MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Technical Report No. 1545 June, 1995

Thread Scheduling Mechanisms for Multiple-Context Parallel Processors

James Alexander Stuart Fiske
stuart@ai.mit.edu

This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. This research was supported by the Advanced Research Projects Agency and monitored by the Air Force Electronic Systems Division under contract number F19628-92-C-0045.

Thread Scheduling Mechanisms for Multiple-Context Parallel Processors

by
James Alexander Stuart Fiske
Submitted to the
Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of
the requirements for the Degree of Doctor of Philosophy
in Electrical Engineering and Computer Science

Scheduling tasks to efficiently use the available processor resources is crucial to minimizing the runtime of applications on shared-memory parallel processors. One factor that contributes to poor processor utilization is the idle time caused by long latency operations, such as remote memory references or processor synchronization operations. One way of tolerating this latency is to use a processor with multiple hardware contexts that can rapidly switch to executing another thread of computation whenever a long latency operation occurs, thus increasing processor utilization by overlapping computation with communication. Although multiple contexts are effective for tolerating latency, this effectiveness can be limited by memory and network bandwidth, by cache interference effects among the multiple contexts, and by critical tasks sharing processor resources with less critical tasks. This thesis presents techniques that increase the effectiveness of multiple contexts by intelligently scheduling threads to make more efficient use of processor pipeline, bandwidth, and cache resources.

This thesis proposes *thread prioritization* as a fundamental mechanism for directing the thread schedule on a multiple-context processor. A priority is assigned to each thread either statically or dynamically and is used by the thread scheduler to decide which threads to load in the contexts, and to decide which context to switch to on a context switch. We develop a multiple-context model that integrates both cache and network effects, and shows how thread prioritization can both maintain high processor utilization, and limit increases in critical path runtime caused by multithreading. The model also shows that in order to be effective in bandwidth limited applications, thread prioritization must be extended to prioritize memory requests. We show how simple hardware can prioritize the running of threads in the multiple contexts, and the issuing of requests to both the local memory and the network.

Simulation experiments show how thread prioritization is used in a variety of applications. Thread prioritization can improve the performance of synchronization primitives by minimizing the number of processor cycles wasted in spinning and devoting more cycles to critical threads. Thread prioritization can be used in combination with other techniques to improve cache performance and minimize cache interference between different working sets in the cache. For applications that are critical path limited, thread prioritization can improve performance by allowing processor resources to be devoted preferentially to critical threads. These experimental results show that thread prioritization is a mechanism that can be used to implement a wide range of scheduling policies.

Thesis Supervisor: William J. Dally
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

Now that this thesis is finally coming to an end, there are many people that I would like to thank wholeheartedly for the help and support they provided me with along the way.

First of all, thanks to my research advisor Bill Dally for providing the knowledge, guidance, encouragement, and resources that were necessary to complete this work. Miraculously he succeeded in convincing me come back to MIT after my world travels, although he had the advantage that my memory had faded somewhat after a year and a half away. I would also like to thank my readers Charles Leiserson and Bill Weihl for their insightful and helpful suggestions on the various proposals and thesis drafts I sent their way.

The great people of Tech Square that I have had the privilege to meet and know greatly enriched my ordeal. The members of the CVA group all deserve my undying gratitude, as they patiently listened to my countless group meeting talks on various thread scheduling topics, and only occasionally nodded off. Rich Lethin, my friend and office mate of many years, gave me valuable support and council, read many of my thesis chapter drafts, set me up with all sorts of great job contacts, and lent me his car whenever some major system on it was about to collapse. Steve Keckler always had a good bad joke ready and waiting for me, and taught me the true meaning of the word dedication as he worked on his various sailboat projects. Kathy Knobe gave me good advice and encouragement, and always offered insightful comments about the various random scribbles I asked her to read. Peter and Julia Nuth were good friends, and provided amusing stories from the West Coast. John Keen gave many interesting talks on ephemeral logging and related topics, and was just generally a very good-hearted guy. Eric MacDonald gave me the chance to share an office with the star of the MIT cable network Star Trek phone in show. Duke Xanthopolous was always around late at night and was always willing to take a break and exchange some good natured verbal abuse. Larry Dennison provided a great example of how hard I should be working. Marco Fillo showed me how not to agonize over job offers. Michael Noakes gave me many lifts to hockey practice, and was full of insightful skepticism. Andrew Chang always kept things up and running, and was always interested in how things were going. Lisa Kozsdiy always found a way to fit me into Bill's Tuesday calendar. Debby Wallach deserves my thanks for always knowing how to do esoteric things on the computer system, and for clearing up lots of disk space when she finished her Master's thesis so I had space to do my work. Fred Chong, Ken MacKenzie, and Kirk Johnson regularly exchanged good hard hockey checks with me over the years. I shared regular talks and good laughs at Dilbert cartoons with John Kubiatowicz and Don Yeung at the coffee machine. Jim O'Toole played tennis with me regularly before we both got too busy finishing our theses. Ellen Spertus foisted off her wooden cow on me, one with a big painted smile, allowing me to drive my roommates crazy by keeping it in our living room. Thanks to the many other past and present inhabitants of Tech Square that have made this a truly colorful and fun place to be including Whay Lee, Waldemar Horwat, Nick Carter, Gino Maa, Silvina Hanono, Russ Tessier, Anne McCarthy, David Kranz, Tom Simon, and John Nguyen just to name a few.

Amazingly enough I did have some friends OUTSIDE of the Tech Square environment. Mark Wilkinson and Shawn Daly were always willing to go for a beer and tear up the town.

My roommates Marilyn Feldmeier and Taylor Galyean were great company, and great cooks. My roommate Mike Drumheller allowed me to put aside any notion that I might be going deaf by treating me to his impressive operatic voice periodically. Kati Flagg deserves my thanks for her faithful phone calls and inquiries about when I was going to graduate. I hope that she does not require treatment for shock when she hears I actually have. Brian Totty provided regular amusing E-mail, and I am hoping we will get together for a beer someday if he ever gets out of central Anatolia where I last left him. Carlos Noack, my long departed Colombian yogurthead friend from the early MIT days, continued to provide encouragement by regular correspondence.

A very big thank you goes to my special friend Adrienne, who has so patiently put up with me through the worst of the thesis and job search stress. She has provided a great incentive for me to lead some semblance of a normal life, and for me to do all sorts of fun things: biking 100 miles in one day down to Cape Cod, spending 8 hours in a car on trip to Washington D.C. with her mom, her mom's cat, and her Hungarian cousin, charging down black diamond mogul ski runs after not skiing for two years, and hiking up Mount Monadnock in gale force winds. Life just would not be much fun without her.

Finally, a special thanks to my entire family, especially my parents. Without their encouragement, support, and emphasis on education, I would probably never have come to MIT at all, let alone got through. I am thankful also that Dad will have to find a new way to greet me since "Are you almost done?" will no longer be appropriate. "Have you found a job?" has a nice ring to it I think — at least for a little while.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Problem

Scheduling tasks to efficiently use the available processor resources is crucial to minimizing the runtime of applications on shared-memory parallel processors. One factor that contributes to poor processor utilization is the idle time caused by long latency operations, such as remote memory references or processor synchronization operations. One way of tolerating this latency is to use a processor with multiple hardware contexts that can rapidly switch to executing another thread[1] of computation whenever a long latency operation occurs, thus increasing processor utilization by overlapping computation with communication. Although multiple contexts are effective for tolerating latency, this effectiveness can be limited by memory and network bandwidth, by cache interference effects among the multiple contexts, and by critical tasks sharing processor resources with less critical tasks. This thesis presents techniques that increase the effectiveness of multiple contexts by intelligently scheduling threads to make more efficient use of processor pipeline, bandwidth, and cache resources.

1.1.1 The Latency Tolerance Problem

Figure 1.1 shows a typical multiprocessor configuration consisting of a collection of processors connected to a high-performance network. Each processor has its own local cache and local memory. Operations that read or write remote data, or that synchronize with a remote processor, require the use of the network and give rise to long latencies. Even high performance, low-latency networks with low overhead network interfaces have round trip messages greater than 50 to 100 instruction cycles [76]. Processors that communicate

[1] In this thesis "task" and "thread" will be used interchangeably.

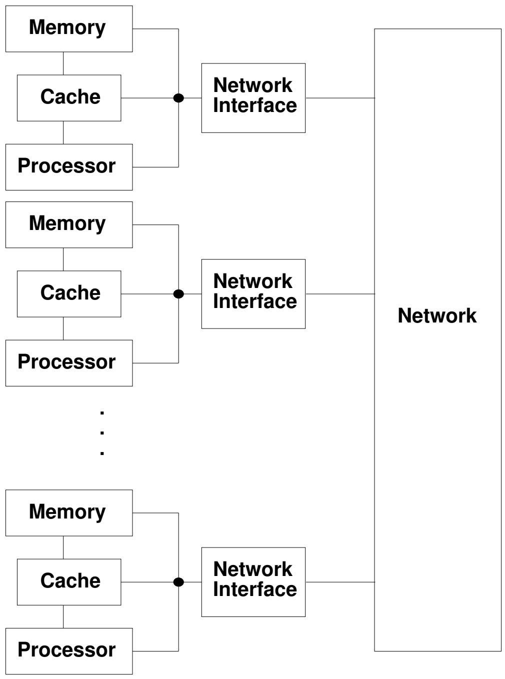


Figure 1.1: General multiprocessor system configuration.

often spend substantial amounts of time waiting for data, as shown in Figure 1.2a. With the increasing ratio of processor speed to DRAM speed [43], the latency associated with accesses that require no remote communication but miss in the cache is becoming increasingly important.

In order to efficiently use processor resources it is necessary to find ways of tolerating long latency data accesses and synchronization events.

1.1.2 Using Multiple Contexts to Tolerate Latency

A multiple-context processor as shown in Figure 1.3 multiplexes several threads over a processor pipeline in order to tolerate long communication and synchronization latencies. A straightforward implementation provides multiple register sets, including multiple instruction pointers, to allow the state of multiple threads to be loaded and ready to run at the same time. Each time the currently executing thread misses in the cache or fails a synchronization test, an opportunity exists to begin executing one of the other threads loaded in one of the other hardware contexts. This is shown in Figure 1.2b.

In a multiple-context processor threads are either *loaded* or *unloaded*. A thread is loaded if its register state is in one of the hardware contexts, and unloaded otherwise. Unloaded threads

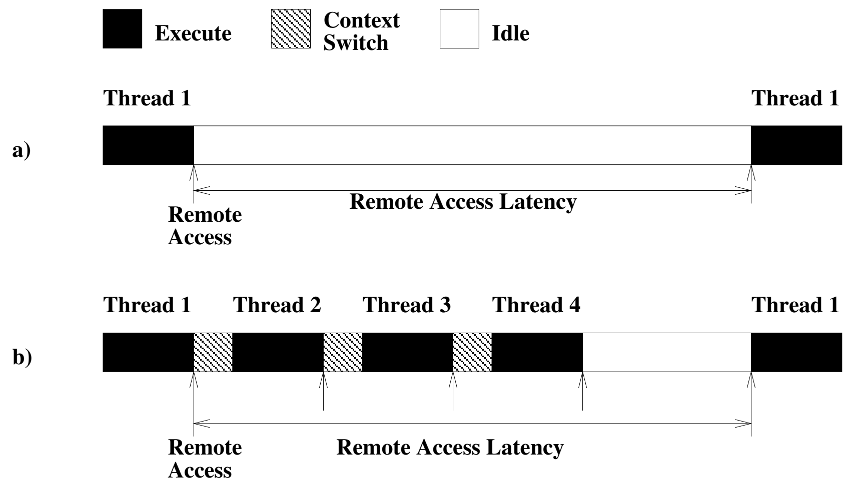


Figure 1.2: Effect of long latency operations. a. Without multithreading long idle periods are spent waiting for long latency operations to complete. b. With multithreading the processor can context switch and overlap computation with communication.

wait to be activated in a software scheduling queue. To allow a traditional RISC pipeline design, we assume a block multithreading model [5, 23], in which blocks of instructions are executed from each context in turn. At any given time, the processor is executing one of the loaded threads. On a *context switch*, the processor switches from executing one loaded thread, to executing another loaded thread, an operation that can typically be done in 0 to 20 cycles depending on the processor design. Keeping the context switch overhead low is necessary for multiple contexts to be effective in tolerating latency.

On a *thread swap* the processor switches a loaded thread with an unloaded thread from the software queue. A thread swap costs one to two orders of magnitude more than a context switch, because the processor must save and restore thread state, as well as modify the thread scheduling queue. Since thread swaps are expensive, it is preferable that they occur infrequently.

1.1.3 Problems with Multiple-Context Processors

Although having multiple contexts can improve processor utilization, and hence performance, a number of factors limit this improvement. Assuming that there is sufficient parallelism in the application[2], these factors include:

[2]In this thesis we do not deal with the problem of not having enough parallelism.

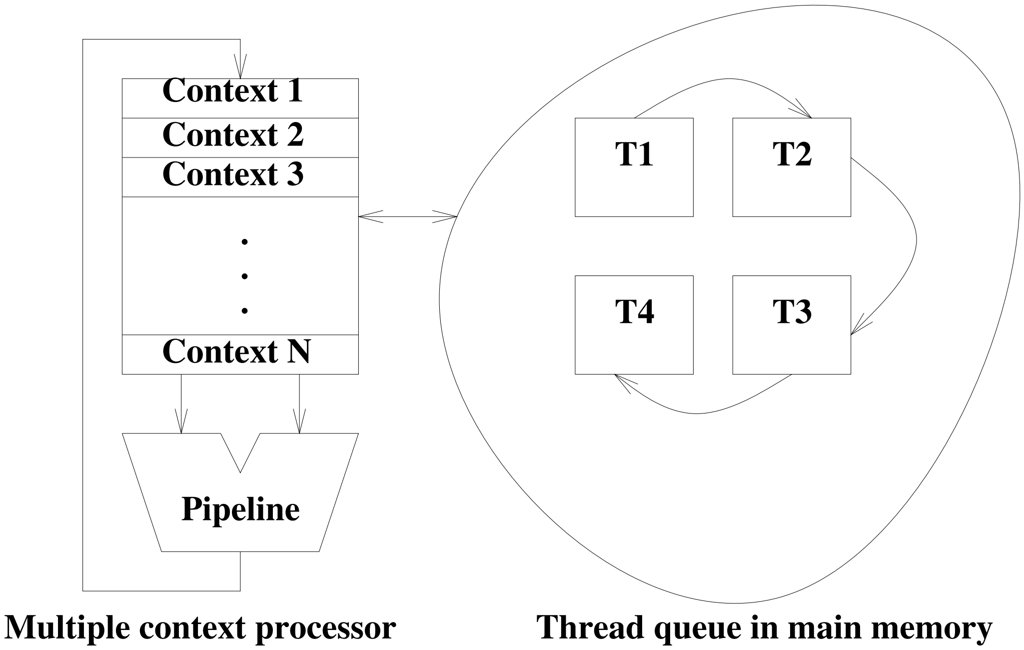


Figure 1.3: Multiple-context processor with N contexts. Loaded threads have their state loaded in one of the hardware contexts. Unloaded threads wait to be activated in a thread scheduling queue in memory.

- Bandwidth effects: the performance improvement with multiple contexts suffers if there is insufficient memory and network capacity to service the increased number of memory and network requests [2, 74]. Increasing the number of requests increases the memory and network traffic and hence the latency. Adding contexts is self-defeating if the latency of requests increases faster than the amount of extra work available to tolerate latency.

- Cache interference effects: Because the contexts share a cache, their working sets can interfere with each other [2, 81, 105, 37, 95]. Techniques for improving cache performance by intelligently placing threads on processors to share data in the cache have proven ineffective [96].

- Critical path effects: Multiple contexts can affect the runtime of the critical path if a critical thread shares resources with other less critical tasks. This has not been explicitly considered in the literature.

- Spin-waiting effects: When a thread spins while waiting for a synchronization event to occur, many cycles can be wasted [112, 48, 67]. A multiple-context processor can switch to executing another thread when it fails a synchronization test [67], but a significant number of cycles can still be wasted, and useful work delayed, especially if several threads are spinning at once.

Naive sharing of the processor resources among the threads is one of the main causes of the performance limiting problems associated with multiple-context processors. Virtually all studies of multiple-context processors assume a round-robin scheduling of contexts [2, 81, 74, 105, 59, 37, 13, 95, 64]. If there are more threads than contexts, then the software scheduler shares the available contexts among the threads so that they all make progress. The hardware scheduler schedules the contexts themselves in round-robin fashion. Even if one thread is more important than the others, it is not given special treatment. Even if there are more than enough threads to tolerate the observed latency, the processor will still run all the threads loaded in the contexts and the cache performance will suffer. Even if a thread is spin-waiting and is not doing any useful work, the processor will still context switch to that thread in the round-robin schedule. In this thesis, we show how more intelligent hardware and software scheduling of threads alleviates these problems.

1.2 Thread Prioritization

In this thesis we introduce *thread prioritization*, a scheduling mechanism that allows the processor to intelligently schedule threads on a multiple-context processor. Thread prioritization is a simple scheduling mechanism that allows the processor to schedule threads so as to try and maintain high processor utilization, and minimize the execution time of the critical path. Each thread is associated with a priority that can change dynamically as the computation progresses, and that indicates the importance of the thread at any given time.

The software uses this priority to decide which threads are loaded in the hardware contexts, and the hardware uses this priority to decide which thread to execute next on a context switch.

Thread prioritization addresses many of the problems caused by the strict round-robin scheduling of contexts. By prioritizing the threads the processor can improve performance by controlling how processor resources are allocated to the threads running in the multiple-contexts. In particular, as we show in this thesis, thread prioritization can reduce the negative cache interference effects due to having more contexts than necessary to tolerate the observed latency, can reduce processor resources devoted to non-critical threads, and can reduce the amount of cycles wasted spin-waiting.

It is important to stress that thread prioritization is a general scheduling mechanism that can be used in many different ways. Good scheduling *mechanisms* provide efficient hardware and software building blocks upon which different scheduling *policies* or *strategies* can be implemented. It is important to be able to implement different strategies because different types of problems require different strategies to achieve good performance e.g. one problem may require dynamic load balancing, while another requires good cache performance. By assigning priorities based on different criteria, thread prioritization can implement scheduling strategies that are aimed at improving synchronization performance, improving cache performance, scheduling the critical path, or a combination of these.

1.3 Contributions

The main contributions we make in this thesis are:

1. Thread prioritization as a general purpose hardware and software scheduling mechanism in multiple-context processors. Thread prioritization is used to address many of the problems associated with naive scheduling of threads on a multiple-context processor by allowing more intelligent thread scheduling.

2. Analytical models that capture the effect of multiple contexts on both processor utilization and the critical path runtime of an application, and that show how thread prioritization can be used to improve performance. Our models incorporate cache and network latency effects, and also show that both spin-waiting synchronization and limited memory and network bandwidth hurt the performance of multiple-context processors. The models show that it is important to extend prioritization so that it prioritizes the use of memory and network bandwidth, because applications can be bandwidth rather than computation limited. Although previous models do consider network [2, 74] and cache effects [2, 81], they do not consider the effect of multiple-contexts can have on the critical path execution time, and do not consider the effects of spin-waiting synchronization.

3. A detailed simulation study of how thread prioritization can be used to improve the performance of different types of benchmarks. Thread prioritization can be used to be used to improve the performance of synchronization primitives, to reduce negative cache effects, and to improve the scheduling of an application's critical path.

4. Data sharing techniques that closely coordinate the threads running in different contexts so that they use common data in the cache at approximately the same time, thus improving cache performance.

The main emphasis of this thesis is on presenting scheduling mechanisms and techniques that exploit the strengths of multiple-context processors, and compensate for their weaknesses. We are concerned with possible benefits, as well as architectural and implementation details. This work suggests important areas for further study including automatic priority assignment, using prioritization in operating system scheduling, and the interaction of multiple contexts and prioritization with other latency tolerance techniques such as prefetching and relaxed memory consistency models. We will touch on these various issues and suggest possible approaches for future work in these areas.

1.4 Outline and Summary of the Thesis

Chapter 2 provides background on the scheduling problem, as well as on multiple-context processors and other latency tolerance techniques. We present a classification of thread scheduling strategies used to improve the runtime of a variety of applications. Specifically, scheduling can be divided into two parts: thread placement (i.e. where tasks run), and temporal scheduling (i.e. when threads run). Within each of these sub-tasks many different approaches are possible. A thread scheduling mechanism's usefulness can be judged in part by evaluating how many different strategies it is useful in implementing. Thread prioritization, the main mechanism we study in this thesis, is a general purpose temporal scheduling mechanism.

Chapter 3 presents our multiple-context processor models with particular emphasis on how multiple contexts can negatively effect the runtime of the critical path. We first develop the model using traditional round-robin scheduling, and consider the effect of network, cache, bandwidth, and spin-waiting synchronization. We then incorporate thread prioritization into the model, and show how it helps solve these problems.

In Chapter 4 we evaluate several different ways of implementing thread prioritization. An efficient hardware prioritization scheme can be used to prioritize the use of the processor pipeline and of limited memory and network bandwidth.

In Chapters 5 through 8 we present our simulation results. Chapter 5 describes the simulation environment and important simulation parameters, while Chapters 6 through 8 describe simulation results from different types of scheduling benchmarks: Chapter 6 deals with synchronization scheduling, Chapter 7 deals with scheduling for good cache performance, and Chapter 8 deals with critical path scheduling. In particular we show that:

- Thread prioritization can be used to substantially improve synchronization performance. For simple synthetic benchmarks such as Test-and-Test_and_Set, barrier synchronization, and queue locks, performance improvement range from 10% to 91% when using 16 contexts.
- Thread prioritization can help implement a number of techniques that improve the cache performance of multiple-context processors. *Data sharing* involves closely coordinating the threads running in each context so that they share common data in the cache. *Favored thread execution* uses thread prioritization to dynamically allow only the minimum number of contexts required to tolerate latency to be running. Cache performance improves because the scheduling minimizes the number of working sets in the cache. Runtime improvements range up to 50% for bandwidth limited applications using 16 contexts.
- Thread prioritization can help schedule threads based on the critical path. If performance is critical path limited then prioritization can have a large impact, 37% for one benchmark using 16 contexts. If performance is not critical path limited, or there is insufficient parallelism to keep the multiple contexts busy, prioritization has little effect.

Chapter 9 briefly looks at how multiple-contexts can be used to reduce software scheduling overhead associated with scheduling threads in response to incoming messages.

Finally, Chapter 10 summarizes the main results of the thesis, and offers concluding remarks.

Chapter 2

Background

In this chapter we give background on both the general scheduling problem and the latency tolerance problem in multiprocessor systems. The general scheduling problem is very difficult and as a result many different heuristic strategies have been used. Scheduling requires both a *temporal scheduling* strategy and a *thread placement* strategy in order to decide when and where threads execute. A thread scheduling *mechanism* is a software or hardware building block that can be used to implement different scheduling strategies. Many of the previously proposed mechanisms only address one particular aspect of either the temporal scheduling problem or the thread placement problem. The mechanisms and techniques proposed in this thesis are general temporal scheduling mechanisms, that can be used to implement many different temporal scheduling strategies. Although we do not deal directly with the issue of thread placement in this thesis, for completeness we describe thread placement strategies in this chapter as well.

Multithreading using multiple hardware contexts has been found to be an effective latency tolerance technique. However, insufficient parallelism and negative cache effects can limit its effectiveness. Alternative techniques such as prefetching and non-blocking loads and stores are also effective for latency tolerance, especially for applications with regular data reference patterns, and little synchronization. Ultimately, the most effective latency tolerance will most likely be achieved with some combination of several different techniques, and this is an interesting area for further research.

Section 2.1 describes the thread scheduling problem and Section 2.2 reviews the many different scheduling strategies that have been found useful in scheduling threads for different types of problems. Section 2.3 looks at thread scheduling mechanisms that have been implemented in various commercial and experimental parallel processors both in hardware and software. Section 2.4 discusses work done on multithreading and other latency tolerance techniques, including prefetching, and non-blocking loads and stores.

2.1 Thread Scheduling

2.1.1 An Application Model

In this thesis we are concerned with minimizing the runtime of a single application, and it is convenient to think of the problem in terms of a simple application model. The execution of any application can be viewed as a Directed Acyclic Graph (DAG), as illustrated in Figure 2.1. Each node represents a task, and each edge represents a dependency between tasks. Any task can execute only once its dependencies are satisfied, i.e., once all its predecessors have executed.

A task can be defined in many different ways. At the most basic level, it can be an abstract task that takes a given amount of time to execute. For the purposes of proving various interesting scheduling theories, it is often assumed that these tasks are of unit or fixed time delay. In practice a task might represent a computer instruction or group of instructions that can take a variable time to execute depending on various system level conditions. When considering a multithreaded computation, it is convenient to think in terms of a simple model similar to the one presented by Blumofe [11]. In this model, each thread is a group of tasks that are executed in sequential order, and edges going between tasks represent different types of dependencies between the threads and the tasks. *Continue* edges represent the sequential ordering within a thread, *spawn* edges represent one thread creating another thread, and *data dependency* edges represent the data being produced by one thread being used by another. An example multithreaded computation is shown in Figure 2.2. As noted by Blumofe, it is important to realize that this type of graph may represent a particular unfolding of the program that is dependent both on the program threads as defined by the user, and the input data.

2.1.2 The Thread Scheduling Problem

The thread scheduling problem involves deciding where each thread should run, and when it should run. Solving the problem optimally is NP-complete [30, 83]. Thus all practical scheduling algorithms are just heuristic strategies used to find a good (hopefully) approximation of the optimal solution. A number of non-ideal factors make it so that even these heuristic strategies must use inexact information to make their decisions. These non-ideal factors are all related to the dynamic nature of the DAG, and include:

- Variable or unknown task costs: The length of each task is variable. Even if each task is broken down to a single instruction, there is a big difference between the cost of a LOAD instruction which hits in the cache, one that misses in the cache but is local to the processor, and one that misses in the cache and requires a shared memory protocol to orchestrate the fetching of the data.

