectly fetched instructions, the computed branch is loaded into the NPC and the mispredicted status bit is set. On the next instruction issue from this context, the NPC will be driven onto the PC Bus and the BTB updated.

7.1.2 Exception and Interrupt Handling

In addition to issuing instructions from the multiple contexts, the PC unit must also be able to properly handle exceptions and interrupts. Exceptions on an interleaved processor could be correctly handled by requiring only the context causing the exception to squash its instructions in the pipeline and enter the exception handler. The exception handler would then run concurrently with the remaining contexts. However, there are several reasons not to adopt this model. First, since the exception handler may want to modify status registers, the processor now has the problem of needing to update status registers simultaneous with the status registers providing state for executing instructions. Second, since another context could except while the first is still being handled, the exception handlers would need to be able to handle multiple exception handler threads running concurrently. Finally, since exceptions and interrupts are infrequent events, there will be little performance gain from not flushing the entire pipeline. Therefore, on an exception or interrupt, the processor will flush the entire pipeline.

In addition to flushing the pipeline, the address of the first unexecuted instruction for *all* contexts must be saved. This is in contrast with the blocked scheme, where the excepting context only had to worry about saving its own PC address. Saving the excepting context's PC value is done by stopping the loading of its EPC with the guilty instruction. However, for the other active contexts, their EPC will contain the last *completed* instruction. Since the EPC needs to contain the last uncompleted instruction, these EPCs need to be updated as follows. At the point of the exception, all instructions in the pipeline are squashed, and the exception vector is forced onto the PC Bus. The squashed instructions are then advanced down the pipeline ahead of the exception vector. As these instructions reach the writeback stage, the EPCs of the other active contexts need to be loaded with the first squashed instruction from their context. To implement this properly, the EPCs need to have both a valid bit and a *gather* bit associated with them. The valid bit controls whether the EPC is updated with PC values as they fall off the pipeline. Normally, the valid bits on the EPCs are all invalid, and the EPC gathers PC values. On an exception, all instructions in the pipeline are squashed, the excepting context's EPC valid bit is set and gather bit cleared, and the gather bits of all other contexts are set. The gather bit signifies that the next instruction from this context to fall off the pipeline should set the EPC to valid (at which point the gather bit is cleared).

Note that it is possible for an active context to have no instructions in the pipeline when an exception occurs. This could occur if the number of active contexts is greater than the pipeline depth or if a context had just become active before the exception. For these contexts,

the EPC valid bit will remain invalid, and the NPC will hold the address of the first uncompleted instruction. Thus, to return from an exception, the contexts simply issue their EPC to the PC Bus if the valid bit is set, otherwise they issue from the NPC. When the EPC (or NPC) restarts the context, the valid bit and gather bit of that context are reset to handle the next exception.

7.1.3 Context Availability Change

In addition to handling exceptions, the interleaved PC unit must be able to handle context availability changes due to cache misses and backoff instructions (as mentioned earlier, the backoff instruction causes a context to become unavailable for a specific number of cycles). We will start by discussing cache misses. When the cache miss is detected, the issuing of instructions from that context needs to be stopped and all partially completed instructions in the pipeline squashed. Further issuing of instructions is stopped by clearing the context availability signal to mark the context as unavailable. Squashing of already issued instructions can be accomplished by providing a conditional squash signal coupled with a squash CID. All instructions in the pipeline that match the CID will be marked to not affect any state. The PC value of the instruction causing the miss is loaded into the EPC and the EPC valid bit is set.

When the context eventually becomes available again, the EPC of that context will be driven on the PC Bus when the context issues its first instruction. The context will then start reexecuting from the instruction which caused the context to become unavailable.

The backoff instruction has a similar behavior, however, the PC value of the backoff instruction cannot be stored in the EPC, as the context will then repeat the backoff instruction forever. Backoff instructions can be handled by including a *next* bit in the EPC. When a backoff instruction causes the EPC to be loaded with its address, both the valid and next bits are set. The next bit signifies that the real EPC is the subsequent instruction. When the context becomes available and next bit of the EPC is set, the EPC plus instruction size is driven onto the PC bus instead of just the EPC.

7.2 Context Availability Control

A context is available to issue instructions as long as it is not waiting on a memory operation or on a backoff instruction. In the previous section we discussed how the PC unit handles availability changes; in this section we cover the hardware mechanisms needed to maintain the availability status. We will start with the cache miss hardware, and then look at support for the backoff

instruction.

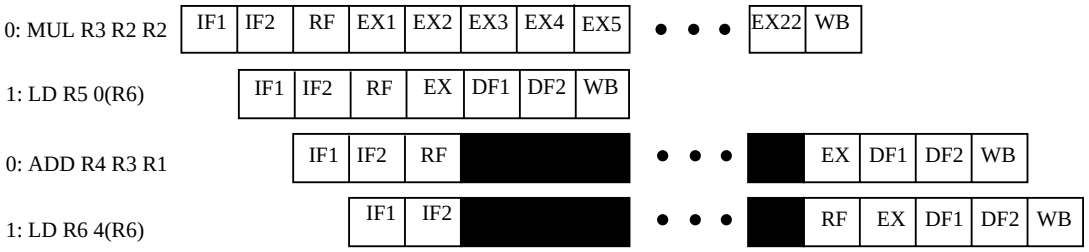
When a context encounters a cache miss, its availability signal needs to be reset to inhibit the context from being selected by the issuing mechanism. When the reply for this context returns, the context availability signal will be set to indicate that the context is again available. As discussed in Chapter 5, availability status can be tracked by providing a register per context to hold the miss address. We will refer to this buffer as the *Pending Miss* register. When a cache miss causes a context to become unavailable, this register is marked valid and loaded with the address causing the miss. The addresses of replies from memory are compared against this register, and when the proper reply returns, its pending miss register is marked invalid and the context is made available again.

Changing context availability as a result of a backoff instruction can be implemented using a backoff register per context. This register is loaded with the count of the number of unavailable cycles specified by the backoff instruction and then decrements each cycle until it reaches zero. While the backoff register is nonzero, the context is unavailable for issuing instructions.

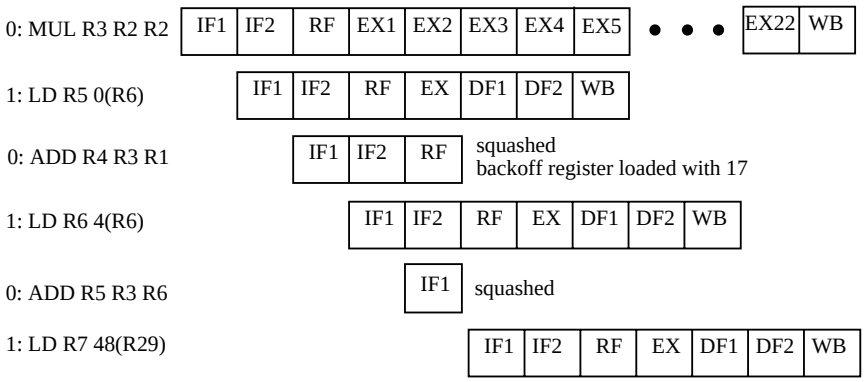
As for the explicit context switch instruction of the blocked scheme, the backoff instruction can take effect either early or late in the pipeline. When being used to tolerate synchronization latency, the decision to make the backoff instruction take effect sooner or later in the pipeline is not as important as for the blocked scheme, since the context interleaving is already reducing the cost of making a context unavailable. Therefore the backoff instruction can be made to take effect later in the pipeline if this results in a simpler implementation cost.

The backoff instruction can also be used to tolerate instruction latency, however doing so is more difficult than via an explicit context switch instruction for the blocked scheme. This is because the interleaving of the contexts makes determination of the proper backoff value dependent on the dynamic interleaving. Luckily, the interleaved scheme can tolerate most short latencies through its cycle-by-cycle switching, so only longer latencies need to be tolerated via backoff. By placing the backoff instruction immediately following the instruction whose latency is to be tolerated, the maximum uncertainty in selecting the proper switch value can be made as small as the number of contexts on the processor. However, the compiler can often place several unrelated instructions following the long-latency instruction, and ideally we would like to be able to force the backoff to occur after these unrelated instructions have been issued. As the number of instructions increases between the long-latency instruction and the backoff instruction, the probability also increases that the compiler will select a backoff value that is either too long or too short.

In addition to this difficulty, the value used by a backoff instruction to tolerate instruction latency is tied to a specific implementation, in terms of both number of hardware contexts and instruction latency. Thus, the value used for one implementation may be totally inappropriate for the next generation of processors. This lack of forward compatibility is in contrast to employing a context switch instruction for instruction latency tolerance under the blocked scheme. Since the context switch instruction helps to tolerate any latency greater than the switch cost, forward compatibility only becomes an issue if the cost of the operation latency drops below the cost of the context switch.



(a) pipeline interlock



(b) multiply latency tolerated by scoreboard-triggered backoff

Figure 7.3: Instruction latency toleration using dynamic context backoff detection.

This argues that the explicit backoff instruction should be used only for handling synchronization latency. Pipeline latency can either be handled solely by the context interleaving, or the effects of a backoff instruction can be implemented by hardware means. For example, a scoreboard could cause a context to become unavailable upon detecting a potential stall, making the context available again at precisely the correct time to avoid the stall. An example using this hardware-controlled backoff is shown in Figure 7.3.

7.3 Context Control

Controlling the contexts requires the ability to select the next context to issue an instruction to the pipeline and requires a pipeline capable of supporting the multiple instruction streams. We describe both of these requirements in this section.

7.3.1 Next Context Selection

For the interleaved scheme, determination of the next context is done round-robin, modified by the context availability. To support the processor operating with fewer than its maximum number of contexts, the CIDValid field from the status register maintains information on which contexts are loaded with a valid process. The processor gets context availability information from three sources: the cache miss status, the backoff register, and the CIDValid bits, as shown in Equation 7.1.

$$Context\ Available \equiv \overline{Cache\ Miss\ Pending} \wedge (Backoff\ Count = 0) \wedge CIDValid \quad (7.$$

The interleaved scheme switches contexts each cycle unless the pipeline stalls or the context enable bit in the Context Status register is cleared, and therefore determining the next context for the interleaved scheme depends on: (a) the current context, (b) context availability, and (c) a switch inhibit signal. As for the blocked scheme, a priority encoder which places the lowest priority on the context which issued the last instruction can be employed to implement this next context selection.

For the interleaved scheme a further complication arises, as the processor needs to handle the situation where the last context becomes unavailable. This did not occur for the blocked processor because context availability depended only on the number of contexts loaded. One way to handle this situation is to squash all instructions in the pipeline, stalling the processor when all EPCs have been loaded. When the first context becomes available again, the processor is restarted by issuing instructions from that context. As an alternative, the processor could simply stall the pipeline without squashing any instructions. If the first context to receive a reply was also the last context to become unavailable, the pipeline is unfrozen at no penalty. Otherwise, the instructions in the pipeline are squashed and the newly available context starts executing. This second alternative is likely to give only slightly higher performance (most of the time the last context to become unavailable will not be the first to become available). However, since the processor needs the ability to stall the pipeline on cache misses to support both single-context

execution and adaptive stalling, implementation of this second option may also be simpler than the first.

7.3.2 Context Interleaving

Process state is changing each cycle in the interleaved processor, and Figure 7.4 shows exact cycle during which the various state registers need to transition. The cycle-by-cycle interleaving of contexts also affects the operation of the pipeline itself. First, the pipeline needs to treat all instructions as if their operands came from one large register file. This implies that all pipeline forwarding logic needs to compare not only the register address, but also the context identifier when determining which items to forward. The processor also needs a scoreboard per context to track instruction dependencies.