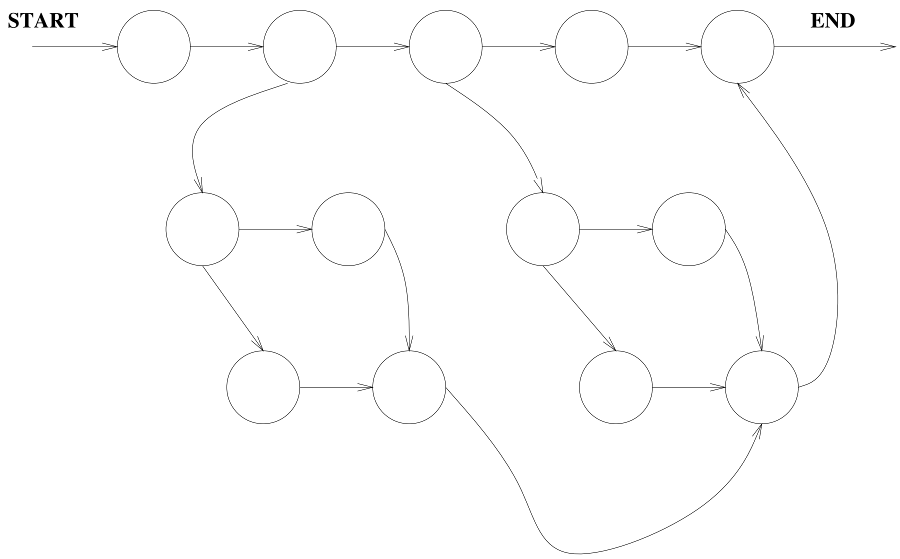


Figure 2.1: General DAG.

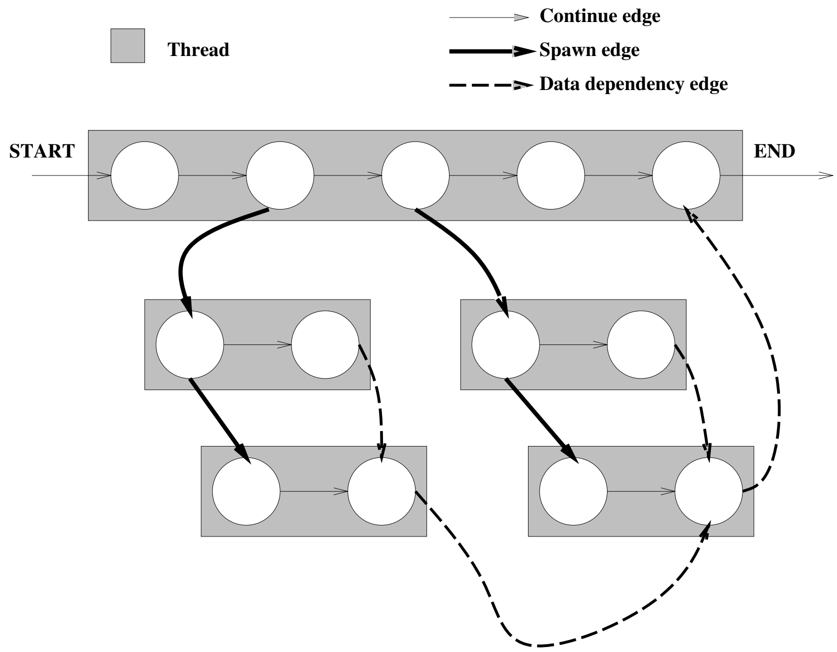


Figure 2.2: General DAG viewed as a set of threads.

- Variable or unknown communication costs: The costs of the dependency edges are non-zero, and depend on the edge type. A continue edge typically represents simply a change in a program counter and is very cheap. A spawning edge on the other hand represents the creation of a new thread, and requires that a context be allocated for the new computation, and that the thread be scheduled in a scheduling queue. The data dependency edges represent data communication between threads which introduces a communication cost. Depending on thread placement, the cost of the edges changes. If a thread is spawned on another processor, then the communication cost must be added. If a data dependency exists between threads on different processors, then this data must be communicated, and the communication cost added in. A continue edge can also have a variable cost if for instance threads can migrate between processors.
- Dynamic DAG generation: The DAG of a computation is often not known a priori, but rather unfolds dynamically as a program executes. For dynamic programs, the number of nodes in the DAG and their interdependence depends on the input data.
- Scheduling overhead: There is overhead associated with doing dynamic scheduling. The scheduling algorithm thus has to be online, and must be efficient. In some sense the scheduling algorithm itself can be seen as adding tasks, edges, and costs to the computation DAG.

Despite the NP-completeness of the scheduling problem, and the non-ideal nature of the data available to make heuristic decisions, there are many different approaches that have been found to lead to good scheduling decisions. Some of the strategies that have been found useful are discussed in the following sections.

2.2 Thread Scheduling Strategies

A complete scheduling algorithm requires that two different types of strategies be defined: a temporal scheduling strategy which decides when threads should run, and a thread placement strategy that determines where threads should run. Temporal scheduling strategies range from precomputed static thread schedules, to dynamically created priority queues. Thread placement strategies range from complete static placement of threads to the use of dynamic load balancing schemes. As we discuss in the next few sections, both the temporal scheduling and the thread placement components of different scheduling algorithms are examined extensively in the literature. The general conclusion that can be drawn from these studies is that different temporal and placement strategies are appropriate for different types of applications. Ultimately then, the ideal is to have a general purpose parallel processor that allows the efficient implementation of all useful scheduling strategies. This thesis focuses on general architectural mechanisms that are useful in implementing different temporal scheduling strategies.

A number of studies and survey papers have looked at classifying different aspects of the scheduling problem, such as load balancing [104, 108], or the specific algorithms used to determine the schedule [17]. Our survey in this chapter is more pragmatic, and provides a reference from which to show that the mechanisms and techniques discussed in following chapters address scheduling problems of interest. Also, any general scheduling mechanism should facilitate the implementation of several of these different scheduling strategies that are appropriate for different types of applications. For instance, we are not so much concerned with the many different possible heuristic strategies that are used to decide when tasks should be scheduled, as with the few mechanisms that should be provided at the hardware and software level that allow the easy specification of a task schedule based on whatever heuristic the user cares to use.

2.2.1 Temporal Scheduling Strategies

In this thesis we will principally be concerned with providing mechanisms that allow the effective and efficient specification of when threads should run. There are a number of program characteristics that can be used as the basis for deciding when threads should run. These include the characteristics of the DAG, the desire to exploit temporal locality, the type of synchronization being used, and the resource requirements of the application.

DAG scheduling

One strategy for deciding when to run different threads is to analyze the program DAG and try and minimize the execution time by carefully scheduling the critical path. Different heuristic strategies [31, 19] decide where tasks should run (see Section 2.2.2), and then given this assignment, decide when each task should run.

If the DAG is dynamically generated at runtime then static DAG scheduling is not possible. However, the user may know which tasks are more important, and want to schedule them first. For instance, in a search problem such as the Traveling Salesman Problem, there may be tasks that are specifically aimed at pruning the tree and reducing the search space. Despite the fact that the exact DAG of the computation is not known, it may make sense to schedule these tasks before tasks that are generating more work.

Temporal Locality

A thread exploits temporal locality in the cache when it brings data into the cache and references it several times. The set of data that a thread needs over a specific period of time constitutes its working set, and if the cache is large enough to hold the working set the processor achieves good cache hit rates. Scheduling decisions can be based on an *affinity* that a task has for a specific processor because its data is likely to be loaded in the processor's cache [90, 26, 97]. Although this affinity scheduling is largely a thread placement issue that requires that a descheduled thread be rescheduled on the same processor (see section 2.2.2), there is also a temporal component: if a task swaps out but soon begins to execute again then some of its data is likely to still be in the cache and will not have to be reloaded, but if the thread has not been run for a long time most of its data will have been removed from the cache.

Different threads can cooperate to exploit temporal locality as a group. If threads that operate on the same data are run at approximately the same time, then they will share data in the cache. If on the other hand threads with largely unrelated data are run at approximately the same time, their data will destructively interfere with each other in the cache. This is particularly important for multiple-context processors where several threads are running at the same time.

Synchronization

Shared memory multiprocessors that use spin-waiting to implement synchronization primitives such as mutual exclusion locks [8, 34, 36, 3, 71], barrier synchronization [111, 71], and fine-grain synchronization using Full/Empty bits [55] raise another set of scheduling issues. The particular problem with spin-waiting types of synchronization is that tasks can be active but not making progress, and thus be uselessly consuming resources [112, 67]. For instance it is typical to have tasks waiting for the release of a lock by spinning on a synchronization variable. In this case it would be best to schedule the specific task that currently controls the lock, or at least, not schedule those tasks that require acquisition of the lock.

Two-phase algorithms have been studied as a method for deciding whether a thread should spin or block (i.e. swap itself out and allow another thread to run) on a synchronization failure [48, 67]. A two-phase algorithm first spins for a determined amount of time in the hope that the synchronization condition will be satisfied and the blocking overhead will be avoided, and then blocks if this is not the case. In particular, Lim and Agarwal [67] study two-phase algorithms in the context of a multiple-context shared memory multiprocessor. Having multiple contexts allows additional strategies such as "switch-spinning", in which a spinning thread can rapidly switch to other contexts, thus doing useful work while waiting for the synchronization to complete.

Resource Utilization

It is important to be able to control the thread generation pattern of a program. Uncontrolled thread generation in problems that exhibit abundant parallelism can overwhelm a parallel machine, exhausting memory or causing severe performance degradation [9, 41]. Conversely, not generating enough tasks can lead to starvation with not enough work for all the processors. Blumofe [11] uses a simple thread model and describes scheduling algorithms that are provably time and space efficient when certain dependency conditions between threads are met.

An example of how scheduling can affect task generation is the expansion of an execution tree in either depth first or breadth first fashion. Scheduling in depth first fashion tends to limit parallelism since at any given time only one path is being followed (as would be done in a serial execution). Scheduling in breadth first fashion tends to generate many tasks, the number of which grows exponentially with the depth of the tree. Ideally, we would first expand the tree in breadth first fashion until there was enough work for all the processors, and then continue in a depth first manner.

2.2.2 Thread Placement Strategies

Although we will not be dealing directly with the issue of thread placement in this thesis, it is the other important component of the thread scheduling problem and is included here for completeness. The most important issues in determining the location of threads are trading off parallelism and communication overhead, load balancing the work across the nodes, and exploiting spatial locality in the memory system.

Parallelism versus Communication

In an ideal system with no communication cost or overhead, the best thing to do is maximize parallelism. In a real system, tasks are on different nodes and must share data and communicate their results to each other. Depending on the communication cost, it may be better to run threads serially in a single processor rather than run them in parallel on different processors. Given a DAG, a number of heuristics can be employed to decide whether given tasks should be run on the same processor [31]. These heuristic approaches start with a DAG that has a computation cost assigned with each node of the graph as well as a communication cost associated with each edge that depends on thread placement. Based on minimizing a cost function such as the parallel running time, they then use different heuristics to merge nodes together in clusters. Performance improvement is achieved because every time two nodes with a direct data dependency are merged, the communication cost becomes 0, potentially decreasing the parallel runtime.

DAG clustering techniques typically make a number of idealistic assumptions to make the problem more tractable. They assume that the tasks are constant length. They assume that the architecture is a completely connected graph so that the communication costs are unaffected by the particular network being used or the network traffic. They assume an unbounded number of homogeneous processors so that if there are fewer processors than clusters, the clustering step must be followed by an assignment of clusters to processors, another NP-complete problem [83].

Despite these assumptions, the heuristic approaches to DAG scheduling can provide good performance results. Also, even if the DAG is not static and depends on the input, it can be advantageous to use these heuristic techniques to determine a good schedule at runtime. This is particularly true if the same schedule can be reused many times [82, 19].

Load balancing

Load balancing involves distributing work across processors so as to minimize running time. Load balancing techniques are either *static* or *dynamic* [104, 17]. Static load balancing schemes assign tasks to processors before the program is run based on execution time and communication pattern information. Once assigned to a given processor, a task remains there for the duration of the computation.

In dynamic schemes, tasks are generated on processors or moved between processors at runtime. Dynamic schemes are either *centralized* or *distributed*. Centralized dynamic load balancing uses either a master process or a centralized data structure to distribute work across the processors. The centralized approach often leads to bottlenecks in the task distribution and is often not appropriate for large systems. Distributed schemes do not have a single point of serialization, and the scheduling data structure and decision making are distributed across all the processors.

Dynamic load balancing schemes differ in the policies they adopt to implement load balancing. Willebeek-LeMair and Reeves [108] define a 4-phase dynamic load balancing model:

- Processor load evaluation: Estimate the amount of work a processor has to do, and use it in deciding whether to load balance.
- Load balancing profitability determination: Determine the degree of imbalance, and decide whether it is worthwhile to do load balancing. The amount of information that goes into making this decision can vary widely from using entirely local information, to centralized schemes that use much more global information but may incur significant overhead.
- Task migration strategy: Determine the source and destination for task migration. Policies include random selection, a fixed pattern such as a simple nearest neighbor pattern, or other more complex patterns. How load balancing is initiated is an important characteristic of the strategy. Load balancing can be initiated at given time intervals, or can be initiated by the producer or the consumer of tasks. If initiated at given time intervals, all the processors cooperate to do the load balancing. In producer-initiated load balancing, a processor with too much work initiates load balancing activity, whereas in consumer-initiated load balancing a processor that needs work initiates the load balancing activity. Consumer-initiated load balancing has the advantages of being more communication efficient [12], and of having the less highly

loaded processors incur most of the load balancing overhead while the highly loaded processors continue executing tasks.

- Task selection strategy: Decide which tasks to exchange. An important issue is whether task migration is allowed, meaning whether threads can migrate between processors once they have begun executing. Migrating a task once it has begun executing can be expensive as it requires that stack information be migrated as well [91].

Spatial Locality

Programs can exploit spatial locality at the cache level and at the local memory level. To do this, data and thread placement policies can place data and threads operating on that data on the same node.

Cache performance improves if different threads use the same data and operate out of the same cache. For instance, in the context of a multiprogramming operating system, space-sharing the processors between applications rather than time sharing the entire machine may result in better performance [98]. A number of studies have looked at affinity scheduling, which schedules tasks on processors to better exploit this locality in the cache [90, 26, 70, 97]. In attempting exploit this locality, tradeoffs are made with load balancing since load balancing and affinity scheduling often are in opposition to each other [69, 70, 91].

Programs can exploit spatial locality at the node memory level by co-locating a thread and its data on the same node. If a thread's data is located in local memory rather than in remote memory, remote memory references can be avoided. In particular, reorganizing data between computation stages to maximize locality can be beneficial [53, 52].

2.3 Thread Scheduling Mechanisms

The goal of thread scheduling mechanisms is to provide efficient building blocks upon which to implement the different scheduling strategies described in the previous section. Hardware mechanisms include such things as hardware support to schedule threads in response to incoming messages, and to manipulate multiple hardware contexts. Software mechanisms may define threads and task queues in ways that allow the easy implementation of different scheduling policies.

2.3.1 Hardware Scheduling Mechanisms

Hardware mechanisms aim at reducing overhead of specific scheduling operations such as scheduling the handling of incoming messages, or at managing the allocation and scheduling of the multiple contexts in a multiple-context processor. They typically lack the flexibility required to do general thread scheduling.

One important hardware mechanism is hardware support for handling incoming messages from other nodes. The processor usually uses some form of automatic enqueuing and direct dispatch to a message handler routine. For instance in the J-Machine [23] there is hardware support for enqueuing tasks in memory as they arrive, without interrupting the processor[1]. When a task arrives at the head of the queue, a direct dispatch mechanism jumps directly to the correct message handler. There are two priority levels each with their own queue of tasks. On the Alewife machine [4] the message interface generates an interrupt and the message is handled in a hardware context reserved for that purpose. Alewife also has special hardware that deals with shared memory protocol messages. In typical dataflow machines, there is specialized hardware for scheduling support. Monsoon [78], for instance, has specialized hardware for dynamically synchronizing and scheduling individual instructions based on the availability of operands. The *T architecture [75] is an example of how dataflow architectures are evolving towards a more conventional multi-threaded approach: it provides special scheduling queues and co-processors for handling memory request messages and synchronization request messages. Henry and Joerg [44] study hardware network interface optimizations that improve the performance of dispatching, forwarding, and replying to messages.

Multiple-context processors such as April [5] or Tera [6] provide mechanisms for managing contexts. April uses the trap mechanism to do a context switch, with the context switch done inside the trap handler. A special instruction changes the context that is executing instructions. The Tera hardware provides special instructions and state for reserving, creating, and de-allocating thread hardware contexts, but the software must decide whether to execute the thread in another hardware context or in the current one [7]. The software generates new threads only when there are hardware contexts available to execute them, otherwise the code executes in the current hardware context.

The hardware mechanisms described above are of limited use for general scheduling because of their lack of flexibility. The hardware message mechanisms aim specifically at reducing the overhead of handling incoming messages, which although important, does not address the larger problem of doing general thread scheduling. The context management mechanisms of the April and Tera processors are useful for managing contexts but they do not address one basic issue: on any given context switch, which context should execute next. The April trap handler could make this type of decision in software, but this would greatly increase the context switch time. Waldspurger [101] proposes a scheme that does the context switching in software in just 4 to 6 cycles, but again it is not clear that scheduling contexts in a way other than in round-robin fashion can be implemented cheaply. The mechanisms proposed in this thesis are more general and are useful for deciding which of multiple available threads should execute, and are not restricted to scheduling messages. The thread prioritization mechanism we propose aims specifically at correctly deciding which context to execute next

[1] Although a memory cycle is stolen when the queue row buffers are written.

in a single cycle.

2.3.2 Software Scheduling Mechanisms

Various scheduling mechanisms are implemented in software as well. These mechanisms are more general then the hardware mechanisms because they allow different scheduling algorithms to be constructed on top of them.

Lazy task creation [72] is one example of a flexible scheduling mechanism. Lazy tasks are a means of allowing dynamic partitioning of tasks. New tasks are created at runtime only as needed to keep all the processors busy, thus providing a mechanism for both increasing the granularity of tasks and throttling excess parallelism. This mechanism is flexible as it allows the possibility of many different load balancing strategies to be implemented on top of the lazy task creation model.

Culler et al. [21] have proposed TAM, an execution model for fine grained parallelism that uses a multilevel software scheduling hierarchy. They follow the basic dataflow model in which a thread does not execute until all its arguments are available, and a thread always runs to completion. They provide a basic system for scheduling related threads as a *quantum*: threads related to specific code block invocation are run at approximately the same time so as to exploit locality. A higher level scheduler schedules these quanta on the processors, and this higher level scheduler can implement different scheduling policies.

At the operating system level, Waldspurger and Weihl's Lottery Scheduling [102] provides a mechanism that allows flexible control over the relative execution rates of different tasks. Further, this mechanism can be generalized to manage different types of resources such as I/O bandwidth and memory.

Thread prioritization as proposed in this thesis is also a software scheduling mechanism that allows different scheduling algorithms to be implemented on top of it. Rather than aiming at dynamic load balancing like lazy tasks or locality improvement like TAM, it aims at allowing a flexible specification of when threads should run relative to each other. Unlike Lottery Scheduling, it is aimed specifically at scheduling the threads in a single application. Each thread has a priority that can be assigned and changed dynamically, and used by the thread scheduler to decide which thread to run at any given time. This priority can also be used to improve data locality as discussed in Chapter 7, and could potentially be useful in making load balancing decisions.

2.4 Multithreading and other Latency Tolerance Techniques

Different techniques are useful for tolerating the long communication and synchronization latencies that occur in parallel processors, including multithreading, multiple-contexts, prefetching, non-blocking accesses, and relaxed consistency models. This thesis is specifically concerned with the use of multithreading and multiple-context processors, but it is probable that the ideal set of latency tolerance techniques involves some combination of all of these.

2.4.1 Multithreading

Multithreading involves dividing a problem into multiple tasks and then running these tasks in parallel to achieve better performance. Latency is tolerated by running a different task whenever a long latency operation occurs. Analytical [2, 81, 46, 74] and experimental [105, 59, 37, 13, 95, 64] studies show that it is a good technique for tolerating latency but that lack of parallelism, as well as cache and network effects can limit performance.

The analytical models [2, 81, 46, 74] use processor utilization as a metric. They show that network bandwidth limits performance because it limits the number of requests that can be outstanding without seeing a substantial increase in the network latency. Cache performance can suffer with multiple contexts because of increased cache miss rates that occur when the working sets of different threads are trying to occupy the cache at the same time [2, 81]. Some of these studies [46, 74] consider the feedback that occurs between the different subsystems of the multiprocessor more carefully and show how this feedback can limit the maximum message rate of the network. None of these models specifically consider the effect of multithreading on the execution time of an application's critical path, or the effect of spin-waiting synchronization latencies.

Simulation studies have also shown some of the benefits and weaknesses of multithreading [105, 59, 37, 13, 95, 64]. These studies show that substantial performance improvements are possible provided that certain conditions hold. Specifically, there must be sufficient parallelism in the application, the context switch time must be low, the run length to latency ratio must be favorable, the distribution of run lengths must be favorable (in particular clustered misses can reduce multithreading effectiveness), and the negative cache effects must be minimal.

2.4.2 Multiple-Context Processors

Having multiple hardware contexts is one useful technique for supporting efficient multithreading. Multiple-context processors come in several different flavors, depending on how instructions from the different contexts are issued. *Block* multithreaded processors run blocks of instructions from each context in turn [105, 5]. This allows a single thread to fully use the processor pipeline, though the data dependencies within a single thread can introduce pipeline bubbles. A context switch occurs on special interrupts or on long latency operations such as a miss in the cache, or a synchronization event. *Finely* multithreaded architectures interleave instructions from different contexts on a cycle-by-cycle basis. Some of these architectures concentrate on eliminating pipeline dependencies by having each context issue instruction only once every D cycles, where D is the pipeline depth [88, 42]. All the instructions in the pipeline are independent from one another since they are from different contexts. The performance of these architectures suffers when there are not enough threads to fully use the pipeline. More aggressive designs provide pipeline interlocks which allow any ready thread to issue an instruction provided it satisfies data and pipeline dependencies [50, 64]. This dynamic interleaving of instructions hides pipeline stalls as well as long latency operations.

The type of multithreading performed depends very much on the design philosophy and budget for the processor. Block multithreading allows the most conventional processor design, and only requires support for multiple register sets. For instance the April processor is a commercial processor that has been modified to provide 8 hardware registers sets to support multithreading [5]. Other schemes for providing multiple register sets are possible, including the mostly software scheme presented by Waldspurger and Weihl [101], and the hardware intensive context cache presented by Nuth [77]. Cycle-by-cycle interleaving has a significant hardware cost to redesign the processor core and pipeline to allow the different contexts to issue instructions simultaneously [50, 64].

2.4.3 Other Latency Tolerance Techniques

Prefetching

Prefetching tolerates latency by requesting data before it is required. Latency is minimized if the data has arrived before it is referenced, or is reduced if the data has not arrived but is on its way. Prefetching schemes are either *binding* or *non-binding* [73]. A *binding* prefetch is one in which the value of the requested data is bound at the time the prefetch completes rather than when the actual load occurs. A *non-binding* prefetch is one in which the requested data is brought close to the requesting processor (i.e. into its cache), but the value is not bound until the actual reference occurs. Prefetching can be implemented in hardware or software.

The fact that prefetched data may become stale if it is modified between the prefetch and the subsequent load [65] limits binding prefetch schemes. In the uniprocessor case this can occur when a write to the same address occurs between the prefetch and the load. In the multiprocessor case it can also occur when another processor modifies the value. Non-binding prefetches have the advantage that they are simply hints to the memory system and do not have semantic significance to the program. Thus they can be placed anywhere in the program and not affect correctness. When latencies become large, the flexibility of being able to place a prefetch well in advance of its subsequent reference becomes important.

The benefits of hardware prefetching are that it discovers the locations to prefetch dynamically, and that it has no extra instruction overhead. However, the more effective schemes are still only good at predicting very simple access patterns, require non-trivial hardware modifications to do limited instruction lookahead, branch prediction, and stride prediction [89, 10]. The amount of latency they can hide is limited by these factors. The benefits of software prefetching [35, 16, 51, 85] are that it requires only the addition of a prefetch instruction, the prefetching can be done selectively so as to reduce the bandwidth requirements, and the prefetches can be positioned to better tolerate long latencies (if non-binding). The disadvantages are that there is extra overhead due to the prefetch instructions themselves, and due to the fact that addresses have to either be calculated twice (once at prefetch time and once at load time), or have to be preserved in a register between the prefetch and the load.

Non-Blocking Accesses and Relaxed Consistency Models

Another way to tolerate latency is to pipeline memory accesses, by allowing non-blocking loads, and by buffering store operations. When a load operation misses in the cache or if there is no cache for shared data, the processor continues executing code, including issuing other loads, until it actually needs the data. Only if the data is needed and it still has not arrived does the processor stall. The processor can pipeline write operations by using write buffers.

To tolerate long latencies it is desirable to move a non-blocking load as far ahead of the instructions that use the returned value as possible. However, non-blocking loads have the same semantics as binding prefetches into registers, and just like binding prefetches, memory disambiguation is required in order to guarantee that a non-blocking load is not moved before a write to the same address. Furthermore, the non-blocking loads require the use of a register to store the value until it is needed. This puts additional pressure on the register file, especially when the loads are moved far ahead of their use. Non-blocking loads require a synchronization mechanism such as Full/Empty bits on the registers in order to signal when a non-blocking load has been completed.

In multiprocessor systems, performing non-blocking accesses is more complicated. This is because multiple processors are reading and writing the same data at the same time, and if not restricted in some way, memory accesses can perform in unexpected orders. The extent to which non-blocking loads and buffered writes can be used to hide latency in multiprocessors is restricted by the *memory consistency model* used, which is the set of allowable memory access orderings [62, 27, 33]. For instance a common model is the *Sequential Consistency* [62] model which requires that the memory accesses appear as if performed by some interleaving of the processes on a sequential machine. Unfortunately, this severely restricts the use of non-blocking loads and stores to tolerate latency [32]. Relaxed consistency models attempt to remove some of these restrictions, to allow better use of non-blocking accesses, while still providing a reasonable programming model.

2.4.4 Comparison of Techniques

Different latency tolerance techniques are appropriate in different situations. Prefetching and non-blocking accesses are good for improving the runtime of a single thread, but do not provide any way of tolerating synchronization latencies. Multithreading requires sufficient parallelism to be effective, does not improve single thread performance, but can be effective in tolerating long synchronization latencies.

Gupta et. al. [37] study different latency tolerance techniques in the context of a shared memory multiprocessor. They conclude that non-blocking accesses in conjunction with prefetching is the most effective in tolerating both read and write latencies. Non-blocking and multithreading was also quite effective, because the non-blocking accesses allowed longer run lengths between context switches, and fewer contexts were necessary to tolerate latency. However, multithreading was not effective in some cases due to cache effects and limited parallelism. They also found that multithreading and prefetching together could actually hurt performance. First, because using both methods adds the overhead of both methods even though only one may be needed to hide the latency. Second, the methods can affect each other in the cache e.g. the time between when data is prefetched and when it is actually used can become long due to intervening executing contexts, and the prefetched data has a higher probability of being removed from the cache before it is used. They do note however that the prefetch instructions were not added with the knowledge that multithreading was also being used. It is hard to draw any general conclusions from this study since only a small number of benchmarks are used, and these are fairly coarse grain benchmarks with limited synchronization. It is clear however that these latency tolerance techniques can be complementary, and this is an area where further research is needed.

2.5 Summary

Scheduling consists of two components: deciding when threads should run and deciding where threads should run. Temporal scheduling strategies attempt to optimally schedule threads based on the form of the DAG, to exploit temporal locality, to manage resource utilization, and to optimize synchronization scheduling. Thread placement strategies attempt to deal with the tradeoff of parallelism and communication, to do load balancing, and to exploit spatial locality. Which strategies are most appropriate depends on the characteristics of the problem, the architecture, and the programming model. Thread scheduling mechanisms are aimed at providing efficient building blocks for implementing different scheduling strategies. A general purpose parallel processor should provide the mechanisms required to implement a wide range of these strategies. This thesis will present a number of scheduling mechanisms aimed at implementing different temporal scheduling strategies in a multiple-context processor. Specifically, strategies that schedule tasks based on the program DAG, that schedule threads to exploit temporal locality, and that schedule threads to improve synchronization performance.

In large scale multiprocessors it is necessary to find ways of tolerating long latency operations. Multithreading using multiple-context processors has been found to be a useful technique for doing this, but performance can be limited by lack of parallelism, by network bandwidth, and by cache performance degradation. Other latency tolerance methods, including prefetching and relaxed consistency models, are also useful in tolerating latency. An interesting topic of research is to determine which combinations of methods should be used for any given problem.

Chapter 3

Thread Prioritization

In this chapter we introduce thread prioritization, a mechanism that allows the processor to devote pipeline and bandwidth resources preferentially to high priority threads in order to increase the utilization of the processor, and decrease the runtime of the critical path. We use an analytical model to show how thread prioritization can be used to improve the performance of multiple-context processors.

Our model is based on existing multiple-context scheduling models, and incorporates both network and cache effects. Unlike other models, it considers not only how multiple contexts affect processor utilization, but also how the multiple contexts can affect the runtime of the critical path. Further, it considers the effect of spin-waiting synchronization and limited memory and network bandwidth on multiple-context performance. The model shows that with simple round-robin scheduling, both processor utilization and the runtime of a critical thread can be hurt when the number of contexts exceeds the minimum number required to tolerate latency. Processor utilization is hurt because the working sets of the cache interfere with each other, leading to more cache misses, more network traffic, and longer latencies. The runtime of a critical thread can increase substantially if there are many contexts because the execution of the critical thread can be delayed by other threads using the processor pipeline, network, and memory resources. Spin-waiting decreases processor utilization and increases the runtime of the critical path because spinning threads do no useful work even when they are occupying the pipeline. Limited memory and network bandwidth can also affect utilization and the critical path. Bandwidth requirements increase with an increasing number of contexts and contexts can be delayed waiting for bandwidth resources.

Thread prioritization is a mechanism that allows contexts to be scheduled based on a priority scheme rather than in round-robin fashion. Each thread is assigned a priority, and on each context switch the context with the highest priority that is ready to execute is chosen next. The model shows that in the case that there are more than enough contexts to tolerate the observed latency, thread prioritization can be used to dynamically choose a minimum set of contexts required to tolerate the observed latency to execute at any given time. This improves cache performance, and as a result processor utilization, because a smaller number of working sets are trying to occupy the cache. Thread prioritization can also be used to minimize the impact of multiple contexts on the runtime of a critical thread, by allowing the critical thread to execute every time it is ready. We show that for bandwidth limited applications we must prioritize the use of bandwidth. Thread prioritization can be used to choose a minimum set of contexts necessary to saturate the available bandwidth, and it can give priority to accesses by critical threads.

The model makes a number of simplifying assumptions having to do with run lengths between context switches, the cache behavior, and network traffic patterns. However simulations done in later chapters using real applications confirm the trends predicted by the model.

Section 3.1 introduces thread prioritization. Sections 3.2 and 3.3 develop our multiple-context processor model without thread prioritization, and finally Section 3.4 shows the effect of thread prioritization on the model.

3.1 Thread Prioritization

Thread prioritization involves associating a priority with each thread based on knowledge about how threads should be scheduled. It is a method for encapsulating in a hardware-and-software-usable form the best guess at identifying the critical thread or threads. This information is used to bias the allocation of processor resources to those threads.

Thread prioritization is dynamic, since which threads are critical can change as the computation unfolds, especially in situations that use spin-waiting synchronization where the critical thread is not known a priori, but only once the synchronization occurs. Furthermore, having many priorities allows more descriptive information to be encoded, such as an estimate of the second most critical thread, and so on. Thread prioritization is a very flexible means of specifying the relative importance of different threads.

3.1.1 Software and Hardware Priority Thread Scheduling

Prioritizing Execution

Consider an application that consists of a set T of threads on each processor, where each processor has C contexts. Each thread $t_i \in T$ has a priority p_i, with a higher value of p_i indicating a higher thread priority. The hardware and software schedulers use the priority to do the scheduling.

First, the software scheduler uses the priority to decide which threads are loaded. Specifically, it chooses a set T_L of threads to load into the C contexts, and a set T_U of unloaded threads to remain in a software scheduling queue. The scheduler chooses the loaded threads such that $p_l \geq p_u$ for all $t_l \in T_L$ and $t_u \in T_U$. Threads of equal priority are scheduled in round-robin fashion.

Second, at each context switch the hardware scheduler uses the priority to determine which loaded thread to execute. At any given time some threads will be ready to execute and others will be stalled waiting for a memory reference to be satisfied. If T_R is the set of ready, loaded threads, the scheduler chooses a thread $t_x \in T_R$ such that $p_x = max\{p_r\}$ for all $t_r \in T_R$. If several loaded threads have the same priority, then these threads are chosen in round-robin fashion. A context switch can occur on a cache miss, on a failed synchronization test, or on a change of priority of one of the threads on the processor. Each change in priority results in a re-evaluation of T_U, T_L, and t_x. In this sense, the scheduling is preemptive.

Prioritizing Bandwidth

The processor pipeline is only one of the important processor resources that can limit performance. Limited memory and network bandwidth can also limit performance. Thus we extend the notion of prioritization to include the prioritized use of both memory bandwidth and network bandwidth. Any context's long latency transaction that is waiting for either the memory resource or the network resource sits in a *transaction buffer* waiting for that resource to become available. In the case that the application is memory or network limited, there can be several transactions waiting for either the memory or the network. With prioritized scheduling, when the resource becomes available the highest priority transaction that needs the resource is issued. Thus the priority of the thread extends to the use of the memory and network bandwidth resources, as well as to the use of the processor pipeline.

3.1.2 Assigning Priorities: Deadlock and Fairness

In our benchmarks of later chapters, the user explicitly assigns a priority to each thread, and changes this priority as the application requires. Although initially the use of thread prioritization is likely to be limited to special runtime libraries (e.g. synchronization primitives) and user-available program directives, we expect that it will eventually be possible to have a compiler assign priorities to threads automatically. Automatic thread prioritization is particularly straightforward when the program can be described as a static DAG.

Prioritizing threads incorrectly can lead to a number of deadlock situations. Specifically, if thread A is waiting for another thread B to complete some operation, and thread B has a low priority that does not allow it to be loaded, deadlock results. There are a number of ways of overcoming this problem, including guaranteeing that the priorities of the threads respect the dependencies of the computation, implementing priority inheritance type protocols [86], or guaranteeing that all threads will receive some amount of runtime even if they are lower priority [102].

For our benchmarks we make sure that the thread priorities respect the dependencies of the computation, and that scheduling is fair only between threads of the same priority. This is the lowest cost alternative for avoiding deadlock, and avoids the overhead of more complicated protocols. Also, as we will show in the examples, doing fair scheduling without regard to priority, or not specifying the priority of threads as exactly as they could be, can lead to a serious performance penalty. It is in our interest to prioritize threads as exactly as possible.

3.1.3 Higher Level Schedulers

The thread scheduling as defined here is purely a local operation. Each processor has its own set of threads, and schedules only these. The assigning of threads to processors is governed by a higher level scheduler, for instance a scheduler that dynamically load balances work between processors by moving threads between them. Note that the thread priorities may be useful to the global scheduler for making its scheduling decisions.

It should also be noted that the thread scheduler uses the thread priority in a very different way than process scheduling in the UNIX operating system for instance, where the goals and constraints are different. The thread scheduler always executes the highest priority ready thread, and is not concerned with fairness or guaranteeing progress of all the threads. Operating system schedulers, on the other hand, aim at achieving good interactive performance, at achieving time sharing between competing processes, and guaranteeing some progress for all jobs [66]. The operating system schedulers also make decisions at a much larger scheduling granularity: the scheduling algorithm can be fairly heavyweight since a process will run for many thousands of cycles before the next process switch occurs. In multiple-context scheduling a scheduling decision is made on every hardware context switch that can take place every few cycles, and must be very inexpensive. The goal of thread prioritization is to identify as exactly as possible which threads are most important and devote as many resources as possible to these threads.

3.2 Effect of Multiple Contexts on the Critical Path

In this section, we develop a simple model to gain intuition about the effect of multiple contexts on processor utilization and critical path execution time. This simple model shows that the processor utilization reaches a peak utilization once it has enough contexts to completely tolerate latency. It also shows that if we increase the number of contexts beyond the minimum required to tolerate all the latency, the runtime of a critical thread suffers.

We consider a number of important special case scenarios including cases where there are threads spin-waiting, and cases in which the application is bandwidth limited. If threads are spin-waiting then both the processor utilization and the critical path suffer because spinning threads use processor cycles doing synchronization tests and generating unnecessary context switches. Limited memory and network bandwidth limits the effectiveness of multithreading when there is not sufficient bandwidth to handle the increased number of requests coming from the multiple contexts.

The simple model neglects cache and network effects that occur when there are multiple contexts. These effects are incorporated into the model in Section 3.3. Thread prioritization is incorporated to the model in Section 3.4.

3.2.1 Total Work and the Critical Path

As discussed in Chapter 2, a general task graph can be viewed as a DAG in which the nodes represent the tasks, and the edges represent the dependencies between tasks. The total work in a DAG is simply defined as the total number of tasks, and the computation depth is the length of the longest directed path in the DAG. Borrowing notation from Blumofe [11], we define T_N as the time to execute the DAG with N processors using a best case schedule. In this case T_1 represents the total work of the computation, and T_∞ represents the computation depth. Trivially, it is clear that $T_N \geq T_1/N$, and that $T_N \geq T_\infty$. Brent's theorem [14] shows further that $T_N \leq T_1/N + T_\infty$. The important intuitive idea is that the execution time is limited by two factors: first by the amount of work that has to be done, and second by the critical path through the DAG.

3.2.2 Previous Models

Previous work on modeling the effects of multiple contexts has concentrated on the ability of multiple contexts to increase processor utilization, and in this way execute more work in a given amount of time [2, 81, 46, 74]. Agarwal [2] presents a model in which he considers the effects of context-switch overhead, network contention, and cache interference due to the multiple contexts. Saavedra-Barrera, Culler, and Eicken [81] develop a mathematical model in which they identify three operation regimes for multithreaded contexts: a linear region in which processor efficiency is proportional to the number of threads, a transition region, and a saturation region in which the efficiency depends only on the run length between context switches and the context switch overhead. This model takes into account variable run lengths by assuming that the run length of a thread is a random variable having a geometric distribution, and includes a first order model of the cache effects of multiple contexts. They do not take into account the variation in memory latency due to increased memory traffic. A number of researchers including Johnson [46] and Nemawarkar et. al. [74] have emphasized the importance of incorporating the feedback effects between the subsystems of the multiprocessor, especially the feedback that occurs between the processor and the network. They show with a limited number of memory requests per processor, latency and the maximum message rate are limited. Neither of these studies considers cache effects.

The main emphasis of all these models is evaluating performance based on processor efficiency or utilization. Thus, they address the issue of executing large amounts of work, but do not address the issue of how the critical path is affected by the scheduling of the multiple contexts. Also, they do not consider long synchronization latencies or the particular effects of spin-waiting synchronization. The following sections examine these issues.

3.2.3 Metrics and Parameters

We consider two metrics when examining multiple context execution: the utilization U, and the runtime of a critical thread T_c. U is the fraction of time that the processor spends doing useful work, and T_c is the amount of time it takes a critical thread to complete execution. In evaluating the effect of multiple contexts on T_c we will typically be interested in the critical thread runtime ratio T_c/T_{c1} where T_{c1} is the runtime of the critical thread when there is only one context. It is important to note that the lengthening of the runtime of a critical thread does not necessarily translate into a lengthening of the application run time. Consider for instance a computation in which all the threads have the same total runtime. Although the run time of each individual thread is increased by the multithreading, improving performance relies almost entirely on the processor maximizing processor utilization. However, if the application is critical path limited, then the increased runtime of a critical thread will have an effect on the overall application runtime.

The metrics U and T_c/T_{c1} are quantified in terms of a number of basic parameters shown in Table 3.1. P is the number of contexts, R is the run length between context switches, C is the context switch overhead, L is the memory latency, and I_c is the number of instructions executed by the critical thread. The main parameters we will vary are the number of contexts P and the run length R. We will first assume that R and L are independent of each other and of P. The complete model of Section 3.3 takes into account their interdependence. For the purpose of calculating the runtime of a critical thread we use the parameter I_c, the number of instructions executed by a critical thread. The number of context switches done by the critical thread is just I_c/R. Finally, when we consider spin-waiting, the parameter P_s corresponds to the number of contexts that are spinning, and R_s corresponds to the time for a spinning thread to do a synchronization test.

Note that a context switch occurs on every cache miss. In the case that a thread is spinning, a context switch is performed explicitly by the spinning thread. Also note that we are assuming that there is only one critical thread on a processor at any given time. The values for the different parameters shown in Table 3.1 will be used for illustrative purposes in the following sections.

Parameter	Description	Value
P	Number of contexts	1-16
R	Runtime between context switches	4-64 cycles
C	Context switch time	5 cycles
L	Latency	20-100 cycles
L_r	Cycles required between memory requests	5-20 cycles
I_c	Number of instr. executed by a critical thread	1024
P_s	Number of spinning contexts	0-16
R_s	Time to do a synchronization test	5 cycles

Table 3.1: Basic model parameters.

3.2.4 Basic Model

Figure 3.1 shows a simple diagram of how execution proceeds on a single processor, assuming there are P contexts, scheduled in round-robin fashion. There are 2 cases to be considered. In the first case (Figure 3.1a), the computation is *communication bound*, meaning that the latencies are long enough that there are not enough contexts to hide all the latency ($(P-1)R+PC < L$). In the second case (Figure 3.1b), the computation is *computation bound* meaning that there are enough threads to hide all the communication latency ($(P-1)R + PC > L$). In general, assuming P threads executing at once, the processor utilizations in the communication limited region (U_{comm}) and in the computation limited region (U_{comp}) are given by the following equations:

$$U_{comm} = \frac{PR}{R+L} \tag{3.1}$$

$$U_{comp} = \frac{R}{R+C} \tag{3.2}$$

This tells us something about the processor utilization, but it does not tell us anything about the critical path. To study this, suppose that thread 1 of the P threads being executed is on the critical path, and consider how the other executing threads affect the performance of this critical thread. If the computation is communication bound and we assume a thread is ready to begin executing as soon as its long latency memory request is satisfied[1], then to first order there will be no effect on the critical path. If however the computation is computation bound, then on each cache miss the critical path will be increased by an

[1]This assumes a signaling mechanism in which contexts waiting for long latency operations are inactive, and are woken up when the memory request is satisfied. If contexts are polling instead of stalling, there is extra delay due to extra context switching. The effect of polling is the same as the effect of spin-waiting discussed in the next section.

amount $(P-1)R + PC - L$. In terms of the basic parameters, it is easy to show that the execution time of the critical thread in the communication limited (T_{c_comm}) and the computation limited (T_{c_comp}) regions is given by:

$$T_{c\_comm} = I_c + ((I_c/R) - 1)L \tag{3.3}$$
$$\approx (I_c/R)(R + L) \tag{3.4}$$

$$T_{c\_comp} = I_c + ((I_c/R) - 1)((P-1)R + PC) \tag{3.5}$$
$$\approx (I_c/R)P(R + C) \tag{3.6}$$

$(I_c/R - 1)$ is the number of times the critical thread context switches, and the approximations in equations 3.4 and 3.6 hold when $(I_c/R) >> 1$.

Figure 3.2 shows the utilization U, and the ratio T_c/T_{c1}. With small values of R it takes more threads for the processor to be working at its maximum utilization rate, and this maximum utilization rate increases with increasing R. The runtime of the critical thread remains unaffected until there are more contexts than necessary to tolerate latency at which point it begins to increase. Thus T_c is made worse by increasing R and there is a tradeoff between guaranteeing maximum U and minimizing T_c. T_c/T_{c1} is also worse for L=20 than for L=100 because the computation limited region is reached sooner, and the extra delay due to the increased number of contexts is more important.

3.2.5 Spin-waiting Synchronization

When spin-waiting is used to do synchronization in a multiple-context processor, both processor utilization and the critical path performance can suffer. Synchronization performance is a very important parameter of any parallel processor, especially in terms of latency tolerance, because the latencies can be much longer than simple remote reference latencies. Spin-waiting is an attractive, low overhead way of allowing threads to wait at a synchronization point without incurring the overhead of swapping the thread out of the context [71]. The thread repeatedly checks a value in shared memory until it becomes equal to a certain value, at which point the synchronization condition is satisfied, and the thread can proceed beyond the synchronization point.

In multiple-context processors, several threads may be spin-waiting in different contexts on the same processor. Processor utilization can suffer because each time a spinning thread unsuccessfully checks a synchronization variable, it uses cycles to do the context switching and to do the flag checking. These cycles could potentially be used by some other thread to

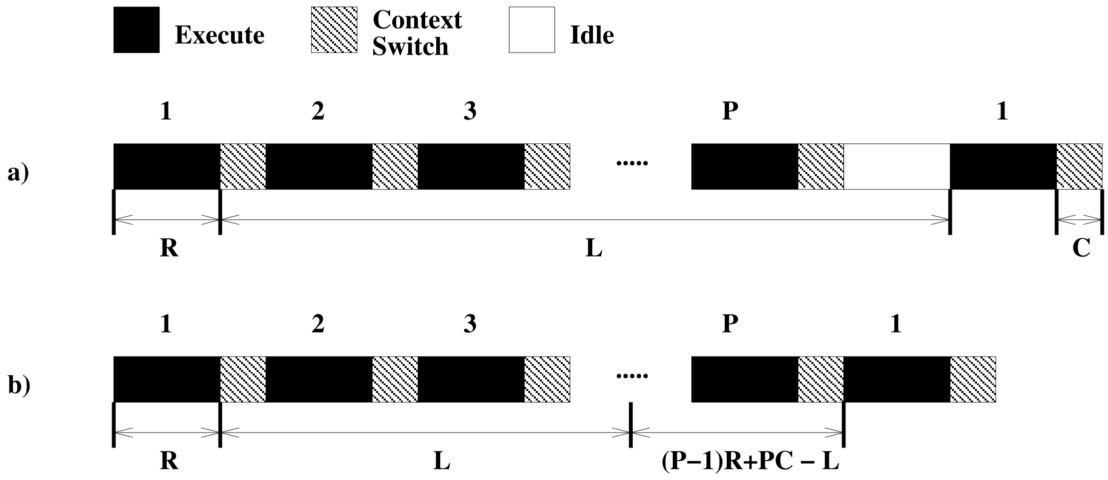


Figure 3.1: Multithreading using P contexts. a. Communication bound ($(P-1)R+PC<L$). b. Computation bound ($(P-1)R+PC>L$).

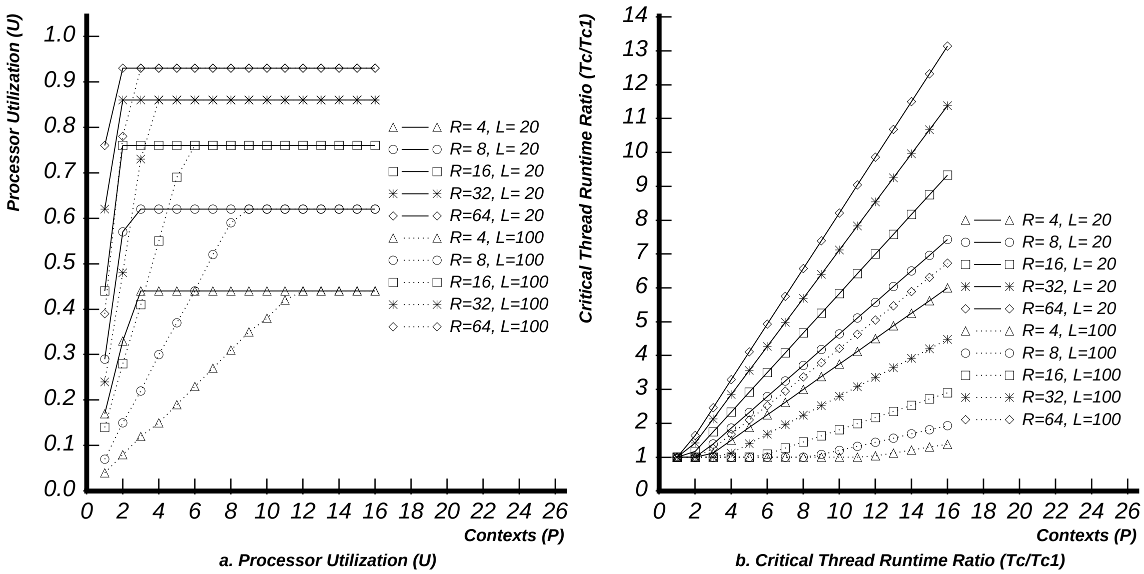


Figure 3.2: U and T_c/T_{c1} for different values of R (4, 8, 16, 32) and L (20, 100).

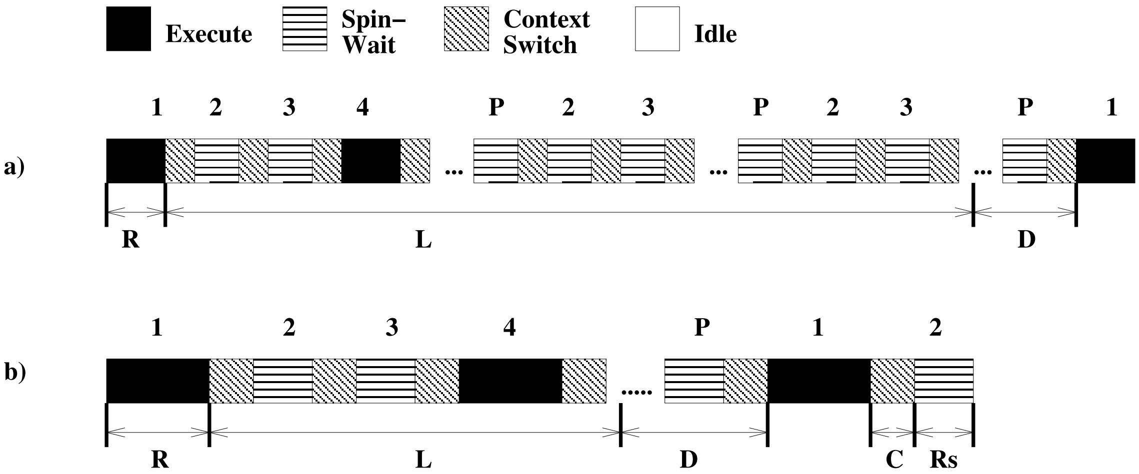


Figure 3.3: Multithreading assuming some threads are spin-waiting. D is the extra time added to the execution of a critical thread. a. Communication bound ($(P - P_s - 1)R + P_sR_s + PC < L$). b. Computation bound ($(P - P_s - 1)R + P_sR_s + PC > L$).

do useful work. The runtime of a critical thread can also suffer because the critical thread is delayed while other threads execute spin-wait cycles.

The effect on processor utilization and the critical path is illustrated in Figure 3.3. The time D represents the extra delay in the critical path execution for each time the critical thread context switches. A spinning thread checks its flag and if it is unsuccessful does a context switch[2]. Some threads are shown as spin-waiting, and some threads are shown as doing useful work. In this case we use two additional parameters to express the processor utilization: the number of spinning threads, defined as P_s, and the time to do an unsuccessful check of the synchronization variable, defined as R_s. The computation is communication bound when there is sufficient latency for all spinning threads to check their synchronization variable at least once, and all the non-spinning threads to take a cache miss. This is true if $(P - P_s - 1)R + P_sR_s + PC < L$. Otherwise, the computation is computation bound. The resulting equations for the expected values of the processor utilization and the critical path thread are shown in equations 3.7 through 3.10.

$$U_{comm} \approx \frac{(P - P_s)R}{R + L + P_s(R_s + C)/2} \tag{3.7}$$

[2]A number of different policies are possible when spin-waiting that involve deciding on whether to spin, to swap out the thread and free the context, or to spin multiple times before eventually swapping [67].