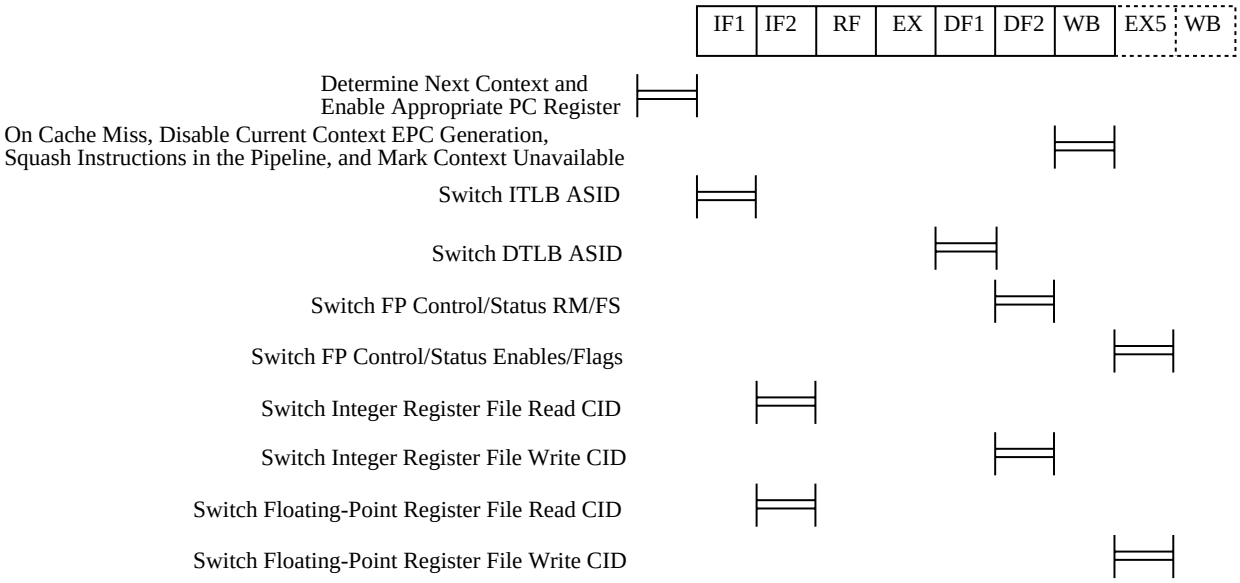


Figure 7.4: Timeline for control of the interleaved multiple-context processor pipeline.

7.4 State Replication

As for the blocked scheme, the per-process state needing replication can be broken into three components: the program counter, the process-specific portion of the processor status word, and the register files. We have already covered program counter replication in Section 7.1. We now discuss the PSW and register file replication issues.

The same PSW register modifications are needed by the interleaved processor as was done for the blocked processor. However, since the state from any given context needs to be available continuously, the registers need to be able to switch state every cycle. This implies that replication of the floating-point control/status register is more difficult for the interleaved scheme, as it is user writable and providing the illusion of a single floating-point control/status register per context under cycle-by-cycle context interleaving is complicated, as shown in Appendix E.

For the interleaved scheme, the register file needs to provide the ability to read and write the registers of any of the contexts during any given cycle (in fact, it may be reading from one context and writing from another context during the same cycle). Therefore, the fully replicated register file discussed in the previous chapter will need to be supported. The replicated register file does have a longer access time than either an apportioned register file (where only the memory cell for the active context has full drive capability) or the single-context register file. This increased access time may well be acceptable for many processors where the register file access is not on the critical path. For those processors where the increase in register file access time will impact the critical path, either a more aggressive register file design will need to be employed or a reduction in the number of contexts will need to be considered. In the worse case, the additional cycle time resulting from the interleaved scheme will need to be carefully evaluated against the benefits of adding multiple contexts.

7.5 Summary

We have seen that the implementation of the interleaved scheme is more complex than the blocked scheme. The blocked scheme is simpler to implement because most of the time it behaves like a single-context processor, changing state between contexts only at the point of the switch itself. On the other hand, the interleaved scheme is continually cycling through the active contexts, requiring the processor state to potentially change each cycle. While the complexity of the interleaved scheme is greater, this extra complexity is not overwhelming, especially compared against the extra complexity required to support multiple instruction issue with out-of-order execution [Joh89], which is starting to appear in off-the-shelf microprocessors [Mot91, Kru91]. The largest difficulty to be surmounted by the interleaved scheme is keeping the access time of the replicated register file to an acceptable level. For many of the current generation of processors [Cyp90, Hei93], the register file access is not in the critical path and increasing the register file access time by a small amount should not impact the processor cycle time. However

with the cycle time of processors getting smaller and smaller, the register file access time may become an issue in the future.

For any design the true complexity cost only becomes apparent when doing an actual implementation. The performance advantages of the interleaved scheme are enticing and the implementation cost seems manageable, making implementation of such a processor a worthwhile effort.

Chapter 8

Conclusions

The potential benefits from using multiple contexts to tolerate memory, synchronization, and instruction latency makes them an attractive candidate for addition to microprocessors. Unfortunately, existing multiple-context designs did not provide both good single-thread performance and fast context switching. Good single-thread performance is important because not all applications can be broken into multiple threads. Fast context switching is crucial for tolerating shorter latencies. Early, *fine-grained* multiple-context processors switched contexts every cycle, supporting a very low-cost context switch. However, any given context could only have a single instruction in the pipeline, causing the single-context performance to suffer. *Blocked* multiple-context processors only switch contexts when the processor encounters a long-latency operation, providing good single-thread performance. However, the context switch cost for the blocked processor is fairly large, as the decision to switch is made late in the pipeline and the pipeline must be flushed before executing the next context. This thesis has shown that is it possible to combine the best aspects of the fine-grained and blocked approaches by designing a multiple-context processor which switches contexts on a cycle-by-cycle basis while providing data caching and full pipeline interlocks. This *interleaved* processor provides both the quick context switch and good single-thread performance.

The interleaved multiple-context processor exhibits a significant performance advantage over the blocked processor for both multiprocessor and uniprocessor environments. Table 8.1 summarizes the speedups we saw for the SPLASH applications on a multiprocessor similar to the Stanford DASH, while Table 8.2 summarizes the throughput improvement for several multiprogramming workloads (consisting of members of the SPEC and SPLASH suites) running on a high-performance workstation. The interleaved scheme results in significantly better speedups

and throughput increases for both environments.

Table 8.1: Summary of speedups with eight contexts per processor for the SPLASH applications.

Scheme	Barnes	Cholesky	Locus	MP3D	Ocean	PTHOR	Water	Mean
Blocked	2.14	1.00	1.14	2.91	1.48	1.14	1.18	1.46
Interleaved	2.92	1.00	1.31	3.46	1.56	1.16	2.05	1.74

Table 8.2: Summary of throughput increase with four contexts per processor.

Scheme	**IC**	**DC**	**DT**	**FP**	**R0**	**R1**	**SP**	Mean
Blocked	1%	23%	9%	21%	2%	8%	15%	11%
Interleaved	28%	65%	46%	79%	41%	54%	44%	50%

The interleaved scheme must be able to be implemented with reasonable cost and complexity if these performance benefits are to be reaped. One of the major costs for multiple-context processors is in providing a lockup-free cache. This thesis examined the requirements placed on lockup-free caches by prefetching, relaxed-memory consistency, nonblocking loads, and multiple contexts. We showed that it is possible to build a lockup-free cache which supports all methods of generating multiple outstanding requests by combining a pending state in the cache with separate transaction buffers for keeping track of (a) write valid bits, (b) result registers for nonblocking loads, and (c) interleaved context availability. Employing a pending state in the cache is important to allow compatible requests to the same cache line to be merged and to prevent incompatible requests from allocating the same cache line.

Both the blocked and interleaved schemes have additional implementation requirements, and this thesis explored those requirements. Both schemes require process-specific state to be replicated per context. The main costs for saving process state between context switches reside in the replication of the register file. The increase in chip area resulting from replicating the register file is fairly small, however the replication does result in a larger register file access time. This increase in access time should not impact the critical path of most modern processors, however, it is an important cost of adding multiple contexts to a processor. This increase in access time is the same as would be found when comparing the register file of a processor with register windows [Cyp90] against a processor with a single register file. Thus, as long as processors with register windows can be made to have competitive cycle times, a multiple-context processor

should be able to also achieve these cycle times.

In addition to replication of process-specific state, the blocked scheme requires the ability to switch between contexts, while the interleaved scheme requires the ability to issue instructions from multiple streams to the pipeline and to change context availability to tolerate long-latency operations. Providing an instruction issue unit and pipeline capable of handling multiple instruction streams increases the complexity of the interleaved scheme beyond that of the blocked scheme. Despite this increased complexity, the implementation cost of the interleaved scheme still seems manageable, and because the performance of the interleaved scheme is so promising, the next logical step for this research is to build a multiple-context processor using the interleaved scheme to truly evaluate this additional complexity. In addition to providing concrete evidence that such a processor can be built, the processor would also open the door for interesting research into many of the unresolved issues involving multiple-context processors, such as the interaction of the operating system with multiple contexts and the effects of combining multiple contexts with prefetch.

Appendix A

Details of the Tango-Lite Based Simulator

In this appendix we discuss how our simulation environment allows execution-driven simulation of one pipeline on a machine with a different pipeline.

Tango-Lite is the lightweight threads-based successor to the Tango reference generator [GD90, DGH91] and allows parallel programs to be simulated on a uniprocessor. The real power of Tango-Lite comes from combining the reference generator with a detailed architectural simulator. This allows simulation of applications running on realistic parallel architectures.

Our architectural simulator models both the pipeline and memory system in detail. Tango-Lite uses execution-driven simulation [GH93], however because we are simulating a pipeline different than the pipeline of the processor on which we perform our simulations, we need to produce an "executable" for the target pipeline. This target executable is then used by the pipeline simulator to generate accurate pipeline timing.

The steps involved in generating the target executable are shown in Figure A.1. First, ANL macros in the application source are expanded using a special Tango-Lite macro library to make the synchronization references visible to the Tango-Lite scheduler. The result of the macro expansion is a set of plain C or Fortran source files. This C or Fortran source is then compiled down to assembly code. Both the application assembly files and the assembled code for key library routines are then assembled (and scheduled) for the target pipeline. Finally, these target object files are linked to create the target executable.

Now that we have the target executable, the next step is to generate the actual Tango-Lite program that will perform the simulation. The steps for this process are shown in Figure A.2.

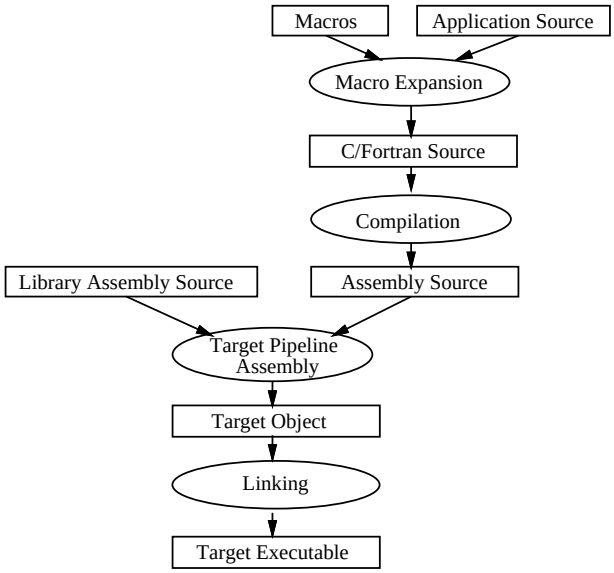


Figure A.1: Target "executable" compilation path.

First, the pipeline and memory simulator is compiled down to object code. Then, the application assembly files and assembly code for the key library routines are augmented with additional instructions to make the memory references and basic block entry points available to Tango-Lite. Since we will be simulating the target pipeline, the addresses used in the basic block augmentation are provided by the target executable generated in Figure A.1.

At this point, we have assembly code containing enough hooks to allow Tango-Lite to properly simulate the pipeline and memory system for the application and its key library routines. This augmented assembly code is assembled and linked with the architectural simulator and the libraries which could not be instrumented (due to being used by the simulator routines) to form the simulator executable. Before starting the application simulation, the simulator reads in the target executable to allow the pipeline behavior of the application to be properly modeled.