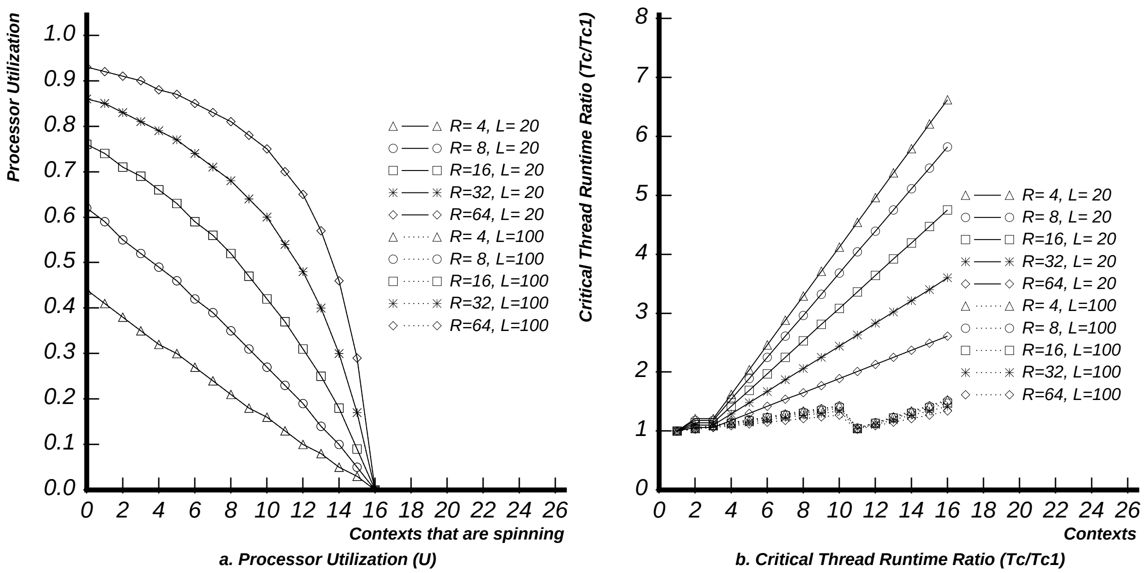


Figure 3.4: U and T_c/T_{c1} when threads are spin-waiting for different values of R (4, 8, 16, 32, 64) and L (20, 50). a. Processor utilization U assuming that there are 16 threads running and that an increasing number of these threads are spin-waiting. b. Critical thread runtime ratio T_c/T_{c1} assuming that only one thread is running and all the other threads are spin-waiting.

$$U_{comp} = \frac{(P - P_s)R}{(P - P_s)R + P_s R_s + PC} \tag{3.8}$$

$$\begin{aligned} T_{c\_comm} &\approx I_c + ((I_c/R) - 1)(L + P_s(R_s + C)/2) \\ &\approx (I_c/R)(R + L + P_s(R_s + C)/2) \end{aligned} \tag{3.9}$$

$$\begin{aligned} T_{c\_comp} &= I_c + ((I_c/R) - 1)((P - 1 - P_s)R + P_s R_s + PC) \\ &\approx (I_c/R)((P - P_s)R + P_s R_s + PC) \end{aligned} \tag{3.10}$$

Note that equations 3.7 and 3.9 assume that the processor will have to run through exactly half the spinning contexts before resuming execution of the critical thread ($P_s(R_s + C)/2$ cycles).

Figure 3.4a shows U when there are 16 threads, and the number of these threads that are spinning is gradually increasing from 1 to 16. Utilization drops off as an increasing number of spinning threads begin using cycles. The curves are the same when L=20 as when L=100 because the processor is always in the computation bound region with this number of threads.

Figure 3.4b shows T_c/T_{c1} in a slightly different situation: only one thread is running useful instructions, and all the other threads are spinning. When the latency is only 20 cycles, there is a very large impact on the runtime of a critical thread because the time to switch through all the other spinning contexts is much longer than the 20 cycles for the critical reference to be satisfied. When the latency is 100 cycles, the time spent running through the other contexts is not so important since the critical thread would have had to wait a long period anyway. The discontinuity of the curves at 3 contexts for L=20, and at 11 contexts for L=100 represent transitions from the communication bound region to the computation bound region, and are due to the simplifying assumptions we made regarding the communication bound region. We assumed once the critical reference was ready the processor would still have to on average run through half the spinning threads before resuming the critical thread. In fact we expect the model to show several discontinuities in the communication bound region as the number of spinning threads increases: in some cases the timing is such that the processor has to switch through only a few of the spinning contexts before resuming the critical thread, and in other cases the processor may have to switch through many spinning threads.

These example shows that the spinning threads negatively affect both the utilization and the runtime of the critical thread by consuming processor resources. As in the basic model, the runtime of a critical thread increases the most when latencies are short and there are many contexts.

3.2.6 Memory Bandwidth Effects

Memory bandwidth can limit the effectiveness of multiple-contexts, reducing processor utilization and increasing the runtime of the critical path. The previous cases do not consider the effects of limited memory bandwidth. All the memory requests are pipelined, and L was taken to be independent of the number of contexts. It is useful to consider the other extreme, in which all memory accesses are serialized. This case is shown in Figure 3.5. If $R + C < L$ the processor is communication bound and the multiple contexts will be ineffective in tolerating latency. All contexts will have outstanding references and each of these references has to wait until all the outstanding references that have been issued before it are handled. If on the other hand $R + C > L$ then the processor is computation bound and utilization will be good with just two threads. At most one context will have an outstanding reference at any given time. The processor utilization in each of these cases is:

$$U_{comm} = \frac{R}{L} \tag{3.11}$$

$$U_{comp} = \frac{R}{R + C} \tag{3.12}$$

It is worth noting that in the second case, having more than two contexts can still be useful in covering up transient variations in the run length R i.e. if there are sections of code in which $R + C < L$ so that the number of outstanding memory requests increases, followed by sections of code in which $R + C > L$ so that the number of outstanding memory requests decreases. The runtime of the critical thread in each of these scenarios becomes:

$$\begin{aligned} T_{c\_comm} &= ((I_c/R) - 1)PL + R \\ &\approx (I_c/R)PL \end{aligned} \tag{3.13}$$

$$\begin{aligned} T_{c\_comp} &= I_c + ((I_c/R) - 1)((P - 1)R + PC) \\ &\approx (I_c/R)P(R + C) \end{aligned} \tag{3.14}$$

Figure 3.6 shows U and T_c/T_{c1} for different numbers of contexts assuming the memory

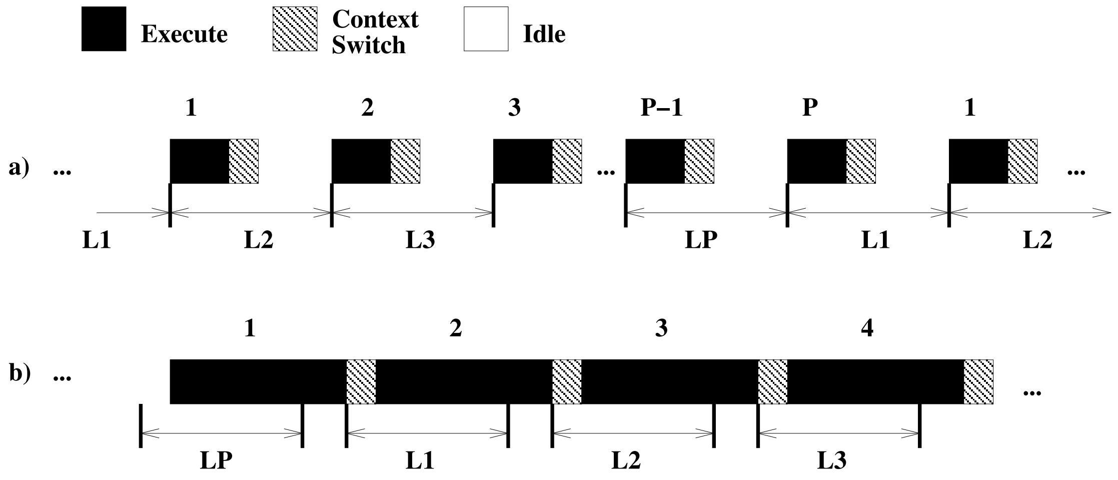


Figure 3.5: Multithreading in a single processor multiple-context system, assuming memory requests cannot be pipelined. a. Communication bound ($R + C < L$). b. Computation bound($R + C > L$).

requests cannot be pipelined. Maximum utilization is always reached with 2 contexts. When L=100 the processor is always communication bound and utilization is low. T_c increases as soon as there is more than 1 context per processor because either the thread is waiting for other threads' requests to be issued and satisfied, or it is waiting for other threads to finish executing.

This simple example shows that limiting memory bandwidth can seriously limit the effectiveness of multiple contexts to tolerate latency related to local misses in the cache, and can also cause a large increase in the runtime of a critical thread. In the case that memory requests cannot be pipelined to a memory module, multiple contexts will provide very limited latency tolerance for local accesses. Note however that the multiple contexts will still be useful in tolerating remote latencies since requests that have to go through the network will typically be pipelined.

Pipelined Memory Systems

The large negative effect memory bandwidth can have on processor utilization and critical thread execution time implies that in order to tolerate local misses using multiple contexts there must be sufficient local bandwidth. This in turn implies the use of more complicated memory systems, such as the use of interleaved memory banks in order to allow the pipelining of memory requests to a single memory module. Such a pipelined memory system is characterized not only by its latency L, but also by its throughput or bandwidth. The throughput determines how often the memory can accept a memory request. We define L_r

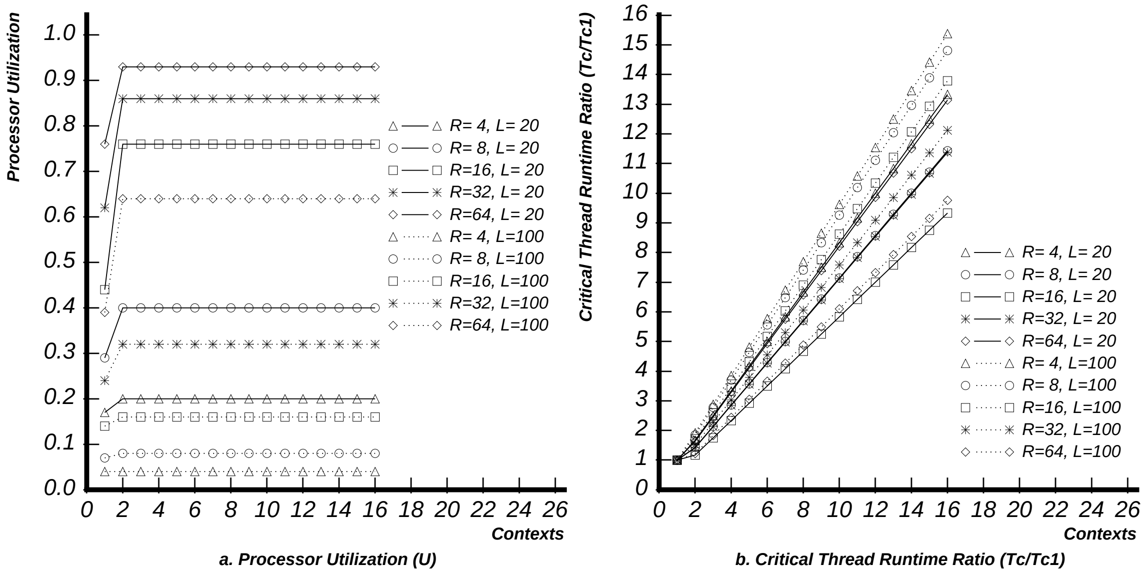


Figure 3.6: U and T_c/T_{c1} assuming references cannot be pipelined, for different values of R (4, 8, 16, 32, 64) and L (20, 100).

as the minimum number of cycles between memory requests. For instance, a 4-way interleaved memory might have a latency L of 20 cycles, and a throughput of 1 request every 5 cycles ($L_r = 5$). A non-pipelined system simply implies that $L_r = L$ and a fully pipelined system implies that $L_r = 1$.

Given a finite memory bandwidth, we find that an application will be communication limited if $R{+}C < L_r$ and will be computation limited if $R{+}C > L_r$. Figure 3.7a and 3.7b show each of these scenarios starting from an initial state in which there are no outstanding memory requests. It is obvious that the equations for the processor utilization and the runtime of a critical thread will be the same as equations 3.11 through 3.14 with L replaced by L_r. Even with a pipelined memory system the application can be limited by the memory bandwidth.

For the purposes of our simulations in later chapters, we will distinguish between the memory latency and the memory throughput or bandwidth, and we will explicitly take into account the serialization of memory requests at each node. Assuming that the throughput is sufficient, the memory latency tolerance properties of multiple contexts have a chance of being effective.

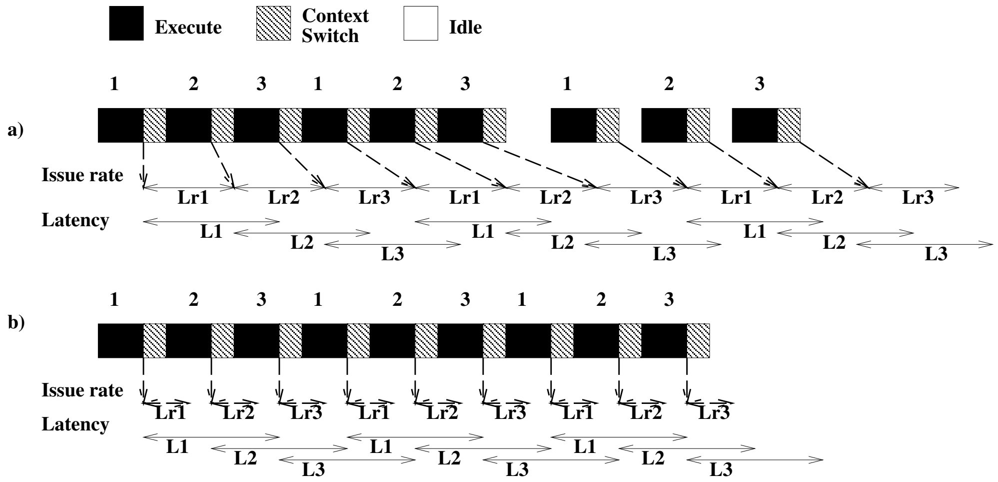


Figure 3.7: Multithreading in a single processor multiple-context system, assuming memory requests can be pipelined at a rate of one request every L_r cycles with a latency of L. a. Communication limited ($R + C < L_r$). b. Computation limited ($R + C > L_r$).

3.2.7 Network Bandwidth Effects

To first order, the effect of limited network bandwidth is the same as the effect of limited memory bandwidth. The network input and output port can only accept a message once every L_{r_net} cycles, and once a message is sent, it requires a latency of L_{net} for a response to come back. In practice, the situation is more complicated.

First of all, the latency through the network is dependent on the traffic in the network, and this depends on the activity of the other processing nodes. The next section discusses a model that takes the network into account when calculating the network latency. Second, there may be additional queuing delays at the remote processor's memory, especially if this memory is not highly pipelined. This component of the latency depends on how busy the remote memory module is. Third, the rate at which the network can accept messages or the minimum number of cycles between message sends L_{r_net} is variable because there can be contention for the outgoing network channel, that can delay the launching of a message.

Though the dependencies are more complicated, the implications are the same as for the memory bandwidth. If there is insufficient network bandwidth, both U and T_c suffer.

3.3 Network and Cache Effects

Section 3.2 looked at a first order model in which the parameters R and L were considered to be independent of the number of contexts P. In fact, the average run length R decreases with P, and the latency L increases with P. The run length R decreases because the cache miss rate increases when more contexts use the same cache. The latency L increases because with an increasing number of contexts the number of outstanding requests to the network increases, thus increasing the network load and latency. This section first considers these effects independently, and then incorporates them into the multiple-context model.

3.3.1 Network Model

We consider a packet-switched, direct interconnection network, of the k-ary n-cube class. A model for the latency associated with such a network was derived by Agarwal [1] and we use the model in the same way as he uses it to estimate the expected response time for a multiple context processor [2]. This model assumes cut-through, dimension ordered routing. It also assumes infinite buffering at the switch nodes, uniform traffic rates from all the nodes, and uniformly distributed and independent message destinations. This model gives the following expression for the average remote latency:

$$L = \left(1 + \frac{\rho B \frac{1}{k_d}\left(1 - \frac{1}{k_d}\right)}{(1-\rho)}\right) h + B + M - 1 \tag{3.15}$$

$$\rho = \frac{i_r B k_d}{2} \tag{3.16}$$

In these equations, M is the memory access time, B is the message size, h is the number of network switch delays and depends on n, the network dimension, and k, the network radix, ρ is the network channel utilization, k_d is the average distance traveled by a message in a given dimension, and i_r is the per node message injection rate. The delay L is the sum of the memory access time (M), the pipeline delay (B-1), and the h switch delays, $h/2$ hops for the request and reply respectively. The per hop contention delay is a function of the channel utilization ρ, the message length B, and the average distance in a dimension k_d[3]. The average number of hops k_d in any given dimension assuming bi-directional channels with no end-around connections can be shown to be $k_d = \frac{k-1/k}{3} \approx \frac{k}{3}$. The expression for the channel contention ρ assumes that there are separate channels in each direction, or $2n$ channels per switch.

[3] A slightly more accurate result includes an extra $(1 + 1/n)$ factor to the per hop queuing delay. The expression shown is the same approximation used in [2].

The channel utilization is a function of the injection rate i_r. The injection rate depends on the number of contexts and the regime of operation. Specifically, from Figure 3.1, i_r in the communication and computation limited regions is given by:

$$i_{r\_comm} = \frac{2P}{R+L} \tag{3.17}$$

$$i_{r\_comp} = \frac{2}{R+C} \tag{3.18}$$

The factor of two accounts for the fact that messages are generated both for requests and for responses. The network delay without contention is defined as $L_o = h + M + B - 1$. Using this expression, as well as $k_d = k/3$ and $h = 2nk_d$, we find two expressions for L depending on whether the thread is communication bound or computation bound:

$$L_{comm} = \frac{L_o}{2} + \frac{BPk}{6} - \frac{R}{2} + \frac{1}{2}\sqrt{\left(L_o - \frac{BPk}{3} + R\right)^2 + 8PB^2 n\frac{k}{3}\left(1 - \frac{3}{k}\right)} \tag{3.19}$$

$$L_{comp} = L_o + \frac{2nB^2(\frac{k}{3} - 1)}{C + R - B\frac{k}{3}} \tag{3.20}$$

The expected latency for the parameters of Table 3.2 and different values of R are shown in Figure 3.8. In the communication bound region, latency increases approximately linearly with the number of contexts, and becomes constant if and when the computation bound region is reached. As noted in other studies [46, 74], in the communication bound region with a finite number of outstanding transactions per processor, there is feedback between the network and the processor such that the message injection rate and latency do not become unbounded. The more outstanding references per processor there are, the higher the injection rate, the longer the latencies become, which in turn causes the message injection rate to decrease. Latency per hop does not become unbounded as in studies which decouple the injection rate from the network [22, 1], but rather reaches a steady state at which the injection rate is equal to the network service rate. Johnson [46] shows that the per hop latency tends to a limiting constant when the network becomes very large ($k_d \rightarrow \infty$). Nemawarkar et al. [74] shows that the rate at which each processor sends messages increases with P only as long as the network remains unsaturated.

Once the computation bound region is reached, the latency becomes constant since the injection rate is constant. In practice, increasing the number of contexts beyond the minimum required will cause R to decrease because of increased cache interference. This is discussed

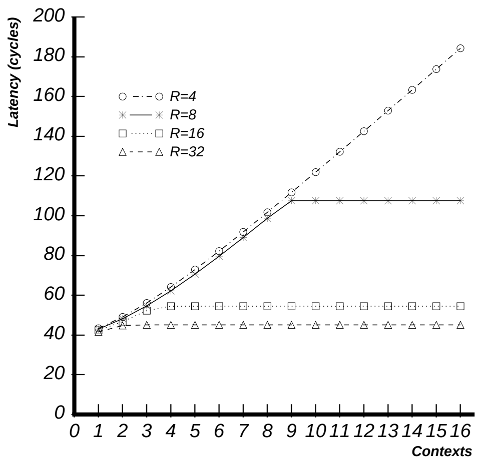


Figure 3.8: Predicted latency for different values of R (4, 8, 16, 32).

in more detail in the next section.

3.3.2 Cache Model

Intuitively, having multiple contexts will lead to decreased cache performance because the cache will contain the working sets of multiple threads at once. The miss rate for each thread will be larger than it would be if the thread were running with only its own working set in the cache, resulting in lower values of R, and higher values of (I_c/R). This means there will be more context switches and a longer critical path.

Determining an analytical model for how multiple contexts will affect cache performance is a tricky problem. Cache behavior is highly problem and machine dependent, and cache miss rates vary widely depending on the cache size, the working set size, and data reference pattern. A number of different models have been used to approximate these cache effects, and found to be useful in predicting the effect of cache behavior.

Saavedra-Barrera et al. [81] use a simple approximation which divides the cache evenly between the different contexts, and then uses results from the uniprocessor domain to estimate the effect of multiple contexts. Specifically, they note that the cache miss rate m can in general be approximated as $m = AS^{-K}$ where S is the cache size, and A and K are positive constants that depend on the workload. Using this relationship, they show that the miss rate for P contexts $m(P)$, and the corresponding run length R(P) can be expressed as:

$$m(P) = \begin{cases} m_1 P^K & if P \leq \left\lfloor m_1^{-1/K} \right\rfloor \\ 1 & if P > \left\lfloor m_1^{-1/K} \right\rfloor \end{cases} \tag{3.21}$$

$$R(P) = \begin{cases} R_1 P^{-K} & if P \leq \left\lfloor R_1^{1/K} \right\rfloor \\ 1 & if P > \left\lfloor R_1^{1/K} \right\rfloor \end{cases} \tag{3.22}$$

m_1 and R_1 are the miss rate and the run length for a single context. Typical values of K range from 0.2 to 0.5. Note that this model does not take into account data sharing between processors that can result in misses due to invalidations.

Agarwal [2] does a more detailed analysis, and considers a breakdown of the miss rate in terms of its different components. Specifically, the steady state miss rate consists of the *non-stationary* component due to misses that bring blocks into the cache for the first time, the *intrinsic* miss rate due to the size of the cache, the *extrinsic* interference due to the multi-threading, and the *invalidation* miss rate due to coherence related invalidations. Qualitatively, the model we use and Agarwal's models are similar. Both attempt to capture the cache effects based on the problem characteristics, specifically working set size, cache size, and data reference pattern. In the model we use, low values of K imply either a high fixed miss rate, or a small working set size in Agarwal's model, resulting in small change in the run length R with increasing P. Conversely, a large value of K implies a large intrinsic miss rate and a large working set size. In Agarwal's model, the effects of fixed miss rate are isolated from the effects of the intrinsic miss rate more clearly, rather than lumped together into a single parameter.

3.3.3 Complete Model

The complete model incorporates the network model and the cache model into the basic model of Section 3.2.4. Thus both the latency L and the runtime between context switches R are now functions of the number of contexts P. For simplicity we will use the simple cache model used by Saavedra-Barrera which gives the following expression for the runtime R:

$$R = R_1 P^{-K} \tag{3.23}$$

The calculation is communication bound when $(P-1)R_1P^{-K} + PC < L$, and computation bound when $(P-1)R_1P^{-K} + PC > L$. Note that L depends on P and the network as determined by the latency equations 3.19 and 3.20. Substituting equation 3.23 into equations 3.1 and 3.2 we find the following equations for the processor utilization:

Parameter	Description	Value
M	Memory latency	20 cycles
C	Context switch time	5 cycles
B	Message length	4
n	Network dimension	3
k	Network radix	8

Table 3.2: Baseline system parameters.

$$U_{comm} = \frac{PR_1P^{-K}}{R_1P^{-K} + L_{comm}} \tag{3.24}$$

$$U_{comp} = \frac{R_1P^{-K}}{R_1P^{-K} + C} \tag{3.25}$$

Similarly, substituting into equations 3.4 and 3.6 we find the following equations corresponding to the communication bound and the computation bound cases respectively:

$$T_{c\_comm} \approx (I_c/(RP^{-K}))(R_1P^{-K} + L_{comm}) \tag{3.26}$$

$$T_{c\_comp} \approx (I_c/(RP^{-K}))P(R_1P^{-K} + C) \tag{3.27}$$

3.3.4 Discussion

Having developed a simple model for the effects of multiple contexts on both the processor utilization and the critical path, we now look at the implications of this model. For this purpose, we will use the parameters shown in Table 3.2.

Region of Operation

The first question of interest is whether the system is working in the communication or the computation bound region. As we will show in Section 3.4, only in the computation bound region are we able to influence the execution time of a critical thread. Figure 3.9 shows curves that correspond to the work available for overlap minus the latency $(P-1)R+PC-L$. Whenever the latency is smaller than the work available for overlap, the processor is in the computation bound region, otherwise it is in the communication bound region. The

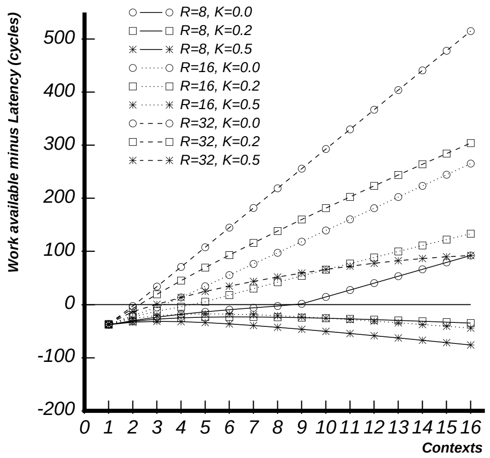


Figure 3.9: Region of operation for different values of R (8, 16, 32) and K (0.0, 0.2, 0.5). The curves plot $(P-1)R+PC-L$ which is just the work available to overlap with latency, minus the latency. The processor is computation bound when the curve is above 0 and communication bound when the curve is below 0.

curves are shown for different single processor run lengths, and different values of the cache degradation index K.

For low values of R, the processor may never reach the computation bound region of operation due to the extra network traffic causing latency to increase at a faster rate than the extra work available to each processor. The lower bound on R for which the computation bound region is reached increases as the cache degradation factor increases. Thus when K = 0.2, the R = 8 curve no longer reaches the computation bound region, and when K = 0.5, neither does the R = 16 case. Increasing K both reduces the average run length R, and increases the network latency.

Processor Utilization

Figure 3.10a shows the processor utilization for different values of R and K. These curves are much the same as the results found in [2, 81]. Initially utilization improves almost linearly with the number of contexts, until it becomes computation bound at which point adding more contexts decreases the utilization because of decreased cache and network performance[4]. It should be noted that except in the case that cache effects are very high (K = 0.5), the processor utilization remains quite close to its maximum value even when

[4] Saavedra-Barrera et.al. [81] also identify the *transition region* where utilization increases, but at a less than linear rate. This transition region is caused by variations in the run length R. In the transition region the processor is sometimes compute bound if there is a series of long run lengths, and sometimes latency

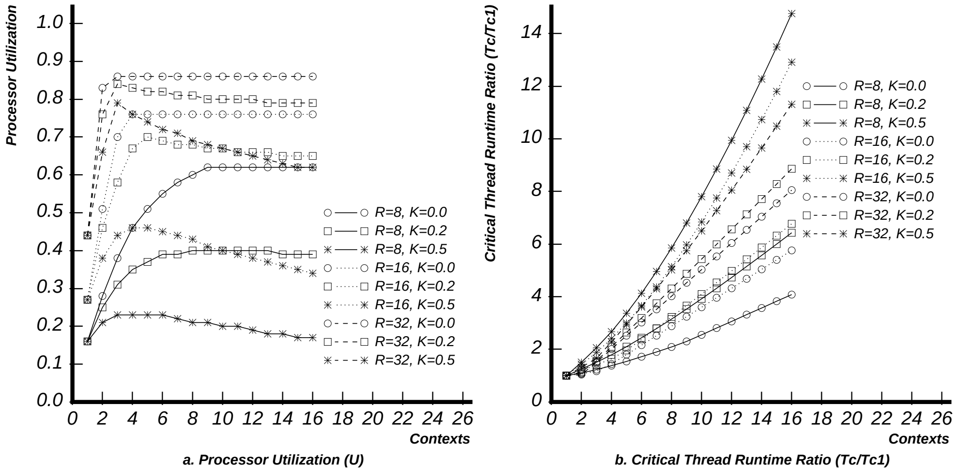


Figure 3.10: U and T_c/T_{c1} for different values of R (8, 16, 32) and K (0.0, 0.2, 0.5).

there are more contexts than required. This is because in the computation bound case, the message injection rate remains relatively constant, and increases only slightly due to cache performance degradation.

Critical Thread Runtime

Figure 3.18b shows the increase in the runtime of a critical thread with multiple contexts. The runtime of a critical thread now increases more than just linearly with the number of contexts as it did in the basic model. The decreased cache performance causes the critical thread to cache miss more often, and the increased number of contexts causes the network latency to increase.

3.3.5 Cache and Network Effects with Spin-Waiting and with Limited Bandwidth

The cache and network effects can be incorporated into the spin-waiting models and bandwidth limited models, with similar implications as when it is incorporated into the basic model.

When there are threads spin-waiting, to first order the spinning threads do not generate any

limited if there is a series of short run lengths. Adding more contexts causes the processor to be more and more in the computation bound region. With sufficient contexts it will be in this region with very high probability.

network traffic since they repeatedly hit in the cache. Also, aside from the synchronization variable the spinning threads' working sets do not have to be in the cache. As a result, the cache effects and the network effects will not be as significant as when all threads are executing. With limited bandwidth, there is an additional constraint on the message injection rate: the injection rate cannot exceed the maximum bandwidth of the memory and network systems.

3.4 Thread Prioritization in the Multithreaded Model

In this section, we examine the effect of thread prioritization on the multithreading models discussed in Sections 3.2 and 3.3. For the simple model, thread prioritization improves the runtime of the critical thread by allowing it to proceed as soon as it is able, rather than executing other ready threads. If spin-waiting is used, thread prioritization improves both processor utilization and the runtime of a critical thread by avoiding unnecessarily switching to threads when they are spinning. If the application is either memory or network bandwidth limited, prioritizing the use of the available bandwidth also improves the runtime of the critical thread.

The complete model shows that provided the processor reaches the computation limited region, prioritizing threads appropriately can improve both processor utilization, and the runtime of the critical thread. We can prioritize threads so as to avoid the negative cache effects of having more contexts than needed to reach the computation limited region, thus improving utilization. If a critical thread is given the highest priority, it will execute whenever it is able, and thus its runtime will improve.

3.4.1 Prioritizing Threads in the Basic Model

Consider the basic model of section 3.2.4. In the communication bound case, there is little that can be done about reducing the critical path. In the computation bound case however, the critical path is lengthened by the fact that there are more threads than necessary to tolerate latency. If we are not restricted to round-robin scheduling, then the critical thread could begin executing as soon as its memory request was satisfied. This is shown in Figure 3.11. In Figure 3.11a we assume that scheduling is *non-preemptive* so that the critical thread only resumes execution on the next context switch. In Figure 3.11b we assume that scheduling is *preemptive* so that the currently executing thread is interrupted as soon as the critical thread memory reference is satisfied. The expressions for the processor utilization remain essentially the same as equations 3.1 and 3.2. Similarly, the expression for the communication bound case, T_{c_comm}, is still given by equation 3.4. However, the runtime for the critical thread in the compute bound case changes to:

$$\begin{aligned} T_{c\_comp} &\approx I_c + ((I_c/R) - 1)(L + R/2 + C) \text{ with nonpreemptive scheduling} \\ &\approx (I_c/R)(3R/2 + L + C) \end{aligned} \tag{3.28}$$

$$\begin{aligned} T_{c\_comp} &= I_c + ((I_c/R) - 1)(L + C) \text{ with preemptive scheduling} \\ &\approx (I_c/R)(R + L + C) \end{aligned} \tag{3.29}$$

In the first case[5], the critical path is increased by an average amount of $(I_c/R)(R/2+C)$ and in the second case, $(I_c/R)C$. With the prioritized scheduling the increase in the runtime of the critical thread no longer depends on the number of contexts executing as it does with round-robin scheduling, but only on R and C. Note that the decision of whether it is better to context switch immediately or whether it is better to wait until the currently executing thread misses in the cache involves a tradeoff between processor utilization and critical path length: switching immediately decreases the critical path length, but incurs the cost of a premature and unnecessary context switch, whereas waiting until the next cache miss increases the critical path length slightly, but does not incur the premature context switch. Figure 3.12 shows T_c/T_{c1} when threads are prioritized and when they are unprioritized assuming preemptive scheduling. T_c/T_{c1} increases slightly when the processor becomes computation bound, but does not increase linearly with P when prioritized.

3.4.2 Prioritizing Threads for Spin-Waiting Threads

Giving a critical thread high priority can also prevent it from being needlessly delayed by spinning threads. This is illustrated in Figure 3.13, assuming preemptive scheduling with an immediate context switch upon completion of the long latency reference. When threads are prioritized the definition of when the processor is communication bound and when it is computation bound changes because now the processor no longer needs to execute any of the spinning threads before resuming execution of a critical thread. The processor is communication bound when there are not enough non-spinning threads to keep the processor busy $((P - P_s - 1)R + (P - P_s)C < L)$, and it is computation bound when there are enough $((P - P_s - 1)R + (P - P_s)C > L)$. The resulting equations for the processor utilization and the critical path assuming preemptive scheduling are:

[5] This expression is approximate, and valid when $C << R$. A more exact expression assumes that the critical reference can become satisfied either during a context switch or while a thread is executing, and that if it happens during a context switch, another context switch will be required in order to start executing the critical thread. The expression in this case is $(I_c/R)\left(R + L + \frac{R}{R+C}(R/2 + C) + \frac{C}{R+C}(R + 3C/2)\right)$.

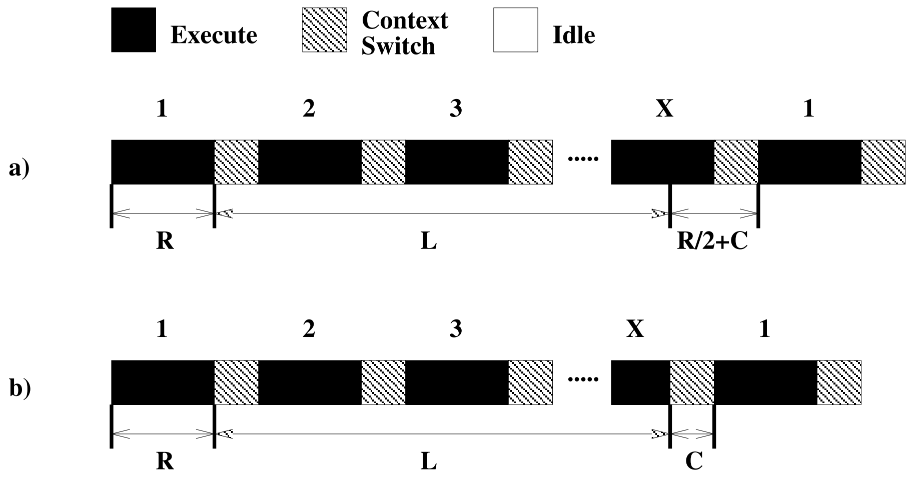


Figure 3.11: Multithreading with thread prioritization in the computation bound case. Thread 1 is the critical thread. a. Non-preemptive scheduling. b. Preemptive scheduling.

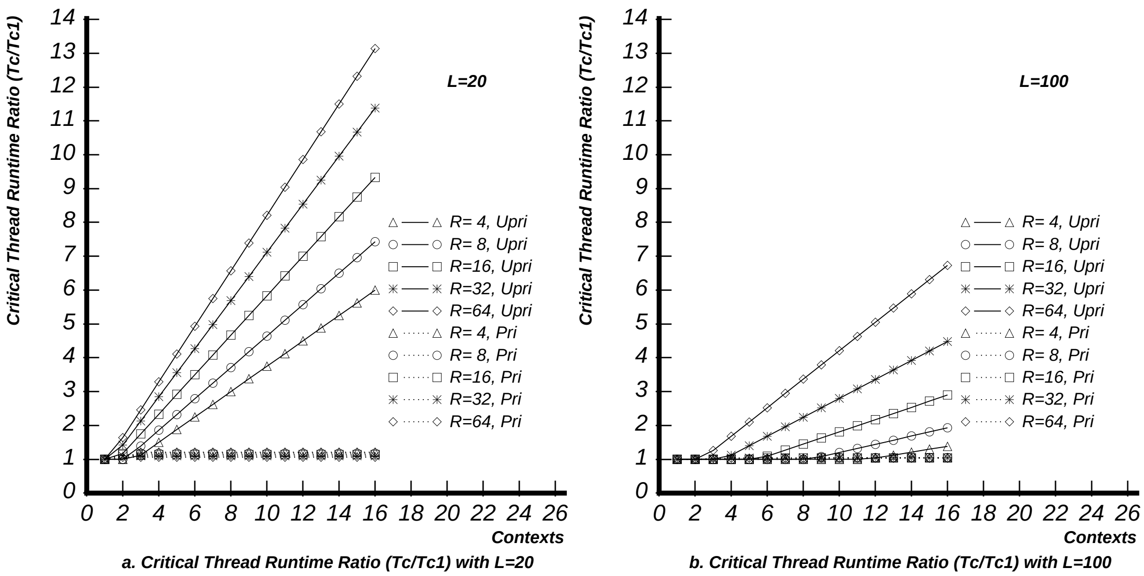


Figure 3.12: Comparison of T_c/T_{c1} with prioritized (Pri) and unprioritized (Upri) scheduling for different values of R (4, 8, 16, 32) and L (20, 100).

$$U_{comm} = \frac{(P - P_s)R}{R + L + C} \tag{3.30}$$

$$U_{comp} = \frac{R}{R + C} \tag{3.31}$$

$$\begin{aligned} T_{c\_comm} &= I_c + ((I_c/R) - 1)(L + C) \\ &\approx (I_c/R)(R + L + C) \end{aligned} \tag{3.32}$$

$$\begin{aligned} T_{c\_comp} &= I_c + ((I_c/R) - 1)(L + C) \\ &\approx (I_c/R)(R + L + C) \end{aligned} \tag{3.33}$$

Figure 3.13 makes the assumption that all threads that are not spinning have higher priority than threads that are. This clearly cannot be the case indefinitely as the spinning threads would never execute. The issue of how to prioritize threads in the context of this type of synchronization depends on the application and we will look at a number of different scenarios in later chapters.

Figures 3.14 and 3.15 plot U and T_c/T_{c1} respectively, when threads are prioritized and when they are unprioritized. When threads are prioritized, utilization does not fall off due to spinning threads wasting cycles, but rather falls off only once the processor leaves the computation bound region. T_c/T_{c1} increases slightly even when threads are prioritized due to the overhead of switching to the critical thread once its long latency reference is satisfied, but does not increase linearly with the number of contexts as when threads are not prioritized.

3.4.3 Prioritizing Bandwidth Utilization

In the case that memory or network bandwidth is the limiting factor, then prioritizing the bandwidth utilization also helps the critical path. Figure 3.16 shows how in a bandwidth limited situation the prioritization allows the most critical thread to proceed before less critical threads. In this example thread 1 is critical and this thread's critical reference will proceed as soon as the desired resource is available. Thread 2 and thread 3 have the same priority and so they share equally whatever bandwidth remains. Note that though the processor utilization remains unchanged, the runtime of the critical thread suffers much less. Thus when $L_r \geq R + C$, the expected value of the critical thread runtime given in

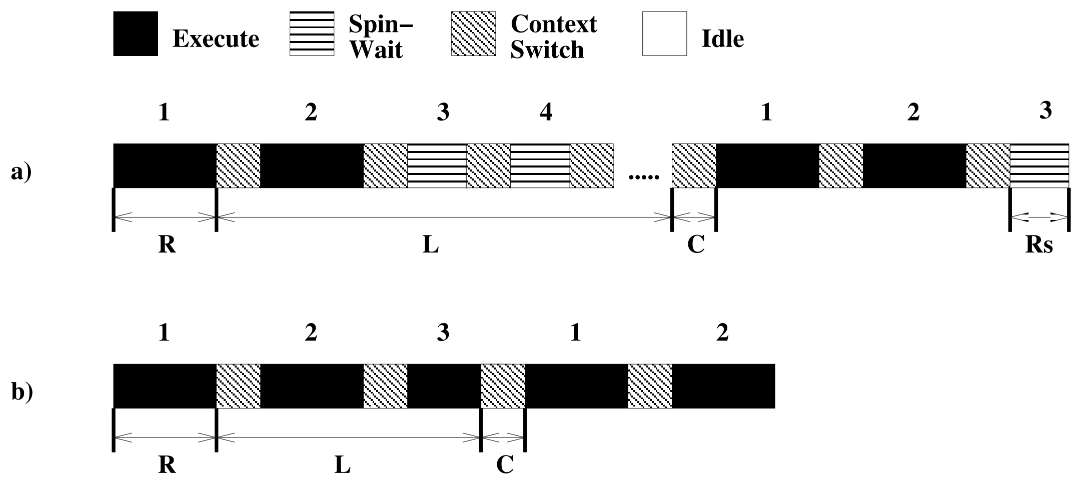
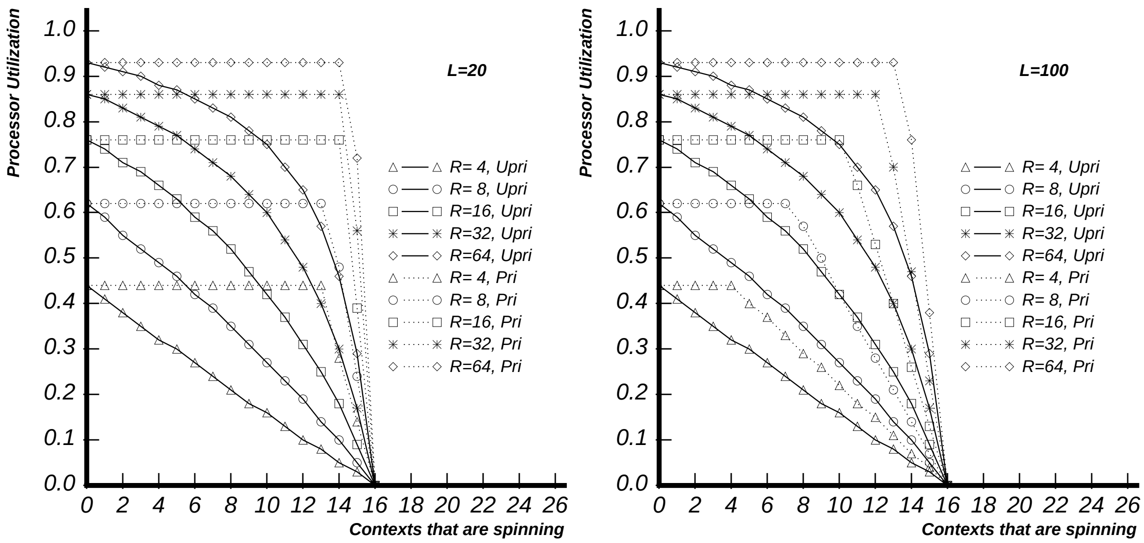


Figure 3.13: Multithreading with thread prioritization assuming some threads are spin-waiting. Thread 1 is the critical thread and preemptive scheduling is used. a. Communication limited ($(P - P_s - 1)R + (P - P_s)C < L$). b. Computation limited ($(P - P_s - 1)R + (P - P_s)C > L$).



a. Processor Utilization (U) with L=20 b. Processor Utilization (U) with L=100

Figure 3.14: Comparison of U with prioritized (Pri) and unprioritized (Upri) scheduling when threads are spin-waiting, for different values of R (4, 8, 16, 32, 64) and L (20, 100). Assumes that there are 16 threads running and that an increasing number of these threads are spin-waiting.

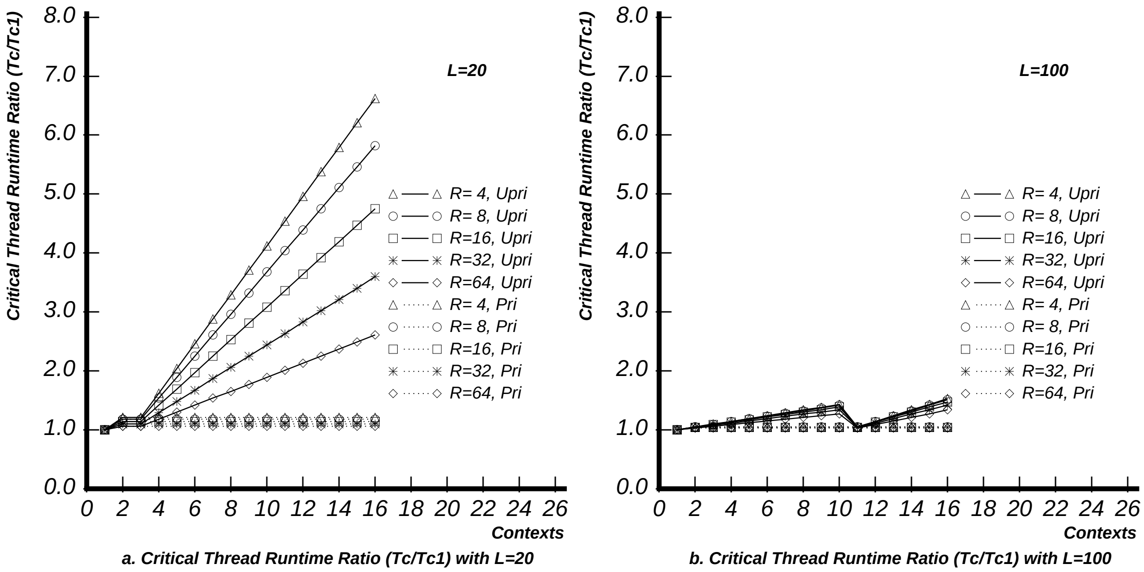


Figure 3.15: Comparison of T_c/T_{c1} with prioritized (Pri) and unprioritized (Upri) scheduling when threads are spin-waiting, for different values of R (4, 8, 16, 32, 64) and L (20, 100). Assumes that only one thread is running and all the other threads are spin-waiting.

equation 3.13 becomes:

$$\begin{aligned} T_c &\approx I_c + ((I_c/R) - 1)(L + L_r/2) \\ &\approx (I_c/R)(R + L + L_r/2) \end{aligned} \tag{3.34}$$

This assumes that the critical thread has to wait $L_r/2$ cycles before it can issue its memory request. Although the runtime still suffers because the thread may have to wait until the resource is next available, it no longer has to wait until all previous transactions are processed. T_c/T_{c1} is shown in Figure 3.17 for the case that $L = L_r$.

3.4.4 Prioritizing Threads in the Complete Model

The effect of thread prioritization in the model that takes into account network and cache effects depends on the exact details of how threads are prioritized. For instance, if only a single thread has a high priority, and all the other threads have the same lower priority, then as in the basic model the only thing that changes is the expression for the critical thread runtime in the computation bound case. Specifically, equation 3.27 becomes one of

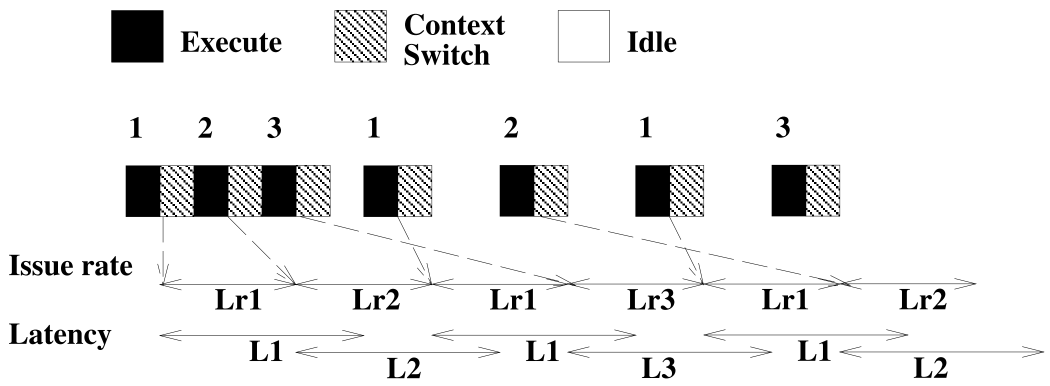


Figure 3.16: Multithreading with prioritization assuming a bandwidth limited application ($L_r > R + C$). Thread 1 is the critical thread.

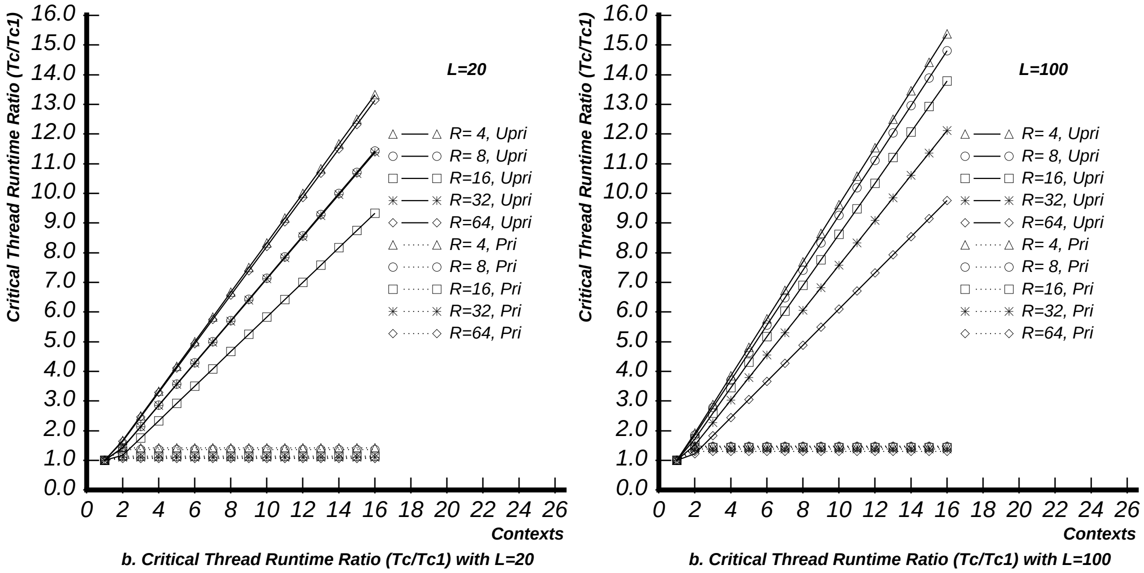


Figure 3.17: Comparison of T_c/T_{c1} with prioritized (Pri) and unprioritized (Upri) scheduling assuming references cannot be pipelined, for different values of R (4, 8, 16, 32, 64) and L (20, 50).

the following depending on whether the scheduling is preemptive or not:

$$T_{c\_comp} \approx (I_c/(RP^{-K}))(3R_1P^{-K}/2 + L_{comp} + C) \text{ with nonpreemptive scheduling} \quad (3.35)$$

$$T_{c\_comp} \approx (I_c/(RP^{-K}))(R_1P^{-K} + L_{comp} + C) \text{ with preemptive scheduling} \quad (3.36)$$

Thus although the critical thread runtime is much better than it was without prioritization, it is still affected by the number of messages in the network and by the negative cache interference that occurs among all the contexts. By doing more detailed prioritization, it is possible to minimize this effect by adaptively adjusting the number of contexts that are running to the minimum necessary to tolerate latency. This is discussed in the next section.