Figure A.3 shows the run-time relationship between the Tango-Lite scheduler, the augmented application threads, and the system simulation routines. Tango-Lite simulates the execution of parallel programs on a uniprocessor by switching between threads. The scheduler maintains a local clock for each thread, and schedules the thread furthest back in time to execute next. Tango-Lite is responsible for scheduling both the application and memory simulator threads. An application thread runs until it generates a synchronization event, memory reference, or basic block entry. At this point, the application thread issues a reference to the processor simulator and

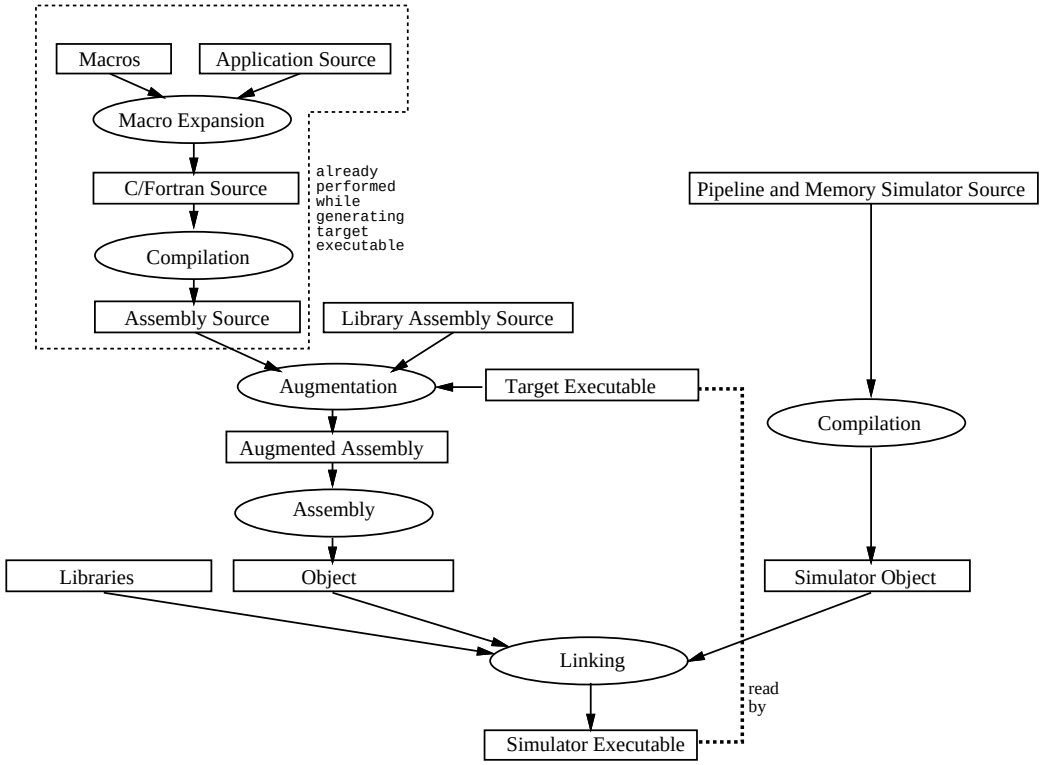


Figure A.2: Tango-Lite compilation path.

blocks. The processor simulator will then simulate the reference executing on the processor's pipeline, passing control to the memory simulator if necessary. When the operation completes, the processor simulator updates the time of the blocked application thread and frees it to generate further references.

By using execution driven-simulation where the addresses generated during execution index into the properly-scheduled executable, cycle-accurate simulation of one pipeline on a machine using another pipeline becomes possible.

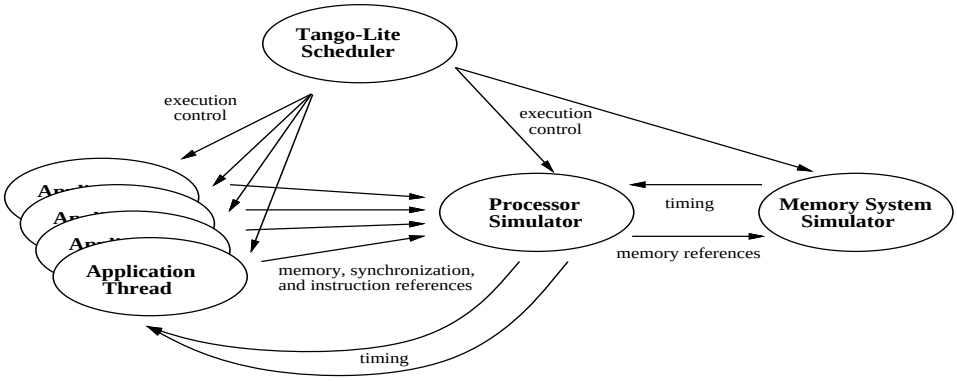


Figure A.3: Tango-Lite execution model.

Appendix B

SPLASH Application Descriptions

This appendix briefly discusses the computational behavior and important data structures for each application in the SPLASH suite. A more detailed discussion of the applications can be found in [SWG92].

B.1 Barnes-Hut

Barnes-Hut performs a hierarchical N-body gravitational simulation [SHG92]. Each body is modeled as a point mass, exerting forces on all other bodies in the system. To reduce the complexity of the algorithm from $O(n^2)$ to $O(n \times log(n))$, a set of bodies far enough away is modeled as a single body residing at its center of mass. To implement this enhancement, physical space is recursively divided into octrees, until there is at most a single body in each space cell. Each element of the octree maintains information on the center of mass of all particles beneath it in the tree.

The primary data structure for Barnes-Hut is this octree, which is implemented as an array of bodies and an array of space cells. The bodies and cells are then linked together to form the octree. Each processor is statically assigned a set of bodies for the duration of a time-step. The processor calculates the forces of all other bodies on its subset of the bodies. This is done by traversing the octree starting from the root. If at any point, the center of mass of the cell is far enough away, the entire subtree under that cell is approximated by its center of mass. Otherwise, the cell is "opened", and each of its subcells recursively visited. After the forces have been computed, the bodies are then moved in response to these forces. Finally, the tree is regenerated and repartitioned for the next time-step. Barriers are used for synchronization between different

phases of the computation and between successive time-steps.

B.2 Cholesky

Cholesky performs a parallel Cholesky factorization of a sparse positive definite matrix. It uses a dynamic version of the supernodal fan-out method [RG90]. A supernode is a set of columns with a nearly identical non-zero structure. Supernodes which have been completely updated are kept on a global task queue. Each supernode is updated only from supernodes to its left in the matrix. The update status of the supernode is determined by an incoming counter. This counter is initialized to the total number of updates to be received and decremented with each update. Once the counter reaches zero, the supernode has received all of its updates and is placed on the task queue. This supernode will then be pulled off the queue by an available processor and used to update other supernodes.

The principal data structure in Cholesky is the structure holding the matrix itself. Since the matrix is sparse, it is stored in a compressed fashion nearly identical to that used in SPARSPAK. Most of the computation in Cholesky consists of the addition of a multiple of one column of the matrix to another column. Contention between processors occurs when they access the task queue or update the same supernode. A set of locks is used to ensure mutual exclusion for these operations.

B.3 LocusRoute

LocusRoute [Ros88] is a VLSI standard cell router. It uses multiple processors to route several wires in parallel. To facilitate parallel routing, the circuit is divided into regions, each with its task queue. All wires with their leftmost pin in a region are placed on that region's task queue. A wire is worked on by a single processor. Each processor has a preferred task queue for retrieving wires, however if this queue is empty, the processor will scavenge from other queues.

LocusRoute attempts to minimize circuit area by routing wires through regions (routing cells) which have few other wires running through them. It does this by evaluating many routes for a wire, calculating a cost function for each route. The lowest cost route is then selected. The cost function is based on the number of wires already routed through the cells selected by the route. Information on the number of wires routed through each cell is maintained in the *cost array*.

The cost array is the primary data structure in LocusRoute. Each row of the cost array

corresponds to a routing channel in the circuit. The elements of the cost array maintain a count of the number of wires that have been routed through that routing cell. The cost array is read repeatly when evaluating possible routes for a wire. Once the final route is selected, the routing cells are updated to reflect the addition of the new wire. The cost array is not locked during this update, as the effect of occasional contention on the final circuit quality is tolerable.

B.4 MP3D

MP3D is a three-dimensional, particle-based simulator for rarefied air flow. It is used to study shock waves created by an object flying at high speed through the upper atmosphere. The overall computation consists of evaluating the positions and velocities of the particles over several time-steps while gathering relevant statistics. Each molecule can be treated independently during a time-step, as molecular collisions are statistically determined. Since they are independent, the particles are statically divided among the processors. During each time-step, each molecule is moved according to its velocity vector, taking into account collisions with the boundaries, the object being simulated, and other molecules. A single barrier is used to separate different phases of the program. Six separate invocations of this barrier are encountered in each time-step.

The main data structures in MP3D are the particle array and the space array. The particle array contains all of the molecules, recording their positions, velocities, and other molecule information. The three-dimensional wind tunnel is divided into a number of cells, which form the space array. Each space cell contains information on the boundaries of wind tunnel and the location of the object. Statistics information is also maintained in the space cells.

B.5 Ocean

Ocean [SH92] studies the role of eddy and boundary currents in influencing large-scale ocean movements. The program starts with a set of spatial partial differential equations. These equations are transformed into difference equations which are solved on two-dimensional fixed-size grids. The simulation is performed for many time-steps until the eddies and mean ocean flow attain a mutual balance.

The principal data structures for Ocean are the two-dimensional arrays holding the values of the various functions associated with the model equations. There are 25 double-precision two-dimensional arrays in all. These arrays are partitioned among the processors, with each processor

responsible for a contiguous set of columns. The array size is allocated as 300x300, as allocation by a power of two causes mapping problems in the cache. Problems smaller than 300x300 use only a portion of the arrays. The primary form of synchronization is a number of barriers which are used to separate both the phases of computation during a time-step and subsequent time-steps.

B.6 PTHOR

PTHOR [Sou92] is a parallel, event-driven, logic simulator. It is based on a conservative distributed-time algorithm which is a modified version of the Chandy-Misra algorithm [CM81]. The Chandy-Misra algorithm allows each simulation element to advance its own time value independently of other elements.

The primary data structures in PTHOR are the element and node structures, and a set of distributed task queues. The element structure holds the logic elements while the node structure contains information on the nets linking the logic elements. The task queues holding activated elements are distributed to increase locality; as in LocusRoute, if a process runs out of work it will search for work in other task queues. PTHOR has two distinct, alternating phases: element evaluation and deadlock resolution. During element evaluation, a processor removes elements from its task queue and computes the new output behavior (if any) of the element. It then uses the node structure to determine which elements are affected by the new output value and places those newly activated elements on the appropriate task queue. It is possible for the simulation to deadlock, which is manifested by all task queues being empty. PTHOR then performs a deadlock resolution phase which results in new elements being activated.

B.7 Water

Water is an N-body application adapted from the Perfect Club Benchmarks [BCKK88]. It evaluates forces and potentials in a system of water molecules in the liquid state. The computation is performed over a number of time-steps, until the system reaches a steady-state. Each time-step consists of setting up and solving the Newtonian equations of motion for water molecules in a cubical box with periodic boundary conditions. For the parallel version, molecules are statically divided among the processors.

The primary data structure in Water is a three-dimensional array of molecular structures. Each molecular structure contains a three dimensional array, indexed by simulation variable type (e.g.

displacement, velocity, force), spatial direction, and atom number within the water molecule. The structure also contains a smaller array with three entries per molecule. Barriers are used to separate both successive time-steps and phases within a time-step. In addition to the barriers, it is possible for multiple processes to be computing intermolecular interactions involving the same molecule. A lock per molecule is used to ensure mutually exclusive updates.

Appendix C

Coherent Cache Design

Many protocols for maintaining cache coherence for shared-memory multiprocessors have been developed. These protocols can be broken into two parts, the protocol providing coherence between the processors, and the protocol maintaining coherence within a single processor's cache hierarchy.

In this appendix we will examine both portions of the coherence protocol. A large number of protocols for coherence between processors have been developed, and we will start by summarizing these protocols, emphasizing the impact they have on the design of the caches themselves. The issues for maintaining coherence within a single processor's cache hierarchy have not been explored in much detail, and the bulk of this appendix will be devoted to developing an intra-hierarchy coherence protocol for a general cache hierarchy. This protocol is used in Chapter 5 as the base protocol between the caches to which lockup-free capability is added.

C.1 Coherence Between Processing Nodes

Many protocols providing coherence between processors have been developed — both for snoopy-bus and scalable multiprocessors. These protocols allow a processor to modify a memory location either by first obtaining an exclusive copy which it can then modify (invalidate-based) [PP84, KEW+85, EK89] or by broadcasting the new data to all cached copies of the memory location (update-based) [TS87]. Scalable protocols have primarily been invalidate-based [LLG+90, JLGS90, CKA91, Sim92, Web93], as the update-based protocols lack write atomicity, that is the write does not appear to happen as an atomic unit when viewed by multiple observers. This lack of write atomicity makes implementation of cache coherence and the memory consistency

model difficult. On the other hand, snoopy-bus protocols have used both schemes, as the bus provides atomicity for the write. However, the performance of the update-based protocols is much more sensitive to the application sharing behavior [AB86, EK88], and most recent bus-based multiprocessors use an invalidate-based variant [BJS88, TS90, GW94]. Exploring the issues for coherence between the processors is beyond the scope of this work — issues for scalable protocols are covered in detail in [Len92, Sim92, Web93]. The commonality between these scalable protocol proposals is that there is an entity which is responsible for tracking outstanding requests, in order to implement both normal message handling and to detect and correct coherence races.

As an example of a coherence race, if the network does not guarantee strict point-to-point ordering, (e.g due to adaptive routing [CK92] or due to multiple networks such as in the DASH interconnect [LLG+90]), it is possible for an invalidation to bypass the reply it was supposed to invalidate. This coherence race must be detected and corrected by the entity tracking outstanding requests. The external protocol may also allow situations in which negative acknowledgements may be generated in response to a request. Using the DASH protocol as an example again, a negative acknowledgement can be received in response to memory request whenever multiple processors simultaneously try to read and write the same memory line. Requests which receive negative acknowledgements must then be repeated.

The entity tracking these external requests can either be located outside of the processor's cache hierarchy (as done for the Remote Access Cache of the DASH multiprocessor [LLG+90]) or can be integrated into the cache hierarchy (as in the transaction buffers of the Alewife [KCA92]). Determining whether these races should be handled outside the cache hierarchy depends on a number of implementation issues which are outside the scope of this thesis. [Len92, Sim92, KCA92, Web93] discuss these issues in detail. For this discussion, since this entity is logically separate from cache hierarchy, we will assume it is also physically separate. Thus the caches will only be responsible for maintaining coherence within the cache hierarchy.

C.2 Coherence Within the Cache Hierarchy

The issues involved in maintaining coherence between caches within a processor's hierarchy have not been explored in detail. Historically, most multilevel caches have consisted of a writethrough primary cache backed by a writeback secondary cache [BJS88, Dig92b]. Maintaining coherence between the writethrough and writeback cache in these systems is very straightforward. The primary writethrough cache simply passes all misses and write requests to the secondary writeback

cache, and the secondary writeback cache invalidates from the primary cache any lines that are removed due to coherence operations.

More recently, however, a number of microprocessors supporting two levels of writeback caches have appeared [Mot91, Hei93]. Maintaining coherence within this environment is more difficult than with a single writeback cache, as a modified cache line can potentially live in multiple levels of the cache hierarchy. In this section we will explore the issues for maintaining coherence within a cache hierarchy which may contain multiple levels of writeback caches.

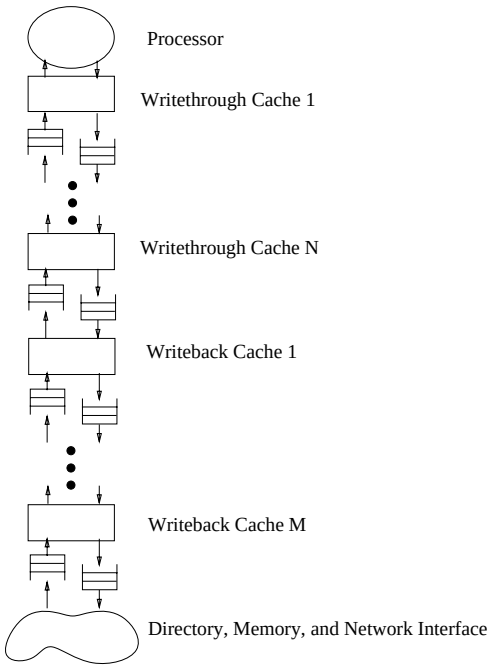


Figure C.1: General cache hierarchy.

Our general hierarchy is given in Figure C.1. Caches closer to the processor are referred to as upper or nearer; caches closer to the network interface are referred to as lower or outer. The processor sits at the top of the cache hierarchy. It generates requests to and accepts replies from the level one cache. At the lowermost end of the cache hierarchy is the external interface. Request for data not in the cache hierarchy are sent to the external interface, which eventually replies to such requests. In addition, from the viewpoint of the cache hierarchy, the external interface can spontaneously generate coherence requests. Note that this requires that the caches be dual-ported, as they will need to accept external requests while a processor request is outstanding. For simplicity, we assume the line size at all levels in the cache hierarchy is the same. Extending our

intra-hierarchy protocol to handle different line sizes at each level is a straightforward extension discussed at the end of Section C.2.2. Finally, in order to keep the design of the coherence protocol reasonable, it is important that the cache hierarchy maintain *multi-level inclusion* [BW88]. Inclusion refers to the property that an item contained in a given cache will also be found in all caches further from the processor.

To support coherence within this hierarchy, the cache hierarchy must maintain coherence information for each cache line, and be able to handle coherence races within the hierarchy. We now look at these requirements in turn.

C.2.1 Maintaining Coherence for Each Cache Line

All invalidate-based schemes have three states at their heart: *exclusive*, *shared*, and *invalid* [AB86]. The exclusive state provides for the existence of a single read-write copy in the system. The shared state allows multiple processors to hold read-only copies of the cache line. Cache lines enter the invalid state whenever coherence transactions remove the line from the cache. Since four states can be encoded in two bits, many protocols provide an fourth state to enhance cache performance. For example, the Illinois protocol [PP84] adds a private-unmodified state to the base three states. The private-unmodified distinguishes between a read-only copy in multiple caches and a read-only copy in a single cache. In the case where the read-only copy resides in a single cache, a write can proceed immediately, changing the state of the cache to exclusive in the process. With multiple levels of writeback caches a need arises to use a fourth cache state to distinguish between an exclusive copy that is the most recently modified and exclusive copies that may be potentially out of date.

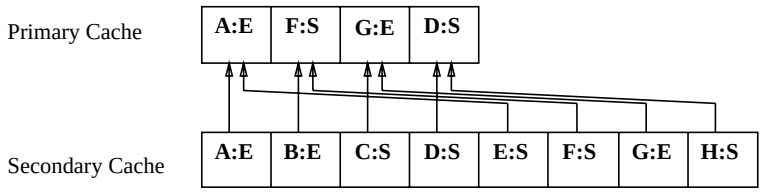


Figure C.2: Initial state for example illustrating complications due to providing only three coherence states.

While this modified copy could be *implicitly* identified as the copy nearest to the processor, this leads to complications when the line needs to be flushed (or copied back) due to an external coherence event. To illustrate this problem with an example, we start with the initial cache state

for a two-level hierarchy of writeback caches in Figure C.2. Each box in the figure represents a cache line. The labeling of the cache lines is in the form *address:state*. The arrows between the caches show the mapping of lines from the secondary cache to the primary cache.

In the figure, lines A and B are both in the exclusive state in the secondary cache. B is the most up-to-date copy, as it is the closest copy to the processor. However, A is potentially out-of-date, as there is also a copy of A in the primary cache. When a flush request occurs at the secondary cache, the cache is unsure of whether it has the most up-to-date copy (as for B) or a stale copy (as for A). Thus the secondary cache must send a flush request to the primary cache. However, a problem arises as the secondary cannot throw away its copy because it may be the most recent. One solution to this problem would be to stall the flush request until either the flush reply or a negative acknowledgement (signifying there was no copy in the primary cache) returns. This is the solution adopted for the R4000 [Hei93]. However, there is a problem with this solution. Since the flush is delayed until the reply returns, we have increased the serialization of flush requests. In addition, the flush forces other requests and replies behind it to wait, increasing the latency for these other operations.

Table C.1: Cache states used to support the intra-hierarchy coherence protocol.

State	Description
D	exclusive, up-to-date copy (Dirty)
T	exclusive, out-of-date copy (sTale)
S	shared copy (Shared)
I	invalid (Invalid)

By making this up-to-date status explicit, this serialization can be avoided. By splitting the exclusive state into two states: *dirty* and *stale*, to distinguish between the up-to-date and out-of-date copies of the exclusive data, a flush or copyback request can proceed immediately, as either the exclusive copy is stale and can be discarded, or is dirty and can be sent in response to the request. We add this explicit tagging of the up-to-date copy to our base states, resulting in the four states listed in Table C.1.

Messages are sent between caches within the hierarchy to request copies of data and maintain coherence. In addition, messages are required to handle requests from both the processor and external interface. The base set of messages used within the cache hierarchy is presented in Table C.2. Most messages have both a request and reply version. For example, an exclusive request ($Exc\ Rq$) is generated in response to a processor write request. This exclusive request

Table C.2: Coherence messages.

Message	Description	Direction
$Exc\ Rq$	Exclusive Request	From upper cache to lower cache
$Shd\ Rq$	Shared Request	
$Fsh\ Rp$	Flush Reply	
$Cbk\ Rp$	Copyback Reply	
$Fsh\ Rq$	Flush Request	From lower cache to upper cache
$Cbk\ Rq$	Copyback Request	
$Inv\ Rq$	Invalidate Request	
$Exc\ Rp$	Exclusive Reply	
$Shd\ Rp$	Shared Reply	

eventually generates an exclusive reply ($Exc\ Rp$). The lone exception to this message pairing is the invalidate request, $Inv\ Rq$, which has no corresponding reply, as invalidates can be immediately acknowledged at the external interface under our cache hierarchy assumptions [Gha92]. The function of most of the messages is self-explanatory, with the exception of flush and copyback. The flush request asks for exclusive data from the cache, requiring that the cache also invalidate its copy. The copyback request also asks for exclusive data from the cache, but allows the cache to keep a shared copy. The flush and copyback replies then return the exclusive data.

C.2.2 Handling Coherence Races Within the Hierarchy

With the addition of this fourth state, we can now develop a protocol which can handle all intra-hierarchy coherence races. Under this protocol, the caches are completely decoupled by the buffers between them. This means that a cache can always process a request based solely on its current state, not on any other states in the hierarchy. In addition, no snooping between incoming and outgoing buffers is necessary to handle race conditions such as a flush request targeting a line that has just been replaced from the cache. Instead, the protocol insures that the cache hierarchy always properly responds to all combinations of external and processor requests.

In this protocol, shared and exclusive requests by the processor percolate outwards until they either encounter a cache which has the requested line in the proper state (shared or dirty for the shared request, dirty for the exclusive request) or are sent to the external interface. Shared replies are loaded into each level of the cache hierarchy in a shared state, while exclusive replies are loaded into all levels of the writeback cache hierarchy except the innermost in the stale state. In the innermost writeback cache the data is loaded in the dirty state, as this will be the most

up-to-date copy. The writethrough caches cannot own cache lines and all data in the writethrough caches is therefore implicitly in the shared state. External flush and copyback requests percolate upward from the external interface, modifying the state of the cache lines at each level (to invalid for the flush, shared for the copyback request) until they encounter the dirty copy, at which point a reply is generated for the external interface. Invalidates simply traverse up the cache hierarchy, invalidating all shared lines they encounter.

All the protocol tables use the symbols given in Table C.3. The cache state subscripts H and M are used to signify a hit or miss respectively. We will start our discussion of the coherence protocol by focusing on the writeback caches, and then turn to the writethrough caches.

Table C.3: Cache coherence protocol key.

$\rightarrow$	state transition
$\downarrow$	message to lower-level cache
$\uparrow$	message to upper-level cache
$\perp$	read data from cache into message
$\top$	write data from message into cache
$\emptyset$	error (should not occur)

The protocol for the uppermost writeback cache is given in Table C.4. Writeback caches *2* to *M-1* use the protocol presented in Table C.5. Writeback cache *M* also uses the protocol given in Table C.5 with two minor differences. First, $Fsh\,Rq$ and $Cbk\,Rq$ messages may be able to hit a shared copy in the cache due to external protocol races. For these races, the requests can be dropped by the cache. Second, an $Inv\,Rq$ may hit a dirty or stale copy in the cache due to external protocol races, and can also be ignored by the cache.

We can explain the protocol for the writeback caches by concentrating on Table C.5. Starting from the first line of the table, a shared request can be satisfied if the cache contains the requested line in either a dirty or shared state. If the cache is in a dirty state, it returns an exclusive reply and changes to stale, as its copy may soon be out-of-date. If in a shared state, a shared reply is returned. If the shared request cannot be satisfied, it is forwarded to the next cache. The old cache line is not replaced at this point; it will be replaced when the reply returns. When the shared reply returns, it loads the data into the cache in the shared state and then forwards the reply to the upper cache. If the line being replaced by the reply is in the dirty state, the dirty data is written back (generating a flush reply). If the line is in a stale or shared state, an explicit flush or invalidate request needs to be generated for the line being replaced to maintain inclusion

Table C.4: Cache coherence protocol — uppermost writeback cache.

Trans-action	**Cache State**				
	D_H	D_M	S_H	S_M	I
$Shd\ Rq$	$\perp Re\ p\ ly\uparrow$	$Shd\ Rq\downarrow$	$\perp Re\ p\ l\ y$	$\$hd\ Rq\downarrow$	$Shd\ Rq\downarrow$
$Shd\ Rp$	$\emptyset$	$\rightarrow S$ $\perp Fsh\ Rp\downarrow$ $Inv\ Rq\uparrow$ $\top Re\ p\ l\ y\uparrow$	$\emptyset$	$\rightarrow S$ $Inv\ Rq\uparrow\top Re\ p\ l\ y\uparrow$	$\rightarrow S$
$Exc\ Rq$	$\perp Re\ p\ l\ y$	$\uparrow Exc\ Rq\downarrow$	$Exc\ Rq$	$Exc\ Rq\downarrow$	$Exc\ Rq\downarrow$
$Exc\ Rp$	$\emptyset$	$\rightarrow D$ $\perp Fsh\ Rp\downarrow\top Re\ p\ l\ y$ $Inv\ Rq\uparrow$ $\top Re\ p\ l\ y\uparrow$	$\rightarrow D$	$\rightarrow D$ $Inv\ Rq\uparrow\top Re\ p\ l\ y\uparrow$	$\rightarrow D$
$Fsh\ Rq$	$\rightarrow I$ $\perp Fsh\ Rp\downarrow$ $Inv\ Rq\uparrow$		$\emptyset$		
$Cbk\ Rq$	$\rightarrow S$ $\perp Cbk\ Rp\downarrow$		$\emptyset$		
$Inv\ Rq$	$\emptyset$		$\rightarrow I$ $Inv\ Rq\uparrow$		

within the hierarchy.