Effect of Prioritization on Processor Utilization

In Figure 3.10 we saw that processor utilization suffers even when we are in the computation bound region, because of negative cache effects that occur with an increasing number of contexts. These negative cache effects cause a decrease in the runtime between context switches which in turn incurs more context switch overhead penalties, and causes increased network traffic and latency. Ideally, we want to minimize the contexts that are actually running instructions to the number required to achieve the maximum utilization. Provided the maximum utilization occurs when the processor is in the computation bound region, we can prevent the degradation of processor utilization by prioritizing the threads appropriately.

To stabilize the utilization, each thread is given a unique priority. If there are P contexts available, but only P_n of them are necessary to reach the computation bound region, then only the P_n highest priority threads will be executing. This is because on a context switch it is always the highest priority unblocked thread that executes. Thus cache performance improves because less than the total number of threads are issuing instructions. According to our simple cache model the miss rate will be $m = m_1P_n^{-K}$ rather than $m = m_1P^{-K}$. This prioritization of threads dynamically restricts the number of threads that are executing to the minimum required to tolerate the observed latency. Note that when the number of contexts actively executing instructions is limited to the minimum required, the average latency L also decreases, since the run length R is longer and there is less network traffic.

The effect on processor utilization is shown in Figure 3.18. Provided the processor reaches the computation bound region, the processor utilization stabilizes and remains constant even with an increased number of contexts. If the thread does not reach the computation bound region, then the prioritization will not prevent the utilization from falling off with increasing contexts. In the case that we are communication limited it may be necessary to

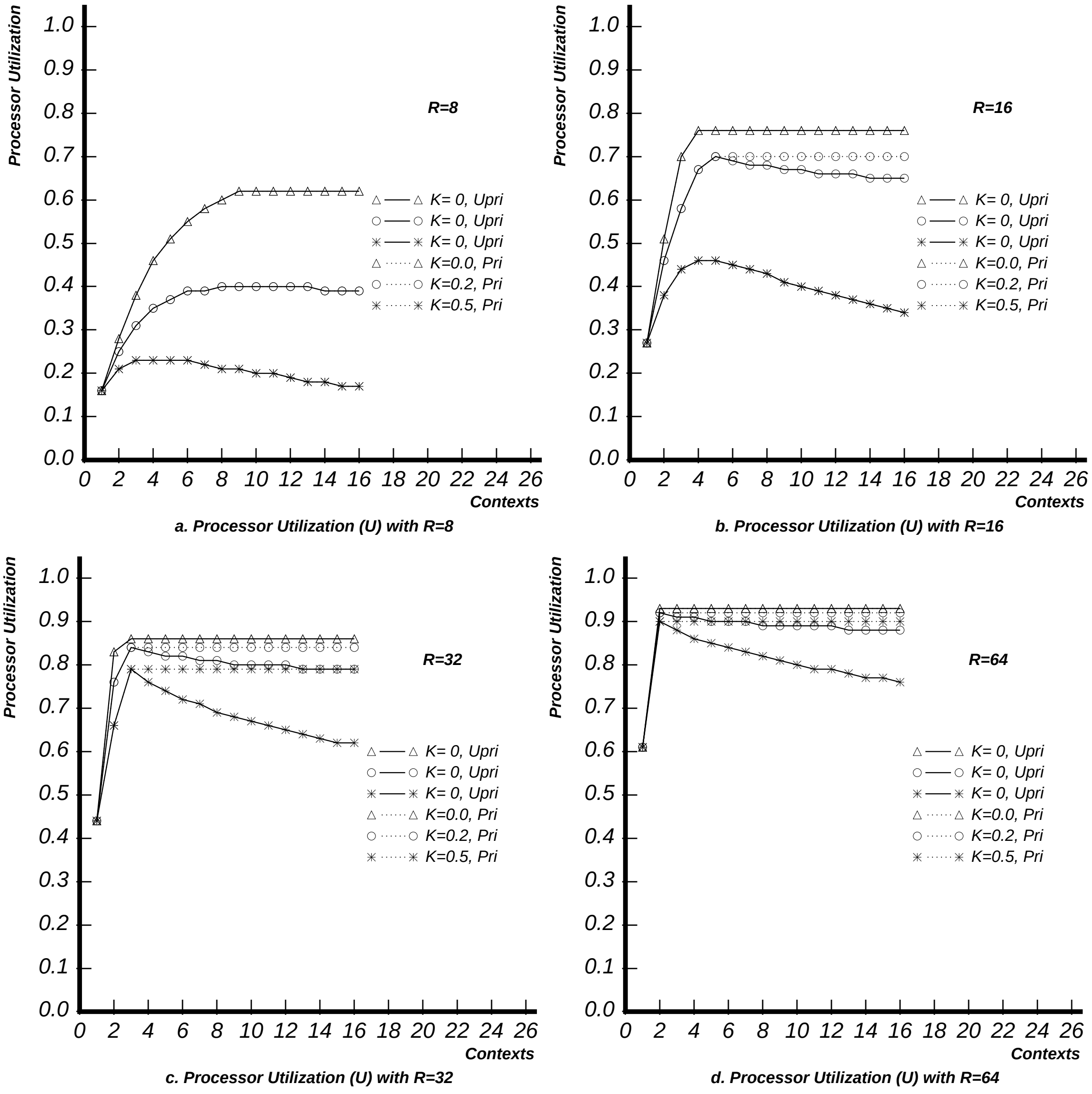


Figure 3.18: Comparison of U with prioritized (Pri) and unprioritized (Upri) scheduling when loaded threads are uniquely prioritized, for different values of R (8, 16, 32, 64) and K (0.0, 0.2, 0.5).

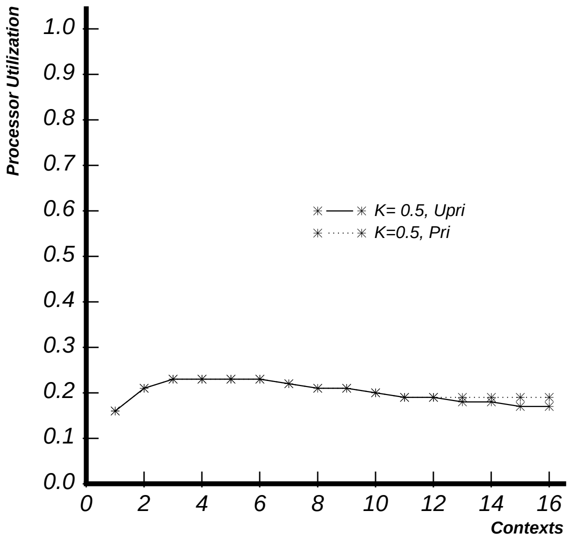


Figure 3.19: Utilization with R=8, K=0.5, and C=10. The peak utilization occurs during the communication limited region.

statically limit the number of contexts that are executing instructions in order to achieve the maximum utilization.

The utilization achieved by prioritizing threads in the computation limited region is not necessarily the maximum achievable utilization rate. Consider for instance the case that 2 contexts executing are just 1 cycle short of being in the computation limited region. Adding an additional context may put the processor into the computation limited regime, but may decrease the run length R of all the threads by more than 1 cycle. If the context switch overhead C were 0, then this still improves utilization to be 100%, but if C is not 0, the utilization will fall off. In general it is possible for the processor utilization to achieve its peak in the communication limited region, and then have lower utilization when in the computation bound region. An example of this is shown in Figure 3.19. The processor only reaches the computation limited region with 11 contexts, and is in the communication limited region before then. Reducing C, increasing R, and decreasing K will reduce this effect since run lengths will suffer less with additional contexts, and C will present a smaller overhead.

3.4.5 Effect of Prioritization on the Critical Thread Runtime

Figure 3.20 shows the effect on the critical thread runtime when context prioritization is used. Only one thread is given high priority and all others are given equal priority.

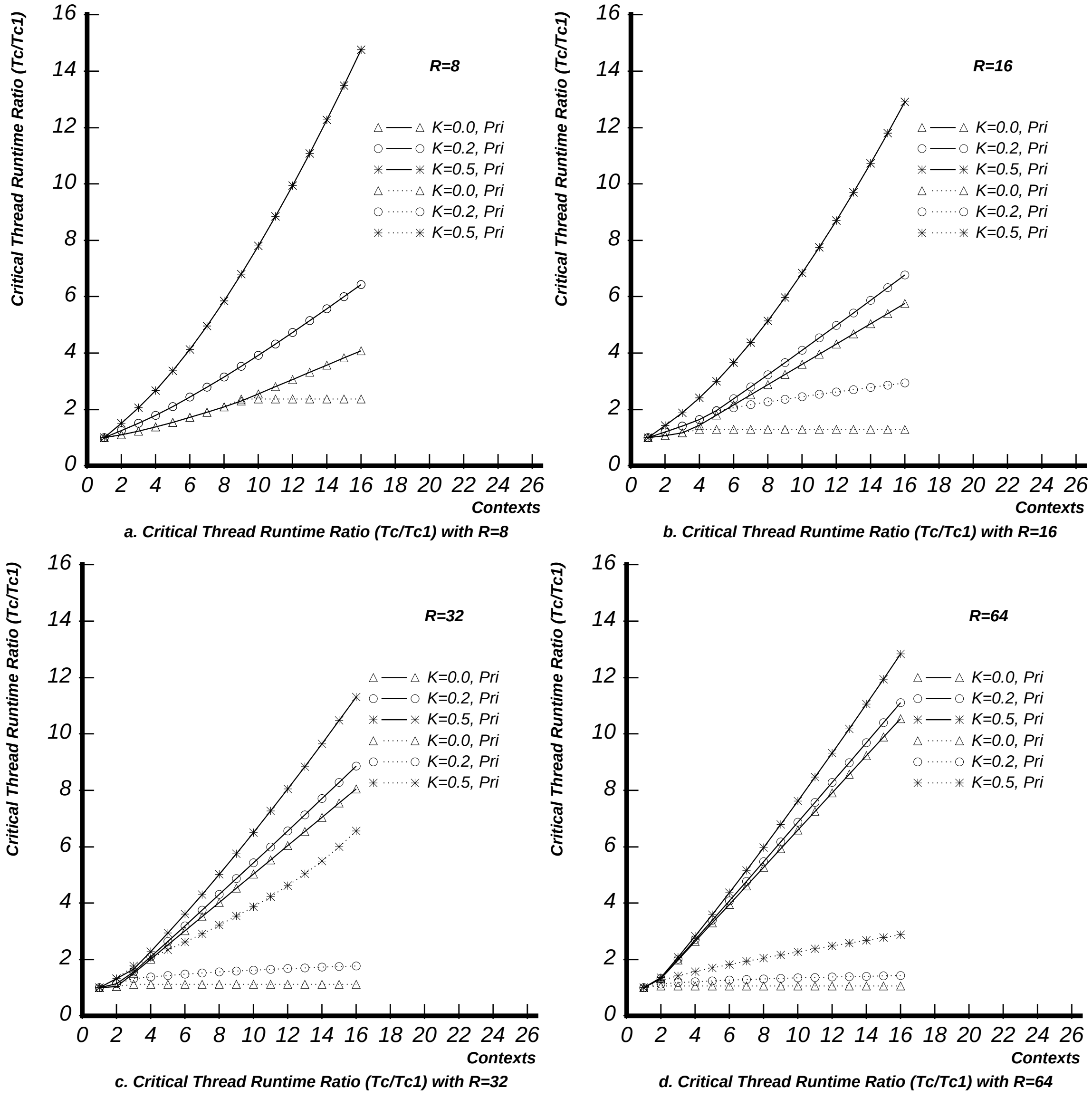


Figure 3.20: Comparison of T_c/T_{c1} with prioritized (Pri) and unprioritized (Upri) scheduling when the critical thread is given high priority and all other threads equal priority, for different values of R (8, 16, 32, 64) and K (0.0, 0.2, 0.5).

All the cases in which the threads are computation bound benefit substantially from the prioritization, whereas those that are communication bound do not.

3.5 Limits of the Model

The model described in this chapter is meant primarily to provide intuition about the possible effects of thread prioritization, and makes a number of idealized assumptions. These include:

1. Uniform run length R: In actuality the run length R is not a constant, and will vary from a minimum value of 1 cycle to some maximum that is determined by the maximum number of cycles that one context can execute before releasing the pipeline. Furthermore, some studies have shown that the effectiveness of multithreading in tolerating latency is negatively affected by clustered loads in which a rapid sequence of closely spaced cache misses leads to very short run lengths, so many contexts are stalled waiting for remote references. The practical consequence of this is that more contexts are necessary to effectively tolerate latency. This situation is somewhat alleviated by allowing several memory references from a single context to be outstanding at once because threads can run longer before context switching [13].

2. Divided cache assumption: The model assumes that the cache is divided evenly between all the available contexts, and does not consider the effects of the threads sharing a cache. Negative cache effects are caused by the different contexts invalidating each other's data in the cache. Positive cache effects are caused by threads having overlapping working sets so that some data in the cache is used by several threads at once. If these effects occur, they tend to change the K parameter in the cache model, thus affecting the run length R. If there are positive cache effects, the average latency can be reduced if threads prefetch data for each other into the cache.

3. Idealized working set behavior: We assume that the cache miss rate is governed by the relationship $m = AS^K$. Recent work by Rothberg, Singh, and Gupta [80] that studies the cache performance based on working sets suggest that the performance of the cache has a more step-like behavior. Specifically, for each thread there is a hierarchy of working sets, and each time the cache becomes large enough to contain another level of the hierarchy, cache performance increases in a step like fashion. For instance if the number of contexts increases beyond the point where their first level working sets all fit into the cache, the drop in cache performance can be substantially larger than that suggested by the K parameter.

4. Uniform remote latencies: In the model all latencies are assumed to be equal. In fact, latencies will vary from reference to reference based on locality, and on the type of reference. For instance, a write operation may require invalidations to be sent and acknowledged, adding substantially to the latency of the operation. Again this implies

that the number of contexts required to tolerate latency will be different at different points in the computation.

5. Constant number of messages per transaction: It was assumed that only a request and reply message are required for each cache miss. Memory writes may require data to be invalidated, thus requiring more than two messages per transaction. When more than two messages are required the base latency increases as does the network traffic.

6. Uniform traffic: Real applications will not generate uniform traffic. This can lead to two effects. First, there can be hot spots in the network, as well hot nodes that are getting an undue portion of the memory references. This will tend to increase the latency of remote references. Second, by exploiting locality the latency of remote memory references can be reduced by reducing the number of hops that each message must make. This reduces both the contention free latency L_0 and the channel utilization, and thus the contention delay. By reducing the latency, fewer contexts are required to tolerate latency, and both the processor utilization and the effect of multiple contexts on the critical path will be reduced.

The main effect of incorporating more realistic assumptions into the model is to cause changes in the basic parameters associated with the model: the number of contexts required to reach the computation bound region, the cache miss rate, and the latency. Some effects cannot be accounted for in the model. For instance the possibility of step-like behavior in the cache performance is best examined by simulation. Also, hot spots are highly application dependent and usually have to be eliminated in the program in order to achieve good performance.

3.6 Conclusions

The general conclusions to be drawn from the model described in this chapter are the following:

1. When the processor is computation bound, that is, there are more than enough contexts to fully tolerate latency given the average run length R, then prioritization can substantially improve the runtime of a critical thread. It can also improve processor utilization by dynamically restricting the number of contexts that are actually executing to the minimum required to fully tolerate latency.

2. When the processor is communication bound, prioritization is ineffective in reducing the run length of the critical path execution. In this case, the only way of reducing it is to reduce the latency L, and increase the average run length R. The only ways of doing this are to restrict the number of contexts, and to promote data sharing between running threads so as to improve both cache and network performance.

3. When there are spin-waiting threads, thread prioritization can be used to minimize the cycles consumed by spinning threads.

4. When the application is bandwidth limited, thread prioritization can be used to prioritize the use of bandwidth and substantially improve the runtime of a critical thread by giving it priority to the available bandwidth. It can dynamically restrict the number of contexts that are actually executing to the minimum number required to fully utilize the bandwidth.

The behavior of real programs will be examined in more detail in the following chapters where a number of benchmarks are run in a simulation environment. These experiments confirm the general trends predicted by the model.

Chapter 4

Implementation

This chapter explores the different software and hardware alternatives that are available for implementing the thread prioritization mechanisms. We look at the performance and cost of these alternatives to justify the range of cost assumptions made in the following chapters. We conclude that for loaded threads, prioritizing in hardware is best because it allows extremely fast context selection on a context switch. For unloaded threads, a software queuing structure is appropriate, since the frequency of thread swapping is much lower than the frequency of context switching.

Sections 4.1 and 4.2 examine implementation issues related to loaded thread prioritization, which include fast implementations of context prioritization, thread stalling based on data availability, memory request prioritization, and preemptive scheduling. How we implement context prioritization is crucial because it affects performance on every context switch and on every change of priority. In the case of frequent context switches or frequent changes in a thread's priority, both these operations must be extremely efficient. We show that a simple hardware implementation can minimize the context selection time and the priority change time, while software implementations can choose the next context efficiently but make changing a thread's priority costly.

The memory system must also participate in the thread prioritization for it to be effective. Thread stalling means stalling a thread while it is waiting for data to be returned. Thread stalling is crucial to multiple-context scheduling because it allows other contexts to use the pipeline. An alternative to stalling a thread is to have it poll regularly for the data, but in a multiple context system this leads to contexts consuming cycles doing polling operations when other contexts could be doing useful operations. To prioritize memory requests, transaction buffers are used. Transaction buffers store and track the multiple outstanding references coming from the different contexts. When several memory requests that are waiting in transaction buffers require the same resource, the highest priority request is issued.

Finally, we must decide whether scheduling is going to be preemptive or not. Preemptive scheduling involves forcing a context switch to a higher priority thread if it becomes ready to execute while a lower priority thread is executing. The performance impact of allowing preemption depends on the cost of context switching. If the cost is low, forcing an extra context switch may be better, whereas if the cost is high, it may be better to wait until the executing thread wants to context switch.

Section 4.3 examines unloaded thread prioritization. Unloaded thread prioritization is done by software and consequently has considerable flexibility. However, it can benefit from a mechanism to preempt a lower priority loaded thread when a higher priority unloaded thread becomes available.

4.1 Context Prioritization

Context prioritization has two costs: the cost of selecting the next context to run on a hardware context switch or the *context selection cost*, and the cost of changing the priority of a loaded thread or the *priority change cost*.

The context selection cost is added to the hardware context switch time to determine the total context switch time. Context switches are frequent, so this cost must be low. The impact of the priority change cost depends on how frequently thread priorities change. In the case that it occurs frequently, the cost must be low, but if priorities only change infrequently, then a higher cost is acceptable.

A number of alternatives for implementing both context selection and priority changing are possible. The alternatives range from very high performance, hardware intensive solutions, to lower performance and lower cost software solutions. These options are described below.

4.1.1 Hardware

In this implementation, each context has a priority register which contains the priority of the loaded thread, and these values feed into a combinational comparator circuit that selects the next context to schedule. A diagram of this logic is shown in figure 4.1. The comparator logic takes as inputs the priorities of all the contexts, and for each context outputs a bit that indicates whether the context has one of the highest priority loaded threads (implements a MAX function). The round-robin selection logic takes these outputs as inputs and schedules all the high priority threads in round-robin fashion.

A one bit slice of a 4-input MAX circuit is shown in figure 4.2. For an N-bit priority, N slices are needed to implement the complete comparison. A MAX circuit for a 4-bit priority is shown in figure 4.3. An output of the least significant bit is a 1 if the given context contains

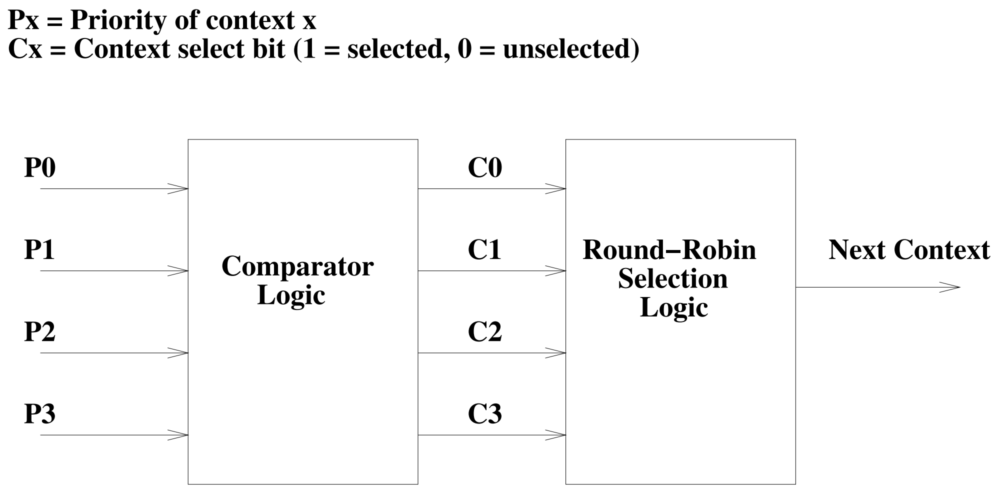


Figure 4.1: Logic for selecting the next context on a context switch. The comparator logic chooses all threads with the highest priority, and the round-robin selection logic chooses among the highest priority threads.

a thread of the highest priority. The hardware cost of an individual bitslice grows as $O(C)$, where C is the number of contexts. Thus the size for the complete N-bit comparator is $O(NC)$. If a context is not loaded with a thread, the carry-in of the highest priority bit for that context can be set to 0, thus guaranteeing that it will have lower priority than all other contexts that are loaded.

For large N, this circuit will be slow due to the long carry-chain dependence and will require several processor cycles to evaluate. An alternative is to use a tree of two-way comparators rather than the single N-way comparator. The advantage of this is that the this time can easily be reduced to time that is proportional to $log(N)log(C)$ using carry-select techniques [107] in each two-way comparator, and allowing comparisons to proceed in parallel. This circuit is described in Appendix A.

Using this implementation it takes zero cycles to choose the next thread to execute on a context switch since the circuit continuously outputs the next choice. A single cycle is needed to change the priority of a loaded thread. The principal cost of this hardware implementation is the cost of the priority registers, one 64-bit register for each context, and the cost of the comparator. If we require fewer priority levels, then we can use smaller registers.

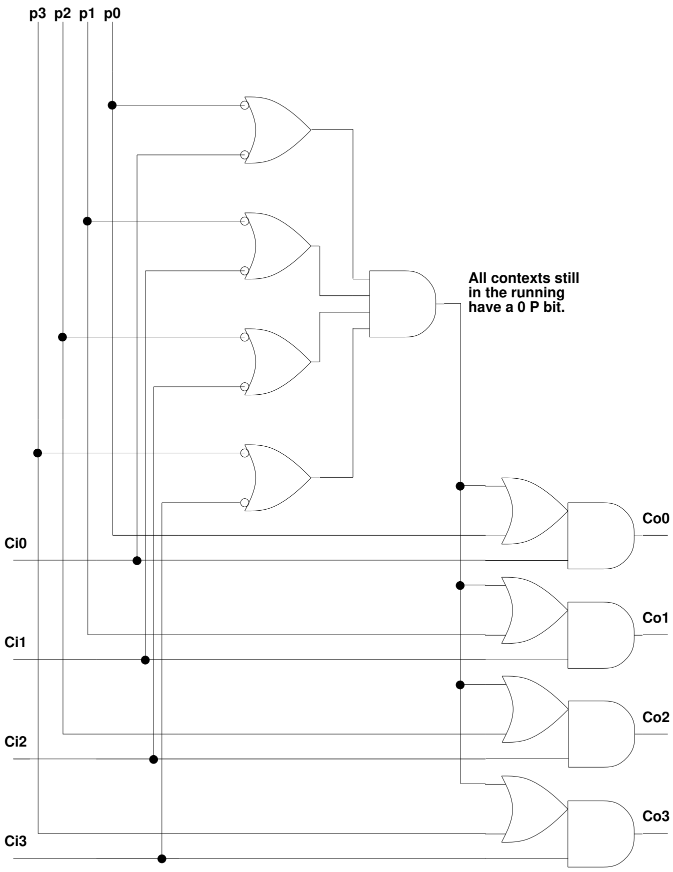


Figure 4.2: Bit slice of the MAX circuit for 4 contexts.

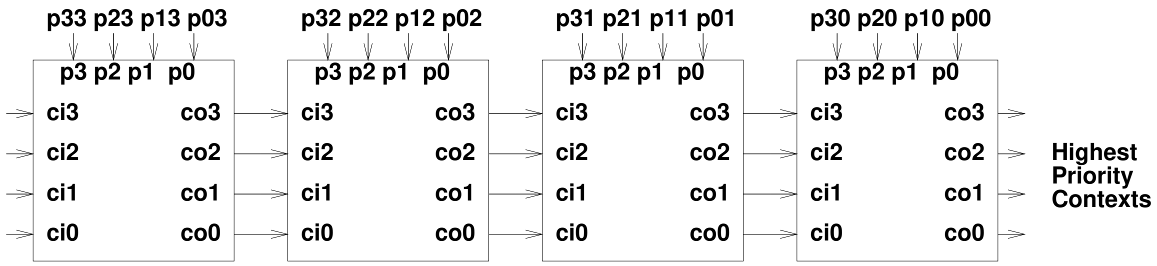


Figure 4.3: MAX circuit for 4 contexts with 4-bit priorities.

4.1.2 Software

It is possible to do both context selection and priority changing using software with minimal hardware support. A fault occurs on a cache miss, and the fault routine selects the next context. The April processor uses a 6 cycle trap routine to do round-robin context switching [5]. Waldspurger and Weihl [101] present one scheme for doing this which uses dedicated registers in each context to contain a thread queue data structure which the processor uses to do software context switching. Each context has its own registers, three of which are used to store an IP, a pointer to the next context, and a Processor Status Word (PSW). On a context switch, 4 to 6 cycles are required to set up the processor to point to the next context's registers, load the PSW, and jump to the new thread's IP[1]. These values are kept in registers so that no memory data structures are being used. Although they looked only at round-robin scheduling, they suggested adapting the scheme to implement more complicated prioritized schemes.

A simple modification to the Waldspurger and Weihl scheme would chain loaded contexts into priority groups. A typical context switch from one context in a group to another context in a group would require just their simple 4 to 6 cycle context selection cost, or less if the IP and PSW registers are duplicated for each context. The cost of changing a priority however would be much higher, in the range of 10's of instructions, as it would require deleting the thread from one group and putting it in another. In this scheme instructions would only be running from threads with the highest priority, and threads with lower priority in other contexts would be idle. One can imagine more complicated data structures, but more complicated data structure will either consume more resources if stored in registers, or have to be stored in memory resulting in slower context switching. Also, as we will discuss in Section 4.2, incorporating thread stalling with this software approach is difficult.

4.1.3 Hardware/Software

To reduce the hardware cost, a combination of hardware and software can be used to prioritize the contexts. Rather than maintaining a priority register for each context, the runtime system maintains a software priority queue in memory, and uses a single *hardware context control register* (HCCR). The hardware control register could consist of $Clog_2C$ bits, log_2C bits for each context, with each set of bits being a *hardware* priority for each loaded thread. The software has the job of translating the N-bit software priorities into log_2C bit hardware priorities that maintain the same relative ordering of the loaded threads. The hardware priorities of the contexts then determine which context to select on a context switch using the same circuit as for the pure hardware case, except that a factor of $\frac{N}{log_2C}$ less register and comparator area is needed.

[1]Their scheme aims at having a flexible number of contexts, and a flexible number of registers per context. This is not our goal here, but the method of context switching is still applicable.

Alternative	Context Selection Cost	Priority Change Cost
Hardware	0 cycles	1 cycle
Software	4-6 cycles	O(C) cycles
Hardware/Software	0 cycles	O(C) cycles

Table 4.1: Summary of context selection costs and priority change costs for different implementation schemes, assuming C contexts.

Alternative	Approximate Hardware Costs
Hardware	C priority registers of N bits $O(NC)$ comparator area
Software	none
Hardware/Software	hardware control register of $Clog_2C$ bits $O(Clog_2C)$ comparator area

Table 4.2: Summary of major hardware costs for the different implementation schemes, assuming C contexts, and N-bit priority. Note that this is the extra hardware required to do prioritization in addition to the extra hardware required for the multiple contexts.

In this scheme, the context selection time is zero cycles, just like the hardware schemes, since the HCCR hardware does the selection. However, the priority changing time is a function of the software that updates the hardware control register. Changing the priority of a loaded thread requires changing its position in the software queue, and determining the new HCCR. In the best case, an application might use a number of priority levels less than or equal to the number of contexts. In the worst case, an application might use many more priority levels than the number of contexts. Maintaining the relative priorities of the different contexts on a thread priority change then becomes linear in the number of available contexts.

Comparison of Alternatives

The relative performance and hardware costs for the different implementation alternatives are shown in tables 4.1 and 4.2. The hardware alternatives are more attractive because they offer high performance context selection and priority changing, for only a small incremental increase in chip area.

The hardware alternative minimizes the number of cycles required for both context selection and priority changing. The chip area cost of doing this is the area of the priority registers and the comparator. Compared to the area cost of implementing a multiple-context processor in the first place, this cost is low. If each context has 32 registers, then the extra priority register represents a 3% increase in total register area, with the comparator area being considerably smaller than this. Current processors devote between 1% and 5% of their area to registers [38, 40]. Assuming that we devote 20% of the chip area to general registers because of the increased number of contexts[2], the extra area cost will be less than 1% of the total chip area.

The hardware/software option is the next best option as it offers the same minimum context switch time, at a considerably smaller area. The difficulty arises in translating a software priority into a hardware priority that maintains the same relative priority of the loaded threads. Experiments are required to determine the impact of the increased number of cycles required to change a thread's priority.

Finally, the software option is the least appealing of the options. Both the context switch cost and the cost of changing a threads priority are non-negligible. In particular, the minimum of 4 to 6 cycles required to choose the next context actually doubles the cost of a context switch when added to the nominal 5 cycle cost of draining the pipeline. This may be alright if the latencies being tolerated are sufficiently long, but it will limit the ability of the multiple-contexts to tolerate shorter latencies (to local memory for instance), and further, the context selection cost can easily rise above this if we try to implement a more sophisticated scheduling policy. For instance, if we wish to incorporate thread stalling, the software now has to include a check to see whether a context is stalled or not before scheduling it. Finally, although the software option has no special hardware costs, it may require that certain registers be reserved to contain a scheduling data structure.

4.2 Memory System Prioritization

As discussed in Chapter 3, the memory system must also participate in the prioritization process in order for prioritization to be effective. It does this in several ways. First, it stalls and unstalls contexts depending on whether their memory reference has been satisfied or not. Second, when bandwidth is limited and several contexts have memory references waiting to be issued, the memory system issues the memory requests in highest priority order. Third, we can optionally implement preemptive scheduling so that when a high priority thread is unstalled the memory system will cause an interrupt and a context switch if a lower priority thread is executing. This section describes a shared-memory system based on *transaction buffers* that allows both of these functions to be performed in a straightforward manner.

[2]This is a pessimistic estimate since the area of a multi-ported register file is typically dominated by the area requirements of implementing the multiple ports rather than by the extra storage cells [39].

4.2.1 Transaction Buffer Implementation

Any system which allows multiple outstanding memory references requires a *lock-up free* cache [56, 84, 92] that allows the cache to continue to accept new requests even if a memory request misses in the cache. An essential component of such a system is some means of tracking outstanding memory references. One way of doing this is by using transaction buffers. When a miss in the cache occurs, an entry is made into one of the transaction buffers that includes all the information necessary for completing the memory request: an address, the type of request and other status bits, and space for data. When the requested cache line arrives and a match is detected on one of the transaction buffers, the transaction buffer controller completes the memory operation which may require storing data in the transaction buffer, and updating the cache.

Figure 4.4 shows the transaction buffer scheme we implemented in our simulator. Each context has a transaction buffer associated with it, and uses this transaction buffer to satisfy its requests. Transaction buffers work as follows to satisfy a memory request:

1. Memory Request: when a context makes a request, it sends the request to both the cache logic and the transaction buffer. The request is satisfied immediately if there is a hit in the cache, or if the transaction buffer contains the valid data. Otherwise, an entry is made in the transaction buffer.
2. Request Issue: the transaction buffer issues the memory request once the necessary resource is available, and there are no other transaction buffers that have higher priority requests. A request requires either the local memory system or the network depending on whether the reference is a local or a remote reference. Note that if several transaction buffers are waiting for the same cache line then the request will only be issued once. The transaction buffer logic performs an associative match on the address field in order to merge memory requests to the same cache line.
3. Request Completion: upon reception of a return message, the transaction buffers and the cache are updated. The transaction buffer logic updates all the transaction buffers that require the cache line with the appropriate status and data. The update occurs as if each request executes sequentially on the cache line, and so the state of the cache line can change. For instance one of the transaction buffers may execute a write, and this would change the value of the cache line. The next transaction buffer could then read this word. Once all the transaction buffers are updated, the cache line is inserted into the cache so that further references to the cache line will be satisfied from the cache. Note that the reference may or may not be successful depending on the protocol message that is returned to the transaction buffer. For instance, one protocol message simply says that any requests must be tried again because the cache line is busy being written in another processor.

This scheme can also be extended to handle multiple outstanding references per context, as well as software prefetching. To do this, additional transaction buffers are used that

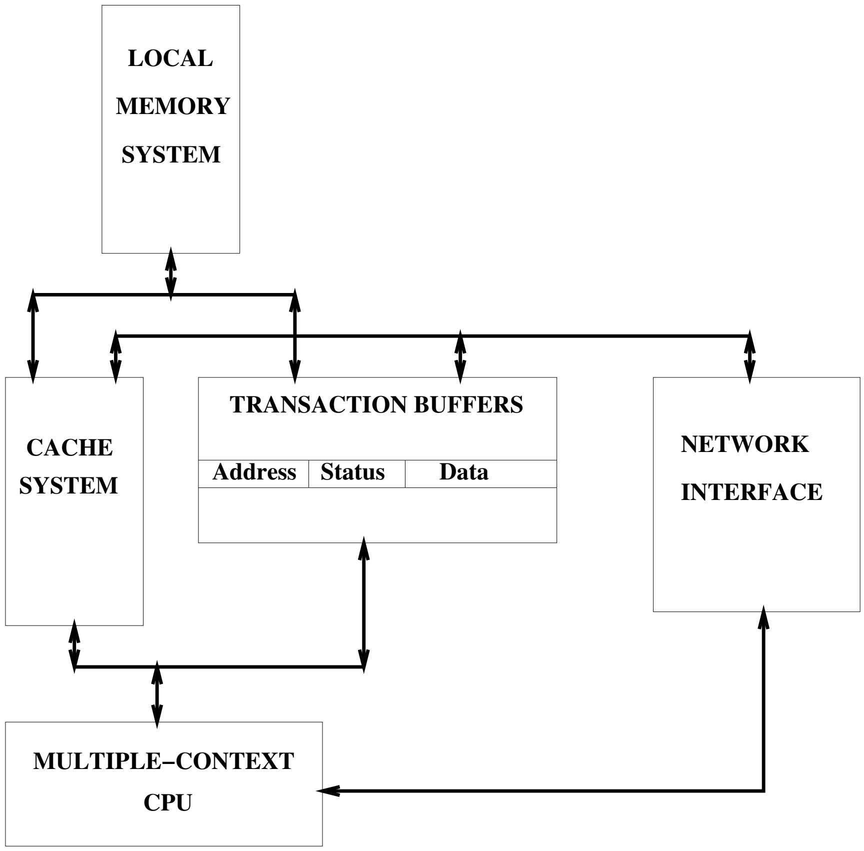


Figure 4.4: Transaction buffer interface to the cache system, the memory system, and the network interface.

allow additional memory requests to be outstanding. In the case that multiple requests from a single context are allowed, the transaction buffer must include information about destination registers. If the memory request is a software prefetch, then the cache line is simply stored in the cache and the transaction buffer does not store the data in its data field.

Comparison to Alewife Transaction Buffers

Note that this transaction buffer scheme is different than the one implemented in the Alewife system [58] from which we take our shared memory protocol. In the Alewife machine the transaction buffers store a complete cache line that continues to participate in the cache coherence protocol as if it were another line in the cache. In addition to keeping track of multiple outstanding memory requests, the transaction buffers are used as an aid to avoid memory thrashing problems, as a fully-associative cache, as a flush queue to local memory, and as storage for prefetched data. Of particular importance is its role as a means of avoiding thrashing problems that occur in their system [57]. Because they rely on polling in which memory requests must be retried until they are satisfied, different thrashing situations can arise in which data arrives at a node but is invalidated by cache conflicts before the memory request can be re-issued to use the data. If this happens repeatedly, the processor can be fatally livelocked. Using a combination of transaction buffers, disabling interrupts, and selective locking, the polling scheme can be made to avoid these situations.

Our implementation avoids the thrashing problem by using a signaling approach. When a cache line arrives at a processor, all transaction buffers that are waiting for this cache line commit before the cache line is allowed to be invalidated. Thus forward progress is guaranteed without having to disable context switching. Note that to implement atomic operations such as Fetch-and-Add in this way, there must also be an ALU associated with the transaction buffers.

Which type of implementation is better is not an issue that we deal with in this thesis. The best approach would probably be a combination of using a small fully-associative cache as a victim cache and prefetch storage, along with a mechanism that would avoid thrashing by having memory requests commit when their cache line becomes available. We implemented the signaling approach to simplify the modification of our simulator, and because modern processors that use register scoreboards and allow multiple outstanding memory references will necessarily use a signaling approach where values are returned directly into registers. Finally, the effects of having such an associative store should be similar to increasing the associativity of the main cache [47].

4.2.2 Thread Stalling

Threads are stalled to prevent them from being scheduled as long as they cannot make any forward progress. It is more useful to run a thread which is of lower priority but is not stalled, then to remain waiting for a higher priority thread to be able to make progress[3]. Stalling threads requires the tracking of which threads are waiting for references to be satisfied, and which are not.

In the simplest case where each context has at most one outstanding memory reference and a single transaction buffer associated with it, we implement stalling by disabling the context on a miss, and re-enabling it when the memory reference is satisfied. We can also do this if there are multiple outstanding references per context, but only a single primary reference on which the context can stall. This might be the case if we are doing software prefetching: each context may have several prefetch requests in transaction buffers, but only a single primary memory request on which it can stall.

In this single primary outstanding reference case, it is possible to use either polling or signaling in order to detect the completion of a memory request. With polling, each context must re-issue a memory reference until it succeeds. The stalling and unstalling of a context prevents a context from polling without the data having first been returned. The advantage of polling is that only the context selection logic has to be modified, and not the processor pipeline itself. Also, all the state of an unloaded thread is contained within the context registers. Alternatively, a signaling approach can be used in which the processor does not have to re-issue the memory request, but rather the data is returned directly to the required register before the context is re-enabled. Kubiatowicz [58] points out a number of the complications in implementing a signaling approach to context switching, including the more complicated register file and pipeline design, and additional complication in dealing with memory operations that may trap.

If multiple memory requests are allowed per context, then it is necessary to return data directly to registers. Modern superscalar processors typically allow multiple outstanding memory requests, and out of order issue and completion of memory requests. Such processors use a scoreboard to keep track of which registers are present and ready for use, and which are still outstanding. If the processor tries to use a register that is not yet available it stalls. When a memory request returns it is sent directly to the register which unstalls the processor. Thus stalling is a very straightforward operation that puts very little burden on the processor design. However, there are some additional complications as the transaction buffer must keep track of the correct register to return data to, and control must be provided to allow value to be inserted into the register file.

[3]This is generally true, but not always true. For instance if running the lower priority thread removes some of the critical thread's data from the cache, overall performance can still suffer.

Stalling and Context Selection

We can easily incorporate stalling into both the hardware context selection scheme, or the hardware scheme with software support. This can be done with a very simple addition to the logic of figure 4.3. We assume that for each context we have a thread presence bit indicating whether the context is loaded with a thread, and stall bit that says whether thread can make progress or not. The thread presence bit can be ORed with the complement of the stall bit and this result can be used as an input to most significant carry-in of the MAX logic. The output of the MAX becomes the highest priority contexts that have a loaded thread and are not stalled. For the purposes of the next context selection, the currently executing context should be considered stalled. This means that the context selection will not choose the executing context on a context switch, and the fact that the stall bit has just been set for the currently executing context does not have to propagate through the MAX circuit. If all the contexts are stalled, we must disable the issuing of all instructions.

Implementing stalling in the purely software case is less straightforward because without explicit hardware support, the software must check to see whether a thread is stalled or not and this adds additional cycles to the context switching. Also, instead of switching immediately to an unstalled thread, the software may have to run through several contexts before it finds one that is unstalled.

4.2.3 Memory Request Prioritization

For prioritization to be effective the processor must issue memory requests in priority order to both the local memory system, and to the network interface. The logic required to prioritize the memory requests is the same as the logic required to prioritize context selection. Whenever the local memory system is available the highest priority memory request requiring the local memory system is issued, and whenever the network interface is available, the processor issues the highest priority pending memory request in the transaction buffers.

Note that there is arbitration that is required for both the local memory system and the network output resources. The local memory system must service requests coming from the transaction buffers, as well as requests from the local cache (when a dirty line is being written back for instance), and requests from remote nodes that come in from the network. Similarly, the network interface must accept messages from the transaction buffers, the local cache, the local memory system, as well as messages sent directly by the user.

Although the exact priority of access to these resources is an interesting issue, we have defined the order somewhat arbitrarily. The local memory system gives first priority to requests from the cache and from the network interface, and second priority to new requests from the transaction buffer. The reasons for doing this are that we want to service already issued requests before issuing new requests into the system, and we want to minimize the backup of messages into the network. Similarly, the output network interface gives first priority to the cache logic, the local memory system, and user messages, and second priority to new requests from the transaction buffers. Independent of priority, any given request to one of these resources will be delayed until the resource is free.

4.2.4 Preemptive Scheduling

Once stalling is introduced, one must decide whether scheduling will be preemptive or not. Preemptive scheduling means that when a lower priority thread is executing and a higher priority thread that is stalled becomes unstalled, then a context switch is forced even if there is no cache miss or synchronization failure. In the non-preemptive case, the switch to the higher priority thread will only occur on the next normal context switch. The preemptive case minimizes the execution time of a potentially critical thread, but pays the cost of a context switch. Also, the preemptive case requires that hardware be provided to force a context switch when a higher priority thread becomes unstalled. The non-preemptive will delay the execution of the higher priority thread until the next context switch.

Context switches usually occur frequently enough that this effect does not occur, and for convenience reasons, our simulations used the non-preemptive approach. It should be noted that the extra costs of the preemptive case are moot in architectures that provide cycle by cycle context switching with instructions from different threads being interleaved on a cycle by cycle basis, since all the mechanisms for preemptive scheduling are already in place.

4.3 Unloaded Thread Prioritization

A software priority queue prioritizes the unloaded threads. Thus there is considerable flexibility in how this is implemented. Preemptive scheduling however requires some hardware support in order to force a loaded thread to swap out when a higher priority thread becomes available.

Preemption may be desirable when a new thread is created, and its priority is higher than the priority of one of the loaded threads. Preemptive scheduling at the unloaded thread level is different than in the loaded thread case. An interrupt must occur, and the context with the lowest priority thread should run an interrupt routine that swaps its thread with the higher priority thread. Whether preemptive scheduling is advantageous or not depends on the particular situation. If all the loaded threads are long running threads, then it makes sense to incur the overhead of a thread swap, and allow the more critical thread to proceed. If on the other hand threads are short running, then a context will soon be free anyway, perhaps in a shorter time then it takes to perform a thread swap, so that it is best to wait.

Implementing preemption in a multiple context processor is straightforward and can be implemented as an asynchronous trap. The trap handler simply runs the code to swap the lowest priority thread with the new higher priority thread.

4.4 Summary

In this chapter, we looked at ways of implementing thread prioritization in a multiple-context processor, both for loaded and unloaded threads, and show that it can done in a straightforward manner.

The loaded thread prioritization must prioritize both the selection of contexts on a context switch, and the memory requests going to the memory system. Simple hardware can implement the context prioritization: one priority register per context and a multi-way MAX circuit. Other schemes that use only software, or a combination of hardware and software are also possible, but they increase the context selection time and/or the priority changing time. All that is required to incorporate stalling into the hardware scheme is a bit indicating if a context is still waiting for an outstanding memory request.

The memory system prioritization allows context stalling, prioritized memory requests, and preemptive scheduling. Transaction buffers are used to issue and track memory requests. When a context is waiting for a memory request it is stalled and cannot be chosen to execute. When several requests are waiting in the transaction buffers to be issued, the highest priority one is selected once the necessary resource is available using the same type of circuit that is used to prioritize the context selection. When a memory transaction completes, the corresponding context is unstalled. Preemptive scheduling can also be implemented by having a trap generated when a higher priority thread becomes unstalled.

Finally, a software scheduler prioritizes unloaded threads and has considerable flexibility in its implementation. Preemptive scheduling is done by forcing a trap when a thread is created that has a higher priority than a currently loaded thread.

Chapter 5

Simulation Parameters and Environment

In the next few chapters we discuss the results from simulation studies of how prioritization affects performance of several different types of benchmarks. With these simulations we accomplish two things: first, we show that the simple model of Chapter 3 correctly predicts some of the important behavior that occurs in real programs, and we uncover effects that occur in real programs but that were not brought out by the model due to the idealized assumptions that were made. Second, we show that thread prioritization is a versatile scheduling mechanism that can be used in many different ways to implement the scheduling strategies that are appropriate for different types of problems.

In this chapter we list and discuss the important system assumptions and the parameters that were varied in the studies, and we give an overview of the Proteus architectural simulator [15, 25] and the simulation methodology. The most important system parameters are the multiple-context processor parameters, memory system parameters, and network parameters. These parameters affect the extent to which latency can be tolerated, as well as how much impact latency tolerance has on performance. Processor parameters include the number of contexts, the hardware context switch time, the context selection time, and the thread switch time. The number of contexts affects how much latency can be tolerated, whereas the hardware context switch time, the context selection time, and the thread switch time are overhead components that can limit the effectiveness of multiple-context processors as a latency tolerating mechanism. Memory system and network parameters affect the latency of memory requests, and the bandwidth available to satisfy requests. A shared memory protocol is used and this increases latency because the protocol can require several messages to be sent and received before the data is available. The local memory latency and bandwidth, as well as the network latency and bandwidth also have first order effects on the latency of data requests.

The Proteus architectural simulator was modified to simulate a multiple-context multipro-

Parameter	Typical	Range
Number of Contexts	-	1-16
Hardware Context Switch Time	5 cycles	1-10 cycles
Time to Unload Registers	32 cycles	4-200 cycles
Time to Reload Registers	32 cycles	4-200 cycles
Software Scheduling Cost	10 - 100 cycles	-
Cache Latency	1 cycle	-
Local Memory Latency	20 cycles	20-160 cycles
Memory Controller Throughput	4 cycles/request	4-20 cycles/request
Network Wire Delay	1 cycle	-
Network Switch Delay	1 cycle	-
Network Flit Size	16 bits	-
Network Interface Input Bandwidth	1 flit/cycle	-
Network Interface Output Bandwidth	1 flit/cycle	-

Table 5.1: Important system parameters.

cessor. It provides a reconfigurable high-level substrate on which to write applications and measure system performance. To achieve reasonable simulation times, it makes certain simplifying assumptions such as only simulating shared memory for certain important data structures, and assuming that all instructions and stack references hit in the cache. The applications express parallelism explicitly, and are written in C with language extensions for concurrency.

5.1 System Parameters

As discussed in Chapter 1, the simulated system consists of a collection of multiple context processors, connected with a high performance interconnection network. Throughout the simulations we vary different system parameters to represent variations in the architecture of the three main components of the machine: the processor datapath and pipeline, the memory system, and the network architecture. We describe the important parameters below, and summarize them in Table 5.1 along with the different values used throughout the simulations.

5.1.1 Processor Parameters

Number of Contexts

The number of hardware contexts (register sets) is as key parameter. More contexts generally allow longer latencies to be tolerated and allows more flexibility in the scheduling. At the same time, more contexts also require more hardware. In our simulations, we vary the number of contexts from 1 to 16 contexts. Our simulations confirm previous results that the optimal number of contexts varies depending on the application and the typical latencies [105, 2, 81]. For the parameters we used, between 2 and 8 contexts typically gave the best performance.

Hardware Context Switch Time

Hardware context switch time is the time from when one context is stalled due to a cache miss or synchronization failure, to the time that a thread in another context begins execution. For a block multithreaded processor with a conventional processor pipeline, the minimum time to switch contexts is the time to drain the pipeline — all the instructions following the stalled instruction are squashed. This cost increases if additional instructions have to be executed to perform the context switch. For instance, to do context switching in the April processor [5], a small trap routine executes to switch the registers being used and save and restore status information.

With suitable modifications to the pipeline, specifically some way of saving the pipeline state of any context that stalls, the context switch time can be reduced to zero cycles. This requires significant redesign of the processor pipeline [63]. When a pipeline does dynamic cycle by cycle interleaving of instructions, the context switch time is by definition zero cycles. However, at the time of a stall there may be several instructions from the stalled thread that are in the pipeline and need to be squashed, resulting in a larger than zero cycle cost for the stall. Once again however, if we store the pipeline state of a context, the cost of thread stalling can be zero cycles.

In our simulations, we consider a range of 1 to 10 cycles, with the typical cost being 5 cycles. Experiments in which we vary the context switch time show that the context switch time can have a large impact on performance when context switching is frequent, and latencies are short. Thread prioritization reduces the impact of the higher context switch time by reducing the number of unnecessary context switches.

Context Selection Time

In addition to the hardware context switch time, there is overhead associated with selecting the next context to be scheduled. This cost can be 0 cycles if a simple hardware scheme is used, or it can be several cycles if the processor must determine in software which context is to be chosen next. Our simulations merge this cost with the cost of the hardware context switch.

Thread Swap Time

The thread swap time is the time it takes to remove one thread from a hardware context, and load another thread into that hardware context for execution. For a conventional processor, this typically requires unloading the state of the context registers into memory and loading the state of the new thread into the context before execution can begin or continue. It also requires manipulating the software scheduling data structures to insert and delete the threads.

For the purposes of our simulation, we consider a range of 4 to 200 cycles to save or restore the registers during a thread swap, with a typical value of 32 cycles. A 200 cycle save/restore time might occur if restoring the context causes several misses in the cache. A 4 cycle save/restore time is achievable with hardware techniques such as the Named-State Register file [77], that dynamically manages a register file that is shared between all the contexts, and only one or two special registers, such as the instruction pointer and a status word, have to be explicitly saved.

The time to manipulate the software scheduling queue is explicitly accounted for in our simulations by having software routines that do the queue manipulation. Depending on the implementation, this cost typically ranges from about 10 to 100 instructions.

5.1.2 Memory System

Local and Global Shared Memory

For our simulations we assume a machine which has both local and global memory. Local memory is only visible to the processor which owns it, whereas global memory is visible to all processors. We assume support for global shared memory in the form of a directory-based cache coherence protocol. The shared memory protocol used is an invalidation protocol based on an early version of the protocol used in the Alewife machine [18] modified to include transaction buffers as described in the previous chapter. This protocol provides a sequentially consistent view of memory. Conceptually, the hardware used to support the shared memory is in the form of a cache controller, and a memory controller. The cache controller handles all protocol requests which require action in the local processor cache, whereas the memory controller handles all protocol messages which require action in the local processor memory, including the management of the directory information.

A current limitation of the simulator as it is currently implemented is that it only allows a single outstanding unsatisfied reference per context. Allowing multiple outstanding references per context would have several effects: first, it would improve single thread performance since references from one thread can proceed in parallel. Second, it would allow fewer contexts to tolerate a given amount of latency by increasing run lengths between context switches, and reducing the number of contexts required to fully utilize the available memory bandwidth. This limitation leads to a worst case scenario for multithreading as a latency tolerance mechanism and for any given latency we find that a higher number of contexts is required to tolerate latency. The effect of allowing multiple references per context can be approximated by reducing the average latency so that fewer threads are required to tolerate latency.