The protocol behaves similarly for the exclusive request and reply. The exclusive request differs from the shared request in that it can only be satisfied if the cache is in the dirty state. For the exclusive reply, the data is loaded into the cache and the cache changes to the stale state. The reply is forwarded to the next upper level along with the proper flush or invalidate request. When the reply reaches the uppermost writeback cache, the data is loaded in the dirty state, as it will be the most up-to-date copy of the exclusive data.

The external coherence protocol can generate flush, copyback, and invalidate requests targeted at this node. The flush request invalidates all stale copies encountered as it travels up the cache hierarchy. When the dirty copy is reached, the flush request causes the data to be written back and the line to transition to the invalid state. The copyback request transitions all copies to shared as it traverses up the cache hierarchy. When the dirty copy is reached a copyback reply is generated, and the line transitions to the shared state. As this copyback reply returns down the cache hierarchy, it updates all shared copies along the way with the most up-to-date data. Finally, the invalidate request simply propagates up the cache hierarchy, invalidating shared copies. Once it reaches a cache without a shared copy, it stops, as the inclusion property guarantees that the

Table C.5: Cache coherence protocol — writeback caches except uppermost.

Trans-action	**Cache State** D_H	D_M	T_H	T_M	S_H	S_M	I
$Shd\ Rq$	$\rightarrow T$ $\perp\ Exc\ Rp$	$\uparrow Shd\ Rq\ \downarrow$	$\emptyset$	$Shd\ Rq\ \downarrow$	$\perp Shd\ Rp$	$\uparrow Shd\ Rq\ \downarrow$	$Shd\ Rq\ \downarrow$
$Shd\ Rp$	$\emptyset$	$\rightarrow S$ $\perp\ Fsh\ Rp\ \downarrow$ $\top\ Shd\ Rp\ \uparrow$	$\emptyset$	$\rightarrow S$ $Fsh\ Rq\ \uparrow$ $\top\ Shd\ Rp\ \uparrow$	$\emptyset$	$\rightarrow S$ $Inv\ Rq\ \uparrow$ $\top\ Shd\ Rp\ \uparrow$	$\rightarrow S$ $\top\ Shd\ Rp\ \uparrow$
$Exc\ Rq$	$\rightarrow T$ $\perp\ Exc\ Rp$	$\uparrow Exc\ Rq\ \downarrow$	$\emptyset$	$Exc\ Rq\ \downarrow$	$Exc\ Rq\ \downarrow$	$Exc\ Rq\ \downarrow$	$Exc\ Rq\ \downarrow$
$Exc\ Rp$	$\emptyset$	$\rightarrow T$ $\perp\ Fsh\ Rp\ \downarrow$ $\top\ Exc\ Rp\ \uparrow$	$\emptyset$	$\rightarrow T$ $Fsh\ Rq\ \uparrow$ $\top\ Exc\ Rp\ \uparrow$	$\rightarrow T$ $\top\ Exc\ Rp\ \uparrow$	$\rightarrow T$ $\uparrow Inv\ Rq\ \uparrow$ $\top\ Exc\ Rp\ \uparrow$	$\rightarrow T$ $\top\ Exc\ Rp\ \uparrow$
$Fsh\ Rq$	$\rightarrow I$ $\perp\ Fsh\ Rp\ \downarrow$		$\rightarrow I$ $Fsh\ Rq\ \uparrow$		$\emptyset$		
$Fsh\ Rp$	$\emptyset$	$Fsh\ Rp\ \downarrow$	$\rightarrow D$ $\top$	$Fsh\ Rp\ \downarrow$	$\top\ Cbk\ Rp$	$\downarrow Fsh\ Rp\ \downarrow$	$Fsh\ Rp\ \downarrow$
$Cbk\ Rq$	$\rightarrow S$ $\perp\ Cbk\ Rp$	$\downarrow$	$\rightarrow S$ $Cbk\ Rq\ \uparrow$		$\emptyset$		
$Cbk\ Rp$	$\emptyset$	$Fsh\ Rp\ \downarrow$	$\emptyset$	$Fsh\ Rp\ \downarrow$	$\top\ Cbk\ Rp$	$\downarrow Fsh\ Rp\ \downarrow$	$Fsh\ Rp\ \downarrow$
$Inv\ Rq$	$\emptyset$		$\emptyset$		$\rightarrow I$ $Inv\ Rq\ \uparrow$		

line will not exist in any lower caches.

The protocol for the writethrough caches is given in Table C.6. This table employs several new states and messages. First, the distinction between the $Exc\ Rp$ and $Shd\ Rp$ messages no longer exists for the writethrough caches, so all reply messages are simply encoded as $Reply$. Second, the states used in the writethrough cache are simply V, valid, and I, invalid. Valid information is kept on a word basis. The states V_H and I_H in the table are interpreted differently depending on the message. For $Shd\ Rq$ and $Exc\ Rq$, they correspond to the valid bit of the targeted word. For $Reply$, the actions specified in the table are taken for each word. Finally, for $Inv\ Rq$, the V_H actions are taken if any word is valid, the I_H actions otherwise. Note that Table C.6 assumes a write-validate policy [Jou93] coupled with a fetch of the cache line which does not stall the processor.

Table C.6: Cache coherence protocol — writethrough caches.

Trans-action	**Cache State**		
	V_H	I_H	$Miss$
$Shd\ Rq$	$\perp Reply$	$\uparrow Shd\ Rq\ \downarrow$	$Shd\ Rq\ \downarrow$
$Exc\ Rq$	$\top Exc\ Rq$	$\rightarrow V_{mine}$ $\downarrow \top Exc\ Rq$	$\rightarrow V_{mine,other}$ $\downarrow\ \top Exc\ Rq\ \downarrow$
$Reply$	$Reply\ \uparrow$	$\rightarrow V$ $\top Reply\ \uparrow$	$\rightarrow V_{all}$ $\top Reply\ \uparrow$
$Inv\ Rq$	$\rightarrow I_{all}$ $Inv\ Rq\ \uparrow$		

Support for Different Line Sizes

Cache hierarchies which do not employ the same line size throughout the hierarchy commonly have line sizes which are strictly increasing as the caches get further from the processor [Hei93]. Under these constraints, a single dirty/stale distinction per cache line will not suffice. Instead, a cache line will need to keep separate modified bits for each distinct cache line in the next upper level. When a line is a target of a flush or copyback, if any portion of the line is dirty in the upper caches, the valid dirty data at this level needs to be buffered and flush or copyback request(s) which target the portions of the line that are stale sent to the next upper cache. Eventually all the flush or copyback replies from the next upper cache will be received and the complete response can be constructed and sent to the next lower level.

Direct-mapped Simplifications

The protocol in Table C.5 sends an explicit flush or invalidate message for lines replaced by a reply, as it does not rely on the same cache line being replaced at all levels of the cache hierarchy. Under certain conditions, this replacement policy can be guaranteed due to a restrictive mapping of caches to sets. This is true for a hierarchy consisting of only direct-mapped caches which all have the same line size. As an additional constraint, the cache hierarchy must consist solely of unified caches or data caches (i.e. a hierarchy which consists of unified caches in the lower levels and separate data and instruction caches in the upper levels violates this replacement policy).

Under these constraints, replies do not need to explicitly invalidate or flush the lines they are replacing, as the traversal of the reply up the cache hierarchy will implicitly perform that flush or invalidation.

C.3 Summary

A coherent, blocking cache hierarchy has several basic needs. First, all caches in the hierarchy must be dual-ported (either via time multiplexing or via a true dual-port cache design) to be able to handle requests from both the processor and external interface. Second, the use of multiple writeback caches is most efficiently handled through four cache states, two of which track whether an exclusive copy of data is up-to-date or stale. Finally, a coherence protocol needs to be developed which can handle all the intra-hierarchy races. We have developed such a protocol in this appendix.

Appendix D

Buffer Deadlock in Multilevel Cache Hierarchies

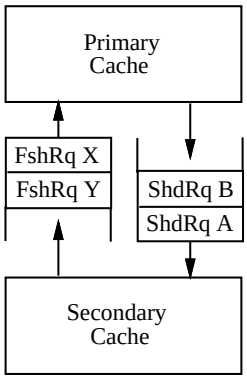


Figure D.1: Buffer deadlock example.

This appendix discusses how to address deadlock which arises due to the finite buffering between writeback caches. An example of this deadlock occurs when there are a pair of buffers between two writeback caches (one incoming, one outgoing), they are completely full, and in order to process the messages at the head of each queue the cache must generate a response for the other queue. This scenario is shown in Figure D.1.

We can look at the deadlock problem more formally by generating a resource graph [Hol72]. A resource graph is a directed graph with two types of nodes: processes (shown as ovals) and resources (shown as rectangles). An arc from a resource to a process signifies that the resource has been granted to the process. An arc from a process to a resource means that the process

is blocked waiting for that resource. A cycle in the graph implies that a deadlock has occurred involving the processes and resources in the cycle. We can see the cycle in the resource graph for our buffer deadlock example in Figure D.2. Each buffer entry is represented as a resource, and each message as a process. Caches are not modeled as resources, since they can always accept a message as long as there is buffer space for the response messages generated.

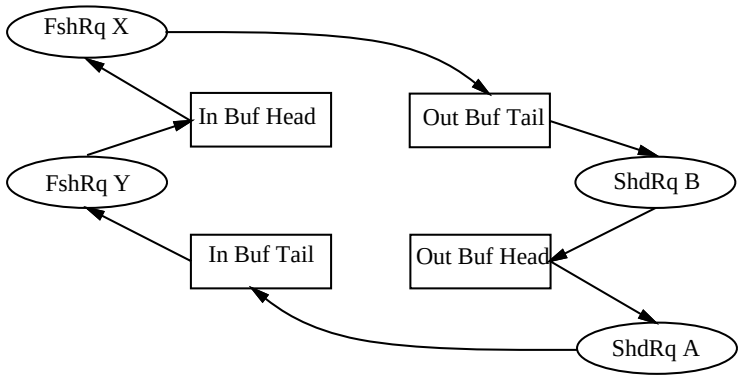


Figure D.2: Resource graph for buffer deadlock example.

Within a cache hierarchy, messages cannot randomly ask for any buffer resource. The buffers maintain FIFO ordering, so for all locations within a buffer except the head, a message can only be blocked for the next FIFO entry. At the head of the FIFO, servicing a message by the cache can require entry into zero, one or two FIFOs as specified by the protocol given in Chapter 5 and Appendix C.

We show the possible inter-FIFO transitions allowed by the protocol in the right half of Figure D.3. The FIFOs are shown as boxes. Note that incoming messages sent to the writethrough caches never generate an outgoing request under the coherence protocol, so messages to the writethrough caches cannot become involved in a deadlock cycle. However, incoming messages to the writeback caches can generate outgoing messages and vice versa. This leads to a possibility for deadlock cycles to arise when the buffers fill. For example, a deadlock cycle could arise between IB *N+2* and OB *N+2*, or between IB *N+3*, IB *N+2*, OB *N+2* and OB *N+3*.

We will now explore a number of solutions to solve the deadlock problem. These solutions are not intended to be exhaustive; the point is to show that buffer deadlock can be handled through a variety of techniques. Deadlock solutions come in three forms: prevention, avoidance, and detection and removal [Tan86]. Prevention schemes provide enough resources to ensure that a circular dependency cannot arise. Avoidance schemes carefully allocate resources in a manner

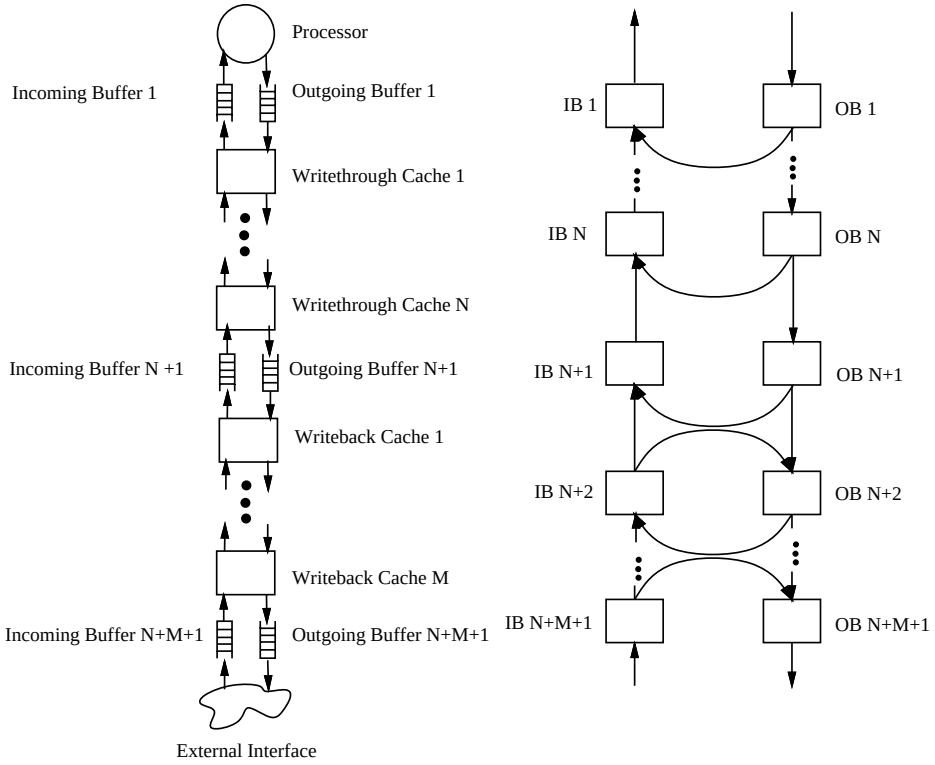


Figure D.3: Permitted message transitions between buffers.

that avoids any circular dependencies which may lead to deadlock. Detection and removal simply ignores the deadlock problem until a deadlock situation arises. At this point, some method of breaking the deadlock is invoked.

D.1 Deadlock Prevention

The simplest deadlock prevention solution is to use only a single level of writeback cache in the cache hierarchy. This removes the possibility of deadlock because the protocol and FIFO ordering of messages guarantees that no cycle involving the buffers can arise as shown in Figure D.4. However, assuming multiple levels of writeback caches are needed for performance reasons, a more general deadlock prevention scheme needs to be developed.

Deadlock can be also be prevented by limiting the number of requests that are injected into the hierarchy, both from the processor and from the external interface. By limiting the number of requests that can be in transit within the hierarchy and providing buffers large enough to handle

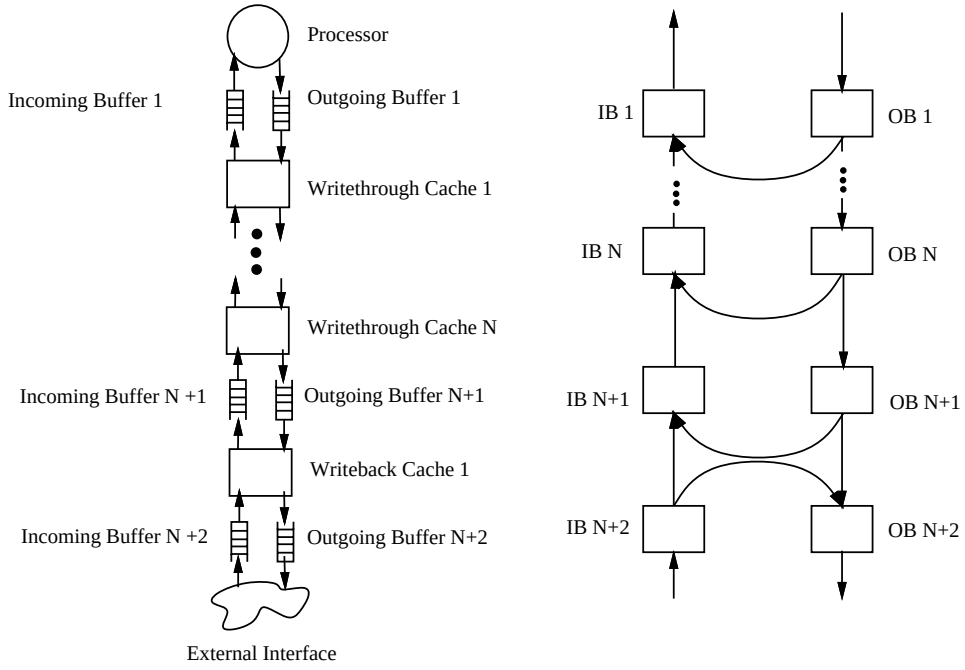


Figure D.4: Permitted message transitions between buffers for cache hierarchy with a single level of writeback caches.

all these messages, deadlock can be prevented. For example, if the processor can only have *m* outstanding requests, and only *n* incoming requests from the external interface are allowed in the cache hierarchy, a maximum of *2m+n* messages can be in transit within the cache hierarchy.[1] Under these constraints, deadlock can be prevented by providing buffers between the caches which can hold $\lceil \frac{2m+n}{2} \rceil + 1$ messages (based on our requirements being most restrictive between two adjacent writeback caches, a pair of incoming and outgoing buffers must be able to hold all $2m + n$ messages plus one extra space per buffer space to allow a waiting message to be processed).

While this solution prevents deadlock, it can reduce performance since limits are placed on the number of requests which can be in transit within the hierarchy. To make the impact on performance minimal the number of requests allowed within the hierarchy can be made reasonably large, however this will also result in fairly large buffers between caches. We will now discuss a deadlock avoidance technique which will allow a large number of requests to be outstanding,

[1]Each external request can only generate one reply, making the maximum messages due to each external request one. Each processor request can generate a response which may also replace a single dirty cache line, making the maximum messages due to each processor request two.

but can do so with more modest buffer requirements.

D.2 Deadlock Avoidance

A deadlock avoidance solution can be constructed using the Banker's algorithm [Tan86]. The Banker's algorithm regulates entry into the queues to avoid deadlock; a message can only enter a queue when it is guaranteed that the resources it will need are available. We show this pictorially in Figure D.5. For the shared request for B to be processed as a miss in the primary cache, not only must there be space in Outgoing Buffer 2, but there must also be space for the secondary cache to place a reply to the request from B in Incoming Buffer 2 in case of a hit or to place an request in Outgoing Buffer 3 for a miss. By only processing messages that meet these guarantees, deadlock can be avoided.

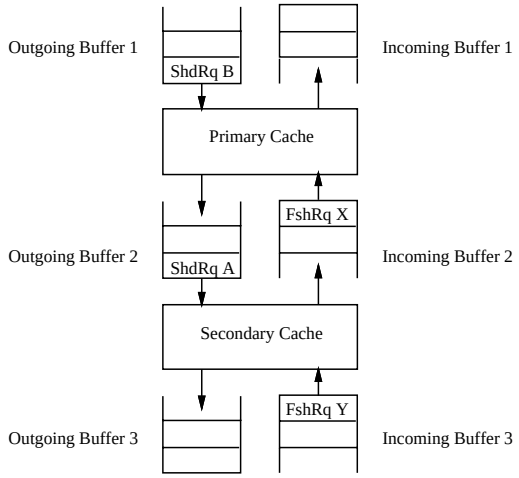


Figure D.5: Example for deadlock avoidance solution.

However, there is a complication with this scheme, in that the secondary cache also needs to look at the status of Incoming Buffer 2 and Outgoing Buffer 2 to determine if it can process incoming flush and copyback messages. This requires the primary and secondary caches to communicate — and avoiding this communication was the one of the driving forces behind providing the buffering.

Luckily, the basics of the Banker's algorithm can be implemented by ensuring that there is always enough buffering for requests making the "loop-around". For example, if all requests and

replies occupy a single buffer entry, if half of the buffer space is reserved for requests looping-around, deadlock cannot arise. Under this scheme, an incoming buffer N would signal full to messages from incoming buffer $(N+1)$ when only half-full. The other half of the buffer is reserved for messages from outgoing buffer N making the loop-around. Outgoing buffer N also signals full to outgoing buffer $(N-1)$ when only half-full. With this scheme, deadlock can be avoided while still using small buffers (the minimum size being two entries).

Of course if the messages have differing sizes, the buffers would need to take the message size into account when determining how many incoming requests it can allow. For example, if requests occupy one buffer entry and replies occupy three buffer entries, an incoming buffer N would need to signal full to incoming buffer $(N+1)$ when only one-quarter full.

This scheme has the advantage that is very simple to implement and retains the decoupling of caches. The cost of this simplicity is a delay in processing messages due to the buffers signaling full when according to the complete resource graph the buffer could accept the message without possibility of deadlock. With buffers not much larger than the minimum allowable size of two, the occurrence of this false full signal should be infrequent.

D.3 Deadlock Detection and Removal

The final general method used to solve deadlock is deadlock detection and removal. This is done by detecting and breaking some of the cycles. For example, a timeout could be generated if the buffers have not progressed for a certain number of cycles. This timeout would cause some entity to intercede and somehow break the deadlock (such as turning all requests into negative acknowledge replies). A timeout mechanism is used in both the LimitLESS [KCA92] and DASH [LLG+90] protocols to break deadlocks in the network buffers. In such instances, it may be possible to integrate the cache buffer deadlock algorithm with the existing network deadlock handling. However, if the cache buffers are relatively small, the deadlock situation may arise frequently enough that the cost of the deadlock detection and removal scheme may be too expensive.

D.4 Summary

We have outlined a number of techniques that can be used to address the problem of buffer deadlock. As mentioned earlier, the techniques listed are not exhaustive, but instead show that the

deadlock problem can be addressed by a variety of methods. The selection of a specific deadlock handling technique will depend on the probability of deadlock occurring and the performance/cost of the various deadlock handling mechanisms. Of course, the lowest cost deadlock solution is to only support a single level of writeback caches, however the performance of this solution may not be the most optimal.

Appendix E

Processor Status Word

In this appendix, we examine the processor status word (PSW) replication requirements for a multiple-context processor. Because each architecture (and potentially implementation) will have a different PSW, in order to make our discussion concrete we have selected the MIPS R4000, as the processor we simulated was partially based on the R4000. The insights gained through this examination should be applicable to other architectures although the specifics may not.

The R4000 has a number of status registers, as shown in Table E.1. More detail on the status registers can be found in [Hei93]. While the R4000 has a large number of status registers, most of them contain state which is not process-specific, but rather provided for use by the kernel or for control of the processor and its interface. The only registers which have process-specific state are given in Table E.2.[1] The WatchHi and WatchLo registers may also need to be replicated to be able to watch a unique addresses for each context — we have assumed that providing the ability to watch a single address for all contexts is sufficient. We now examine how to replicate these process-specific registers in detail, starting with EntryHi and EntryLo. Replication of these registers is the same for both the blocked and interleaved schemes, with the exception of the FP Control/Status register.

E.1 EntryHi and EntryLo

The EntryHi and EntryLo registers are used as an access port to the TLB. They have the format shown in Figure E.1. The MIPS R4000 TLB maps two consecutive pages with a single TLB

[1]The R4000 also has separate integer multiply/divide registers, MHI and MLO. Since our simulated architecture based its floating-point on the DEC 21064, which does not use separate multiply/divide registers, we do not discuss MHI and MLO in this appendix.

Table E.1: MIPS R4000 processor status word registers.

Register	Explanation
Memory Management	
Page Mask	Mask for variable size pages.
EntryHi, EntryLo0, EntryLo1	Registers used to read, write, or probe the TLB.
Index	Register used to control the TLB entry read, written, or probed.
Random	Register used to control the random TLB entry written.
Wired	Register used to control the random replacement of TLB entries.
Exception Handling	
Context, XContext	Pointer to PTE base coupled with bad page number.
BadVAddr	Displays most recent address that failed to have a valid translation.
Count	Free-running timer.
Compare	Interrupts processor when value equals count (timer service).
Status	Various processor status and control.
Cause	Holds cause of most recent exception.
PRId	Processor implementation and revision level.
Config	Processor configuration information.
WatchHi, WatchLo	Provides memory address tracing capability.
ECC, CacheErr, TagLo, TagHi	ECC control and cache diagnostic registers.
ErrorEPC	Exception PC used solely for parity and ECC errors.
Floating-point Control	
FP Control/Status	Floating-point rounding-mode and exception cause, enable, and flags.
FP Implementation/Revision	Floating-point implementation and revision level.

Table E.2: MIPS R4000 process-specific portion of the processor status word.

Register	Explanation
EntryHi, EntryLo0, EntryLo1	Registers used to read, write, or probe the TLB.
Status	Various processor status and control.
FP Control/Status	Floating-point rounding-mode and exception cause, enable, and flags.
LL Bit	Status bit for the Load Linked/Store Conditional instructions.
LL Addr	Address of most recent Load Linked instruction.

entry, requiring two EntryLo registers per EntryHi register. The fields in EntryHi and the two EntryLo registers are:

R Region (user, supervisor, kernel).

VPN2 Virtual page number divided by 2 (maps to two pages).

ASID Address space id (process identifier allowing multiple processes to share the TLB).

PFN Page frame number.

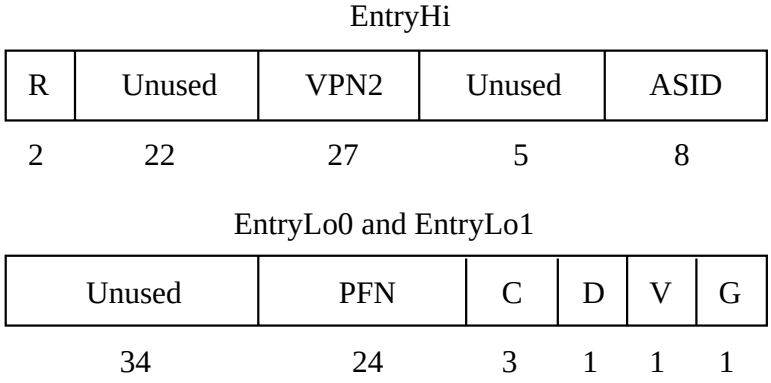


Figure E.1: EntryHi and EntryLo registers.

C Cache coherence algorithm (e.g. non-coherent, invalidate, update).

D Dirty bit.

V Valid bit.

G Global bit (if set, the ASID is ignored).

Most of the functionality provided by EntryHi and EntryLo is not affected by the addition of multiple contexts to the processor. However, the ASID and G fields are of interest for multiple-context processors. The ASID field allows multiple processes to share the processor without having to flush the TLB at each context switch. This is done by assigning a unique ASID to each process. Thus, the TLB needs to be flushed only when all the ASIDs on the processor have been allocated. The TLB also provides a global bit per TLB entry to override the ASIDs. If the global bit is set for a mapping, the ASID is ignored in determining the TLB entry validity. The global bit is useful for operating system pages.

Thus, the current MIPS TLB allows for a process to have its own private mappings, and it also provides for global mappings for use by all processes. However, the TLB lacks the capability to efficiently provide a mapping for a set of processes on the same processor. Assigning the same ASID to all processes in the set does not allow the processes to have any nonshared pages for their stacks and private data. Therefore, the processes must have separate ASIDs, and the EntryHi register will need to provide a separate ASID per context.

In addition, while our simulation studies showed only MP3D to benefit from providing the TLB with the ability to map pages shared between the contexts with a single TLB entry, applications with larger data sets that are placing much more pressure on the TLB will desire this ability. Without further modifications, the MIPS TLB cannot support pages shared between contexts.

Table E.3: Modes for each TLB entry.

Global Bit	Private Bit	Explanation
1	1	*Illegal*
1	0	Valid for all processes
0	1	Valid for processes with same ASID and GID
0	0	Valid for processes with same ASID

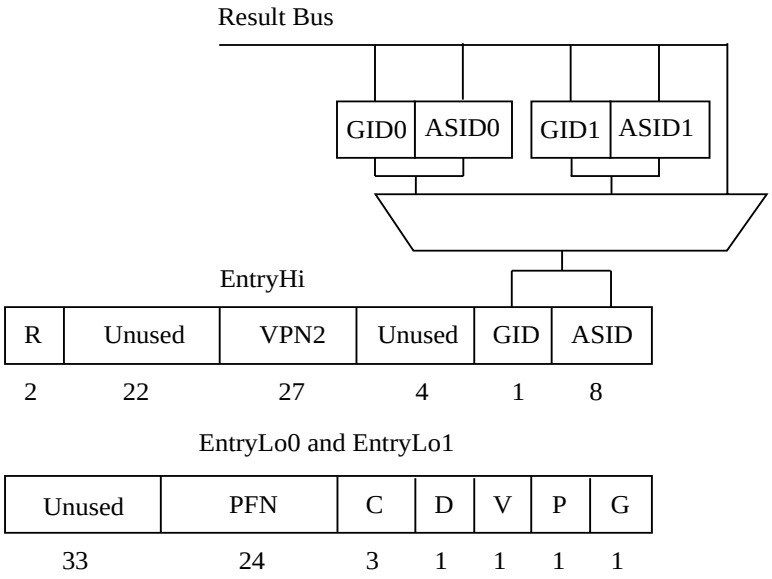


Figure E.2: EntryHi and EntryLo registers, modified for two contexts.

This support can be implemented rather easily by supplementing the ASID by a Group Identifier (GID) of size $log_2(number\ of\ contexts)$ bits and providing an additional *private bit* per TLB entry. The complete process identifier now consists of the GID concatenated to the original ASID. The new operation of the TLB is as follows. If the private bit is set, both the GID and ASID are used in determining TLB hits, providing per-process page mappings. If the private bit and global bit are both cleared, only the ASID is used when determining TLB hits, allowing process set mappings. Finally, if the global bit is set, both the ASID and GID are ignored in determining TLB hits. This operation is summarized in Table E.3. Of course, the operating system must assign a process set the same ASID for the TLB to function effectively. Shared pages within the process set would have both their global and private bits cleared, while private pages would have their global bit cleared and private bit set.

Since the ASID is contained in the EntryHi register, the multiple-context processor will need this portion of the register to be replicated to hold an ASID for each context. These extra ASIDs will need to be transferred on each context switch. Figure E.2 shows a two-context modified EntryHi register. On a context switch, the appropriate ASID is written into the EntryHi ASID. When initializing the ASID for a new processes, the operating system should be executing as the context it is loading. Writes to the EntryHi register come from the result bus and update the appropriate context ASID register in addition to EntryHi.

E.2 Status

Figure E.3: Status register.

The status register is shown if Figure E.3. This register contains various processor control information:

CU Coprocessor usability
RP Enable reduced-power operation.
FR Select 16 or 32 floating-point registers.
RE Reverse endian in user-mode.
DS Diagnostic status.
IM Interrupt mask.
KX Select 64- or 32-bit addressing in kernel mode.
SX Select 64- or 32-bit addressing in supervisor mode.
UX Select 64- or 32-bit addressing in user mode.
KSU Mode (kernel, supervisor, user).
ERL Error level.
EXL Exception level.
IE Interrupt enable.

Several of these fields (RP, FR, DS) provide processor control and therefore do not need to be replicated for a multiple-context processor. Other fields are the same for all user processes or provided for the kernel or supervisor (CU, IM, KX, SX, KSU, ERL, EXL, IE). Finally,

the remaining fields (RE, UX) can vary for different user programs. We could replicate these fields, however, it is much simpler to require that the operating system always schedule a set of processes that have the same addressing mode and endianness. Of course, for processes of a parallel application, these two application characteristics will be the same.

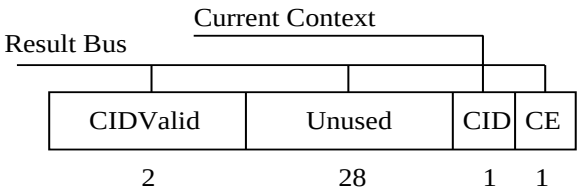


Figure E.4: Context status register (for two contexts).

While no fields from the status register needed to be replicated for the multiple-context processor, some additional control fields are required. Since there is no space in the existing status register, we add a single separate context status register to the processor. A Context Status register which supports two contexts is shown in Figure E.4. The context-switch enable (CE) bit is provided to allow context switching to be inhibited for single-context operation and upon entry into the kernel. A CID field is supported to identify the currently executing context. Finally, a CIDValid field is provided which signifies the contexts loaded with valid processes on the processor. This field can either be a bit vector, as shown, or simply contain the ID of the highest valid context (requiring the operating system to pack the contexts).

E.3 Floating-point Control/Status

The floating-point control/status register (FPCSR) controls floating-point operation. The FPCSR is unique in that it is user-visible and therefore needs to handle both reads and writes from a user process. The R4000 architecture specifies that a FPCSR read will interlock until all previous floating-point operations have completed. FPCSR writes will only work properly if no floating-point instructions are in the pipeline, so all FPCSR writes are preceded by FPCSR reads to guarantee that the floating-point pipeline is empty.

The format of the FPCSR register is shown in Figure E.5. The fields are:

FS Flush denormalized results to zero.

C Result of last floating point compare.

Cause Cause of last floating-point exception.

Enables Enable of floating-point exceptions.

Flags Exception status flags.

RM Rounding mode.

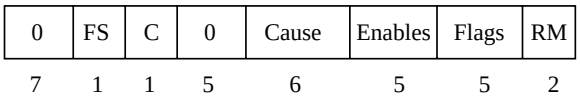


Figure E.5: Floating-point control/status register.

Because it is user-visible, replication of the FPCSR differs between the blocked and interleaved schemes. Replicating the FPCSR for the blocked multiple-context processor is fairly straightforward. The C (compare) field must be replicated since it is part of the process state. If this field was not saved, it could be generated and tested on different sides of a context switch, leading to incorrect execution. All of the other fields except the Cause field (which is provided solely for the exception handler and is never visible to the user) must be replicated to properly implement the IEEE 754 standard. Thus the modified floating-point control/status register shown in Figure E.6 results.

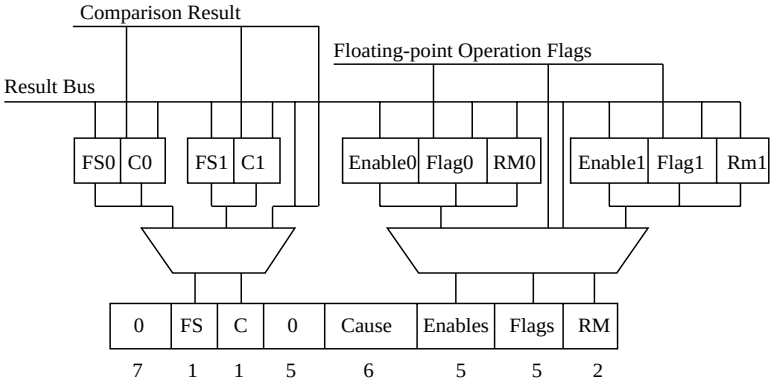


Figure E.6: Floating-point control/status register, modified for two contexts.

Replicating the FPCSR for the interleaved scheme is slightly more complicated. Under the interleaved scheme these reads and writes can be interspersed with FPCSR updates resulting from floating-point operations in progress. By interlocking on all outstanding floating-point operations, regardless of which context initiated them, the FPCSR read can access the appropriate context state after the interlock is resolved. The MIPS architecture specifies that a FPCSR write must only occur when there are no outstanding floating-point operations. Under the interleaved scheme

this requirement can be relaxed to specify that a FPCSR write from a given context must only occur when there are no outstanding floating-point operations from that context.

This looser requirement can be ensured by the having the FPCSR read interlock until floating-point operations from all contexts have completed. When a context immediately precedes a FPCSR write by a FPCSR read, this does not guarantee that no floating-point operations are in progress from *any* context, but will still guarantee that the context doing the read does not have any outstanding floating-point operations. By updating only the replicated fields of the FPCSR register, the FPCSR write will not disturb the operation of the FPCSR register for any of the floating-point operations from other contexts which are currently in operation. However, as shown in Figure E.7, being able to update the replicated FPCSR state from both a FPCSR write and a normal operation requires a separate datapath for the FPCSR write. As an alternative to providing this additional datapath and its control, the architecture could provide interlocks for both FPCSR reads and writes.

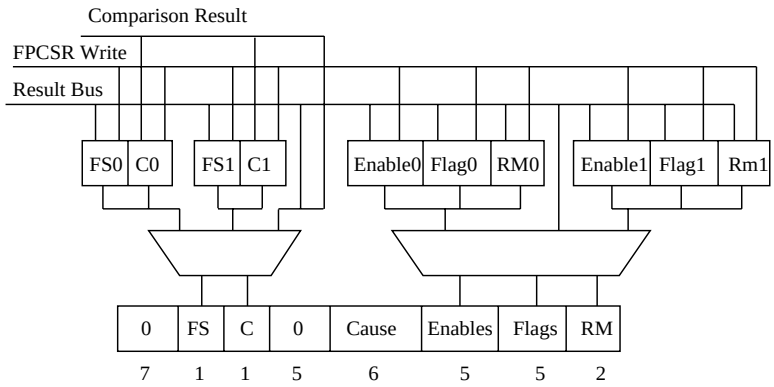


Figure E.7: Interleaved floating-point control/status register.

E.4 LL Bit and LL Addr

The LL Bit is the user-visible state for the load linked/store conditional instructions. The load linked/store conditional pair are used to perform atomic operations on the R4000. The operation of the load linked/store conditional pair is as follows. The load linked instruction performs a load into a register and sets the LL Bit. This LL Bit is cleared if a coherence operation to the memory location specified by the load linked operation or an exception return occurred between the load linked and store conditional. The store conditional instruction stores to the address if

the LL Bit is still set. Regardless of whether the store conditional succeeds, the status of the LL Bit is placed in a result register. For our discussion, we will assume that the LL Addr register[2], a user-hidden register which maintains the address of the most recent load linked, is used by the hardware to track coherence operations.

With multiple contexts, we will need to provide a copy of both the LL Bit and LL Addr register per context. Both of these registers need to be replicated because more than one context can simultaneously be operating in the critical section between a load linked and its corresponding store conditional. Without replication, the processor could get into a deadlock situation where no context can complete its store conditional before another context issued a load linked.

E.5 DEC 21064 PSW

We have just examined the processor status word for the MIPS R4000, and found its replication requirements to be reasonable. To see if this holds for at least one other architecture, we also looked at the PSW for another second generation RISC processor, the DEC 21064. The process-specific portion of the processor status word for the 21064 is surprisingly similar to the R4000. The 21064 has a larger number of processor control registers, however, the number of actual process-specific registers is still small. The first of these registers is the instruction cache control/status (ICCSR) register. This register holds a large amount of processor status and some process-specific status. This process-specific status includes the address space number (ASN), the type of branch prediction (not taken, based on displacement sign, BTB), and some performance monitoring control. It is comparable to the Status register of the R4000, but also includes the ASN, which resides in the EntryHi register on the R4000. The 21064 also has an EXC_SUM register that is used to look at the exception bits associated with the register file (used for arithmetic overflows and other imprecise interrupts). This register returns the exception bit of the registers in sequence with each read. The 21064 also has a lock_flag and locked_physical_address for its load locked/store conditional instruction pair which function the same as the LL Bit and LL Addr of the R4000. Finally, the 21064 also has a floating point control register which contains the rounding mode and exception flags.

Only the EXC_SUM register does not have a corresponding R4000 PSW register. The multiple-context modifications for all the other registers would be similar to the R4000. The

[2]There is a LLAddr register provided for diagnostic purposes on the R4000. We are not referring to this register, but instead to an implementation-specific register which allows for efficient tracking of coherence operations to the most recent load linked address.

EXC_SUM register will simply need to be modified to return the exception bits of the register file of the current context. In addition, the 21064 also only provides two TLB modes, private and global, so the TLB would need to be modified to handle process sets.

E.6 Summary

In this appendix we have examined the PSW replication requirements for two different architectures, the MIPS R4000 and the DEC Alpha. While one cannot draw final conclusions about PSW size from two examples, the similarity of the PSWs is encouraging. For at least two modern architectures, the number of state registers that need to be replicated turned out to be rather small.

Bibliography

[AAC+91] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Workshop on Multithreaded Computers, Supercomputing '91*, November 1991.

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *1990 International Conference on Supercomputing*, pages 1–6, June 1990.

[ACD+91] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Workshop on Multithreaded Computers, Supercomputing '91*, November 1991.

[Aga92] Anant Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[AH90] Sarita Adve and Mark Hill. Weak ordering — A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[ALKK90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[BCKK88] M. Berry, D. Chen, P. Koss, and D. Kuck. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD Report No. 827, Center for Supercomputing Research and Development, November 1988.

[BFKR92] Henry Burkhardt III, Steven Frank, Brue Knobe, and James Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.

[BJS88] Forest Baskett, Tom Jermoluk, and Doug Solomon. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. In *Proceedings of COMPCON Spring '88: Thirty-third IEEE Computer Society International Conference*, pages 468–471, February 1988.

[BR92] Bob Boothe and Abhiram Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 214–223, May 1992.

[BW88] Jean-Loup Baer and Wen-Hann Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, June 1988.

[Car92] Alexander Carlton, editor. *SPEC Newsletter*, volume 4. SPEC, September 1992.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.

[CF78] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[CGB91] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Paradigm: A highly scalable shared-memory multicomputer architecture. *IEEE Computer*, 24(2):33–46, February 1991.

[CGL92] David E. Culler, Michial Gunter, and James C. Lee. Analysis of multithreaded microprocessors under multiprogramming. Technical Report UCB/CSD 92/687, University of California, Berkeley, May 1992.

[CK92] Andrew A. Chien and Jae H. Kim. Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 268–277, May 1992.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[CM81] K. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communication of the ACM*, 24(11):198–206, April 1981.

[Cra93] Cray Research, Incorporated. *Cray T3D Technical Summary*, October 1993.

[Cyp90] Cypress Semiconductor Corporation. *SPARC Risc User's Guide*, 2nd edition, 1990.

[DBCÖ92] Michel Dubois, Luiz Barroso, Yung-Syau Chen, and Koray Öner. Scalability problems in multiprocessors with private caches. In *Proceedings of Parallel Architecture and Languages Europe '92*, pages 211–230, June 1992.

[DCC+87] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a message-driven processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 189–196, June 1987.

[DGH91] Helen Davis, Steven R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, pages 99–107, August 1991.

[Dig92a] Digital Equipment Corporation. *Alpha Architecture Handbook*, preliminary edition, 1992.

[Dig92b] Digital Equipment Corporation. *DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet*, 1992.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[DT91] George E. Daddis Jr. and H. C. Torng. The concurrent execution of multiple instruction streams on superscalar processors. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 76–83, 1991.

[EK88] Susan J. Eggers and Randy H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–383, May 1988.

[EK89] Susan J. Eggers and Randy H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 2–15, May 1989.

[FP91] Matthew K. Farrens and Andrew R. Pleszkun. Strategies for achieving improved processor throughput. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 362–369, May 1991.

[GD90] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[GGH92] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proceeding of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, May 1992.

[GH93] Steven R. Goldschmidt and John Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–157, May 1993.

[Gha92] Kourosh Gharachorloo. Personal communication, 1992.

[GHG+91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceeding of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[Goo91] James R. Goodman. Cache consistency and sequential consistency. Technical Report Computer Sciences Technical Report #1006, University of Wisconsin, Madison, February 1991.

[GTS91] Anoop Gupta, Andrew Tucker, and Luis Stevens. Making effective use of shared-memory multiprocessors: The process control approach. Technical Report CSL-TR-91-475, Stanford University, July 1991.

[GW94] Mike Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, volume I, pages 134–143, 1994.

[Hag92] Erik Hagersten. *Toward Scalable Cache Only Memory Architectures*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, October 1992.

[Hei93] Joe Heinrich. *MIPS R4000 User's Manual*. Prentice-Hall, 1993.

[HF88] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.

[HKN+92] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[Hol72] R. C. Holt. Some deadlock properties of computer systems. In *Computing Surveys*, volume 4, pages 179–196, September 1972.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Scalable coherent interface. *IEEE Computer*, 23(6):74–77, June 1990.

[JMY92] William Jaffe, Bob Miller, and Jeff Yetter. A 200 MFLOP precision architecture processor. In *Hot Chips IV Symposium Record*, pages 1.2.1–1.2.13, August 1992.

[Joh89] William M. Johnson. *Super-Scalar Processor Design*. PhD thesis, Stanford University, June 1989.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[Jou93] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201, May 1993.

[KCA91] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency tolerance through multithreading in large-scale multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 91–101, April 1991.

[KCA92] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the window of vulnerability in multiphase memory transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–284, October 1992.

[KD79] William J. Kaminsky and Edward S. Davidson. Developing a multiple-instruction-stream single-chip processor. *IEEE Computer*, pages 66–76, December 1979.

[KD92] Stephen W. Keckler and William J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213, May 1992.

[KEW+85] R. Katz, S. Eggers, D. A. Wood, C. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, 1985.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 81–87, 1981.

[Kro92] David Kroft. Personal communication, August 1992.

[Kru91] Steve Krueger. SuperSPARC: A fully integrated superscalar processor. In *Microprocessor Forum Proceedings*, pages 8–1–8–7, November 1991.

[KS88] James T. Kuehn and Burton J. Smith. The Horizon supercomputing system: Architecture and software. In *Proceedings of Supercomputing '88*, pages 28–34, November 1988.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[Len92] Daniel Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford University, Stanford, California, February 1992.

[LGH92] James Laudon, Anoop Gupta, and Mark Horowitz. Architectural and implementation tradeoffs in the design of multiple-context processors. Technical Report CSL-TR-92-523, Stanford University, May 1992.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LLJ+93] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[LOB+87] Ewing Lusk, Ross Overbeek, James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[LRW91] Monica Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[LS84] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[LW92] Monica Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[McF93] Grant McFarland. A register file for a multiple-context processor. EE 391 Project Class Final Report, November 1993.

[MLG92] Todd Mowry, Monica Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[Mot91] Motorola, Inc. *MC88110 Second Generation RISC Microprocessor User's Manual*, 1991.

[Mow94] Todd Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Stanford, California, March 1994.

[ND91] Peter R. Nuth and William J. Dally. A mechanism for efficient context switching. In *Proceedings of the 1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 301–304, 1991.

[Omo91] Amos R. Omondi. Design of a high performance instruction pipeline. *Computer Systems Science and Engineering*, 6(1):13–29, January 1991.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

[PW91] R. Guru Prasadh and Chuan-lin Wu. A benchmark evaluation of a multi-threaded RISC processor architecture. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 84–91, 1991.

[RG90] Edward E. Rothberg and Anoop Gupta. Techniques for improving the performance of sparse factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, November 1990.

[Ros88] Jonathan Rose. Locusroute: A parallel global router for standard cells. In *Proceedings of the 25th Design Automation Conference*, pages 189–195, June 1988.

[SB91] Joy Shetler and Steven E. Butner. Multiple stream execution on the DART processor. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 92–96, 1991.

[SBCvE90] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, July 1990.

[SD87] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, June 1987.

[SD88] Christoph Scheurich and Michel Dubois. Concurrent miss resolution in multiprocessor caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume I, pages 118–125, August 1988.

[SD91] Christoph Scheurich and Michel Dubois. Lockup-free caches in high-performance multiprocessors. *Journal of Parallel and Distributed Computing*, 11(1):25–36, January 1991.

[SDL91] Per Stenstrom, Fredrick Dahlgren, and Lars Lundberg. A lockup-free multiprocessor cache design. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 246–250, 1991.

[SH92] Jaswinder Pal Singh and John Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experience, results, and implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[SHG92] Jaswinder Pal Singh, John Hennessy, and Anoop Gupta. Implications of hierarchical N-body methods for multiprocessor architecture. Technical Report CSL-TR-92-505, Stanford University, 1992.

[Sim92] Richard Simoni. *Cache Coherence Directories for Scalable Multiprocessors*. PhD thesis, Stanford University, Stanford, California, October 1992.

[Smi78] Burton J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[Smi81] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

[Smi92] Michael David Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, Stanford, California, November 1992.

[SMM88] Michael Sporer, Franklin H. Moss, and Craig J. Mathias. An introduction to the architecture of the Stellar graphics supercomputer. In *Proceedings of COMPCON Spring '88: Thirty-third IEEE Computer Society International Conference*, pages 464–467, February 1988.

[Sou92] Lawrence Peter Soule. *Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms*. PhD thesis, Stanford University, Stanford, California, June 1992.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[Tan76] C. K. Tang. Cache design in the tightly coupled multiprocessor system. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

[Tan86] Andrew S. Tanenbaum. *Operating Systems*. Prentice-Hall, 1986.

[Thi92] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, January 1992.

[Tho64] James E. Thornton. Parallel operation in the Control Data 6600. In *AFIPS Conference Proceedings, Fall Joint Computer Conference*, volume 26, pages 33–40, 1964.

[Tho70] James E. Thornton, editor. *Design of a Computer — The Control Data 6600*. Scott, Foresman, and Co., 1970.

[TML+82] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A personal computer. In Daniel P. Sieworek, C. Gordon Bell, and Allen Newell, editors, *Computer Structures: Principles and Examples*, pages 549–572. McGraw-Hill, 1982.

[Tor92] Josep Torrellas. *Multiprocessor Cache Memory Performance: Characterization and Optimization*. PhD thesis, Stanford University, Stanford, California, August 1992.

[TS87] Charles P. Thacker and Lawrence C. Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[TS88] Mark R. Thistle and Burton J. Smith. A processor architecture for Horizon. In *Proceedings of Supercomputing '88*, pages 35–41, November 1988.

[TS90] Shreekant S. Thakkar and Mark Sweiger. Performance of an OLTP application on Symmetry multiprocessor system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 228–238, May 1990.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.

[Web93] Wolf-Dietrich Weber. *Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors*. PhD thesis, Stanford University, Stanford, California, January 1993.

[WG89a] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[WG89b] Wolf-Dietrich Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, June 1989.