Cache and Memory Latencies

The base cache latency is 1 cycle in the case of a hit, and the base memory latency is 20 cycles in the case that the data is in local shared memory, and no shared memory protocol messages have to be sent. This 20 cycle latency corresponds roughly to a processor cycle time of 5ns and a memory system access time of 100ns which is a reasonable baseline given current technology [93]. Technology trends indicate that this difference between processor speed and memory speed will further increase over the coming years [43].

The latency of any given memory reference is different depending on whether the data is in the cache, is not in the cache but is in local memory, or is not in the cache or the local memory but on remote node. The memory reference time is also affected by the cache coherence protocol, which may send out several messages per memory transaction, and possibly have to wait for reply messages. Thus, in addition to the base cache and memory latencies, the latency of a memory reference also depends on the performance of the network and the network interface.

We will vary the memory latency between 20 and 160 cycles to simulate longer memory latencies due to differences in processor and DRAM speed, and to simulate the long latencies that occur when data is on a remote node.

Memory Controller Throughput

Another important parameter is the memory bandwidth associated with each node. In our simulations this is modeled as the memory controller throughput, or the rate at which the memory controller can handle incoming protocol messages. Specifically, although the local memory latency may be L cycles, the memory system of each node may be able to accept requests more frequently, say every B cycles, thus pipelining the memory requests so that L/B memory requests can potentially be outstanding at any given time. Pipelining of memory requests in this fashion can be accomplished by having interleaved memory banks. Note that in order for multiple contexts to be efficient in tolerating latency, it is essential to have sufficient memory bandwidth.

In our simulations the bandwidth is varied from a maximum possible local bandwidth of 1 word per cycle or 4 cycles per request for a 4 word cache line, to a minimum bandwidth of 1/5 words per cycle or 20 cycles per request.

5.1.3 Network Architecture

The nodes are assumed to be connected by a low latency, k-ary N-cube network that uses wormhole routing [24]. The network interface consists of a network output interface and a network input interface.

Network Flit Size

A *flit* is the unit of flow control in the network. In our simulations this is 16 bits or half a word.

Network Latencies

Associated with the network are a switch delay and a wire delay. The switch delay is the time for a single flit to be routed through the network switch located at each network node. The wire delay is the time for a single flit to cross from one network node to the other. Our simulations assume a switch and wire delay of 1 cycle.

Network Interface

Although the shared memory view of the machine provides ease of use and programming, it is often convenient to have direct access to the network interface in order to be able to pass information directly via messages. Recent work has shown the benefits of having both shared memory and message passing [54]. Specifically, if the data communication pattern is known, explicit message passing can be used to bypass the shared memory interface and protocol and thus optimize communication. Thus our simulations also assume direct message passing capability, and this capability is used in a number of the benchmarks.

The network interface consists of both a network output interface and a network input interface both of which are tightly coupled to the processor and memory systems. Conceptually, the processor does a message send by assembling a message then doing an atomic SEND operation. Depending on the implementation, the message can either be assembled directly in registers, or can be assembled by writing special memory mapped registers. Thus the cost of sending a message is just the cost of assembling the message arguments. The shared memory controller and cache controller access the network in similar fashion. The output network interface bandwidth is limited to 1 flit per cycle, which is the same as the network channel bandwidth. Messages coming from the cache controller, the memory controller, and the processor are queued and serviced in a first come, first serve manner.

The network input is the more complicated part of the network interface because of its interaction with the thread scheduling. The handling of a message which arrives at a processor must be scheduled along with the threads that currently exist on the processor. We assume that the response to messages is interrupt driven, and is in the style of active messages [100]. This means that when a message arrives it generates a processor interrupt, and runs a message handler which is guaranteed to run to completion in a short period of time. In particular, a message handler can generate and schedule a new thread.

We assume that the cost of an interrupt is the same as the cost of a hardware context switch.

5.2 Simulation Methodology

5.2.1 The Proteus Architectural Simulator

Proteus [15, 25] is a high-performance simulator for MIMD computer architectures. It allows architectural parameters such as the network and the memory system to be varied. Programs are written in C with language extensions for concurrency, and simulator calls that support non-local interactions between processors including shared-memory operations, spinlock operations, inter-processor interrupts, and message passing. Proteus is written in a modular fashion so that certain components such as the network simulator or thread scheduler can be easily modified in order to perform architectural studies. Proteus also has a flexible accounting system that allows the user to modify the costs associated with different functions.

At the software level, Proteus estimates the software cost of the parallel code by compiling it down to SPARC code. This code is run during simulation as if the multiprocessor node were an actual SPARC processor and the simulator keeps track of the number of cycles required to execute the code. It is also possible to change the cost of the code explicitly by adding or subtracting cycles from it. Proteus also provides a number of nice features for debugging applications, and collecting statistics.

Memory Modeling

One of the simplifying assumptions that occurs in order to make the shared memory simulation tractable is to only simulate memory references to data items that are explicitly declared as being shared. Thus references to important shared data structures will be simulated, but instructions references and stack references are local references and are assumed to hit in the cache. This approach is reasonable since instruction cache hit rates are typically very high, as are hit rates on scalar data and stack variables. Cache performance is typically most affected references to large program data structures. To compensate for the fact that all the data accesses are not explicitly being simulated, the size of the cache is reduced.

Proteus Modifications

A number of changes were made to Proteus to reflect the architectural features and assumptions, as well as to correct certain deficiencies of the basic simulator. The most important changes are listed below:

- Multiple Contexts: We extended Proteus to simulate multiple hardware contexts, and to allow the changing of different costs associated with hardware context switching and thread swapping.
- System Routines: We replaced various system routines, specifically the routines that have to do with thread creation, thread deletion, thread suspending, and thread scheduling. We also wrote various system fault routines such as the fine-grain synchronization fault routines.
- Memory System: We changed the shared memory simulation to simulate a more realistic shared memory implementation[1]. We also implemented set-associative caching in the context of the directory based shared memory protocol, and implemented a mechanism for stalling contexts based on data availability. We introduced memory transaction buffers to hold the outstanding memory request from the multiple contexts.
- Network Interfaces: We modified the network interface to reflect the cost of sending a message, and the limited network output bandwidth available to each processor. We also made modifications to cause an interrupt only once a message has completely arrived.
- Instrumentation: We wrote code to measure a variety of interesting parameters such as the number of hits and misses over a given time period.

[1]Thanks to Kirk Johnson for providing us with his version of Proteus which reflected the costs associated with the Alewife shared memory implementation.

5.2.2 Application Assumptions

This thesis discusses small applications that illustrate a variety of problem characteristics, and their interaction with thread scheduling. Specifically, the benchmarks used are meant to reflect the importance of synchronization performance, critical path scheduling, cache performance, and the interaction of thread scheduling with interprocessor interrupt scheduling. Each benchmark will be described in the chapter in which it is first used.

In all the applications, the user explicitly generates parallelism by spawning threads. Each thread is allocated storage that it uses as its stack space to do function calls. The applications use coarse-grained threads, where the thread runs many hundreds or thousands of instructions. This reduces the overhead that comes with the spawning and the destroying of threads.

The synchronization between threads is also explicit and we explore a variety of implementations of locks, barriers, join counters, and fine grain synchronization using Full/Empty bits. Although the threads are coarse-grain in the sense that each thread executes a large number of instructions, they can be fine-grain in the sense that they require frequent access to remote data, and synchronization with remote threads.

Chapter 6

Synchronization Scheduling

In this chapter we show how assigning a priority to threads in a multithreaded computation can improve the performance of synchronization primitives by reducing the number of cycles wasted in spin-waiting, and by preventing spinning threads from slowing down critical threads. Tolerating synchronization latencies is a critical issue since synchronization latencies have the potential to be much longer than simple remote references. Multithreading is the only latency tolerance mechanism that is effective in tolerating these latencies.

Three synthetic synchronization benchmarks are examined: a Test-and-Test_and_Set (TTSET) lock benchmark, a combining tree barrier benchmark, and a queue lock benchmark. In most cases spinning is used as a way of implementing synchronization primitives, but multiple context versions suffer from the problem identified in Chapter 3: spinning threads consume processor resources and delay critical threads.

The results show that by correctly prioritizing threads, synchronization performance is substantially improved. Using the priority to determine which threads are loaded improves runtime performance the most. With exactly prioritized threads, performance improvements ranged up to 66% with 4 threads per processor, and up to 91% with 16 threads per processor. Using the priority to choose among loaded threads is also important, in some cases with runtime improvements up to 20% with just 4 contexts, and improvements of up to 83% for 16 contexts. Unprioritized scheduling shows much higher sensitivity to changes in the thread swap time and the context switch time because many more unnecessary thread swaps and context switches take place than when threads are prioritized.

6.1 Synchronization Scheduling

6.1.1 Synchronization Scenarios

For each benchmark in this chapter, three different scenarios are considered:

1. **SINGLE:** There are several threads, but there is only one context so that only a single thread is loaded at a time.
2. **ALL:** There are sufficient contexts so that all threads created can be loaded. We use 16 contexts in our simulations.
3. **LIMITED:** There are several contexts, but there are potentially more threads than contexts so that only a limited number of the available threads are loaded. We use 4 contexts in our simulations.

Each of these situations represents a different part of the scheduling space. The **SINGLE** and **LIMITED** cases represent situations in which not all threads can be loaded at the same time. These cases can arise in the context of data dependent thread spawning, runtime dynamic partitioning, or in a multiprogramming environment. The **ALL** case is balanced in the sense that all threads can be loaded at once. The **SINGLE** case illustrates how the thread scheduler must keep the critical thread loaded and avoid unnecessary thread swaps to achieve good performance. The **ALL** case illustrates how the thread scheduler must also avoid unnecessary context switches for good performance. Finally, in the **LIMITED** case the thread scheduler must both keep the critical thread loaded in a context, and avoid unnecessary context switches.

6.1.2 Synchronization Scheduling Strategies

A number of different scheduling strategies are possible when considering how to deal with a failed synchronization test. The effect of spinning versus blocking has been studied in the context of shared memory multiprocessors [48, 67], where blocking means suspending the thread by swapping it out of its context and waking it up at a later time. These studies have shown that it is possible to use competitive waiting algorithms in which on a failed synchronization a thread first spends some fixed time spinning, and only swaps threads after this spinning time has elapsed. For instance it is easy to show that if a thread first spins for the amount of time it would take to block before swapping, then the cost of this competitive strategy is no more than two times the cost of the optimal choice of swapping or spinning. With multiple contexts it is also possible to switch-spin [67] rather than just spin while using these 2-phase strategies. Switch-spinning means context switching and running round-robin though all the other available contexts, potentially doing useful work rather than just spinning. All these strategies are heuristics that use only limited information about the problem in order to decide what combination of spinning and blocking is best to adopt.

Our approach to determining what synchronization strategy should be adopted is to use extra information about the situation of the threads. A priority associated with each thread gives a clue as to whether the thread should swap, or whether it should spin. The priority is more useful with multiple hardware contexts where there are more choices than just spinning or swapping: the processor can spin, it can do a context switch, it can swap, or it can do both a swap and a context switch. The priority associated with each thread helps make the correct decision.

The following sections describe different synchronization benchmarks that illustrate different aspects of the synchronization scheduling problem, and show how prioritization can be used to help make scheduling decisions. These benchmarks are shown in increasing order of complexity. The Test-and-Test_and_Set benchmark shows how prioritization can be used to prevent a thread owning a lock from being descheduled before it has released the lock. The barrier synchronization benchmark shows how prioritization can be used to identify a critical thread on a given processor and devote all processor resources to this critical thread. The final benchmark, a queue lock benchmark, shows how thread prioritization can be used to guarantee not only that a thread will not be descheduled when it owns a lock, but also will cause critical threads to be ready and waiting to accept the lock when it is released. In all cases the thread prioritization is used to minimize the number of unnecessary context switches and thread swaps.

6.2 Test-and-Test_and_Set

Mutual exclusion is a means of ensuring that only one processor at a time is accessing shared data. In shared memory multiprocessors, this is often implemented using spin locks, in which threads wait for access to the lock by spinning on a variable waiting for it to be changed to a certain value. Once this value changes, the thread can acquire the lock. Mellor-Crummy and Scott [71] give a good overview of different mutual exclusion algorithms and of other spin lock studies [36, 8]. This section considers the simple Test-and-Test_and_Set (TTSET) lock. Section 6.4 considers a more complex queue lock.

TTSET is a reasonable way of guaranteeing mutual exclusion in the case that there are only a few processors trying to synchronize and the contention is low. Each processor trying to acquire the lock first polls the synchronization variable until it becomes true (The Test portion of Test-and-Test_and_Set). When the lock is released and becomes true, the processor executes a Test_and_Set operation. If this operation succeeds the lock is set to false and the thread has successfully acquired the lock, otherwise the thread must go back to the polling phase.

To benchmark TTSET mutual exclusion, a synthetic benchmark was run on Proteus. A number of threads are created on each processor, and these threads all try to acquire a single lock. Once the thread acquires the lock, it runs a critical section, releases the lock, and then runs a non-critical section before attempting to re-acquire the lock again. With multiple contexts, after each failed Test a context switch takes place. The length of the critical section is fixed, and the length of the non-critical section is based on a uniformly distributed random variable. Two cases are considered: a high contention case in which the non-critical section is short, and a low contention case in which the non-critical section is relatively long. In this test threads swap out of the hardware contexts at the end of an operating system (OS) quantum, and unloaded threads swap in This OS quantum can be considered as an OS scheduling quantum in a multi-programming system, or a slicing quantum that is used to guarantee some scheduling fairness for the threads of a single application. The evaluation criterion chosen is the number of times the lock is acquired in a fixed time interval.

The test was run with just 4 processors, since TTSET mutual exclusion is most appropriate for a small number of processors. The quantum was chosen to be 10000 cycles, and the measurements were taken over a 10^7 cycle period. The critical section is approximately 100 cycles long when run to completion without context switching, but two potential context switches are forced during its execution to simulate a cache miss. If the critical thread does do a context switch while it is executing this section, then the time from when a thread acquires a lock and then attempts to release it can be much longer than 100 cycles. For the high contention case the length of the non-critical section varies from 50 to 150 cycles with a uniform distribution, and for the low contention case the non-critical section varies from 500 to 1500 cycles with a uniform distribution. While a thread runs a context switch is forced approximately every 40 cycles, again to simulate a cache miss.

The following variations on the TTSET benchmark are run:

- **Unprioritized**: Threads all have the same priority. A thread context switches every time there is a miss in the cache even if the critical section is being run. A context switch also occurs every time the polling Test fails. Even if the thread owns the lock the scheduler can swap it out at the end of a quantum.
- **Prioritized**: When a thread is spinning waiting for the acquisition of the lock (in the Test part of Test-and-Test_and_Set), it has low priority. Once the Test succeeds, then the thread becomes high priority for the Test_and_Set operation. If the Test_and_Set succeeds, then the thread maintains its high priority until it has finished executing its critical section and released lock, after which it reduces its priority before executing its non-critical section. If the Test_and_Set fails, then the thread again makes itself low priority before returning to the Test phase. If a thread has high priority at the expiration of a quantum, the scheduler will not swap it out.

6.2.1 Results

Figure 6.1 shows the total number of lock acquisitions for the three scenarios **SINGLE**, **ALL**, and **LIMITED** with a varying number of threads. For each case two sets of curves are shown, one for the high contention case, and the other for the low contention case. Figures 6.2 and 6.3 show the sensitivity of the benchmark to the thread swap time and the context switch time respectively.

Results show that thread prioritization improves performance in the TTSET lock by reducing the amount of time spent executing the critical section, and by preventing a thread from swapping out if it owns the lock. We find further that performance of the lock is not sensitive to thread save/restore time, because thread swaps are done infrequently, when an OS quantum expires. Performance is sensitive to context switch time, particularly in the unprioritized case with many threads, because it contributes directly to the run length of the critical section. These results are discussed in detail below.

SINGLE

In the **SINGLE** scenario, performance falls off dramatically in the unprioritized case as the number of threads increases. This is because a thread owning the lock often swaps out on a quantum expiration. Before the thread owning the lock can run again and release the lock, all the other threads on the processor swap in and run ineffectually for one quantum. Performance drops slightly faster in the high contention case because the thread owning the lock is more likely to swap out due to higher contention, and when the thread owning the lock swaps out the other threads have less work to do.

When threads are prioritized, a thread never swaps out when it owns a lock, and as a result the performance remains relatively constant for an increased number of threads, both in the high and the low contention cases. If we consider the runtime required to acquire a certain number of locks, for 4 threads per processor the prioritized case has 66% and 47% better runtime performance than the unprioritized case, in the high contention and low contention cases respectively. If the number of threads increases to 16, these numbers increase to 91% and 82% respectively.

ALL

In the **ALL** scenario, performance again falls off with increasing number of threads when the threads are unprioritized, and remains approximately constant when the threads are prioritized. The poor performance of the unprioritized case is due principally to the increase in the amount of time a thread spends running its non-critical section. A thread spends more time in its non-critical section because it context switches due to the simulated cache

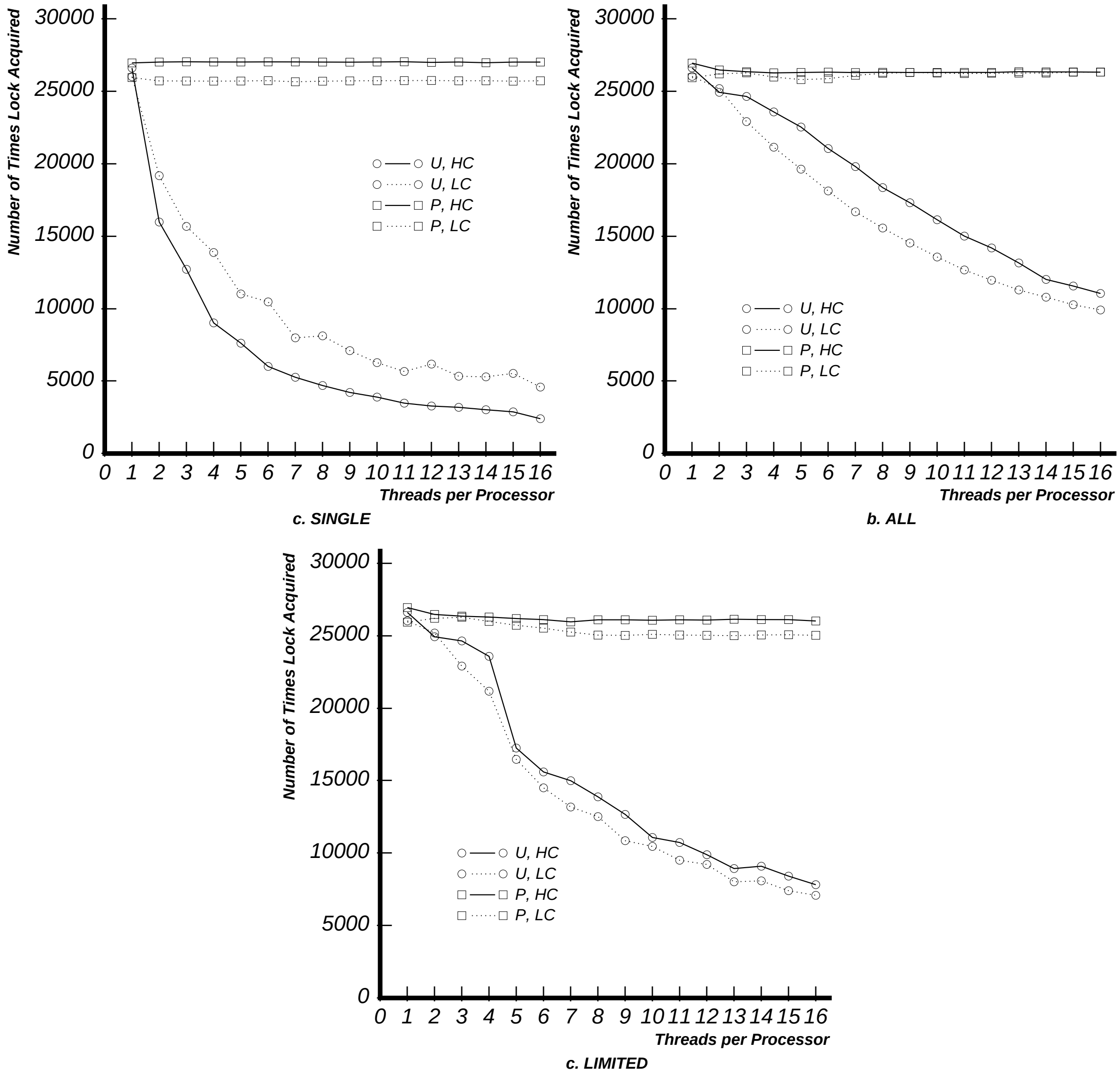


Figure 6.1: TTSET lock acquisitions. Unprioritized (U) and Prioritized (P) cases are shown with both High Contention (HC) and Low Contention (LC).

miss, and the processor must run through all the other contexts before again executing the thread that is in the critical section. Performance falls off slightly faster in the low contention case than in the high contention case because the non-critical threads are more often executing their non-critical sections as opposed to just spinning. A thread executing its non-critical section occupies the pipeline for a longer period of time than a thread that is just spinning, causing it to take longer for the critical thread to resume execution.

Prioritizing the threads solves the problem. A context switch occurs only if there is a thread with higher or equal priority than the one that is currently running in one of the other contexts. Thus the thread will not context switch while running a critical section. Performance remains approximately constant for 1 to 16 threads per processor. Again considering the time required to acquire a certain number of locks, for 4 threads per processor the prioritized case has 11% and 19% better runtime performance than the unprioritized case, in the high contention and low contention cases respectively. If the number of threads increases to 16, these numbers increase to 58% and 62% respectively.

LIMITED

When a quantum expires in the **LIMITED** scenario, as many of the loaded threads as possible swap with unloaded threads in the software queue. If there are no unloaded threads available then nothing occurs, and if there are at least 4 unloaded threads then all the loaded threads can potentially swap out. Thus the **LIMITED** scenario is similar to the **ALL** scenario when there are 4 threads or less per processor because the thread owning the lock never swaps out, and it is similar to the **SINGLE** scenario when there are more than 4 threads because the thread owning the lock can swap out on a quantum. Note however that if a thread that owns the lock swaps out, it takes fewer quanta than in the **SINGLE** case for it to swap back in because the threads swap in and out 4 threads at a time. Thus for 16 threads per processor, the prioritized case has 71% and 73% better runtime performance than the unprioritized case, in the low contention and high contention cases respectively.

Sensitivity to Thread Swap Time

Figures 6.5a and 6.5b show the runtime for the **SINGLE** scenario with a 4 cycle save/restore time, and with a 200 cycle save/restore time. Changing the thread swap time does not have a big impact on performance because the thread swapping only takes place on quantum expiration, and the quantum is relatively large. Also, the quantum can expire at slightly different times on the different processors so that while one processor is swapping in new threads, another processor is still actively acquiring the lock. The larger thread swap causes a small but noticeable reduction in performance of the prioritized, low contention case. This is because time that was previously devoted to running threads in their non-critical section is now being used to do thread swapping.

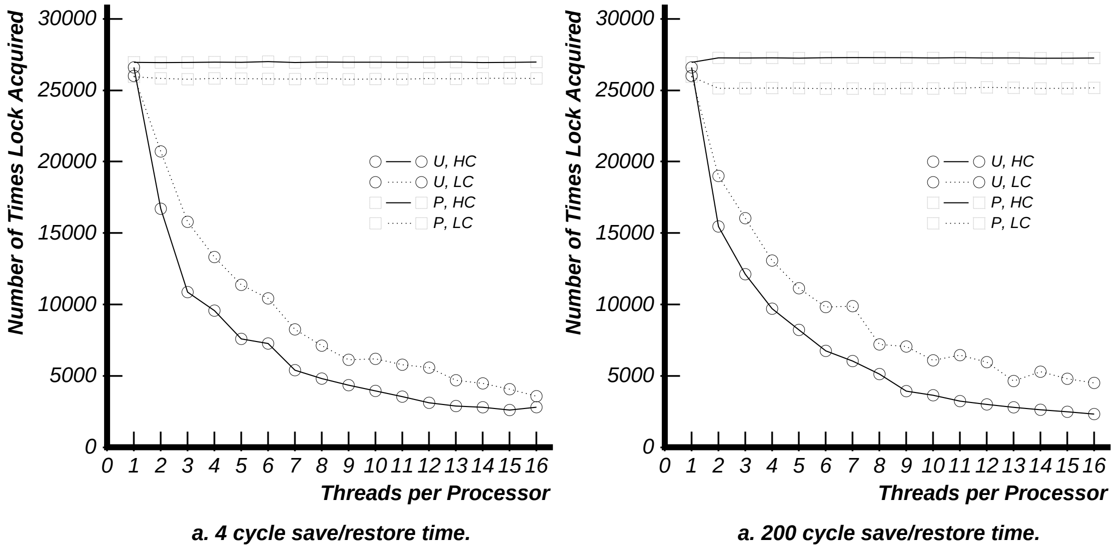


Figure 6.2: TTSET lock acquisitions. **SINGLE** scenario with register save/restore times of 4 cycles and 200 cycles. Unprioritized (U) and Prioritized (P) cases are shown with both High Contention (HC) and Low Contention (LC).

Sensitivity to Context Switch Time

Figure 6.3 shows the runtime for the **ALL** scenario with context switch times of 1 cycle and 10 cycles. The unprioritized case is sensitive to the context switch time. Any increase in the context switch time leads directly to an increase in the time it takes a thread to execute its critical section. For instance, in the high contention case the performance of the unprioritized test drops by 52% in going from 1 to 16 threads with a context switch time of 1 cycle, whereas it drops by 69% when the context switch time is 10 cycles. The prioritized case is also sensitive to context switch time, and an interesting reversal occurs between Figures 6.3a and 6.3b. With a context switch time of 1 cycle, the low contention case performs better than the high contention case, whereas with a context switch time of 10 cycles, it is the opposite. The reason for this is that with a context switch time of 1 cycle the low contention case has more cycles to devote to actually executing threads, while at the same time having lower contention on the lock. The high contention also has these extra cycles but since most of its threads are spinning, it cannot put them to good use. The prioritized high contention case drops by about 3% when the context switch time increases from 1 to 10 cycles, and the prioritized low contention case drops by 13%.

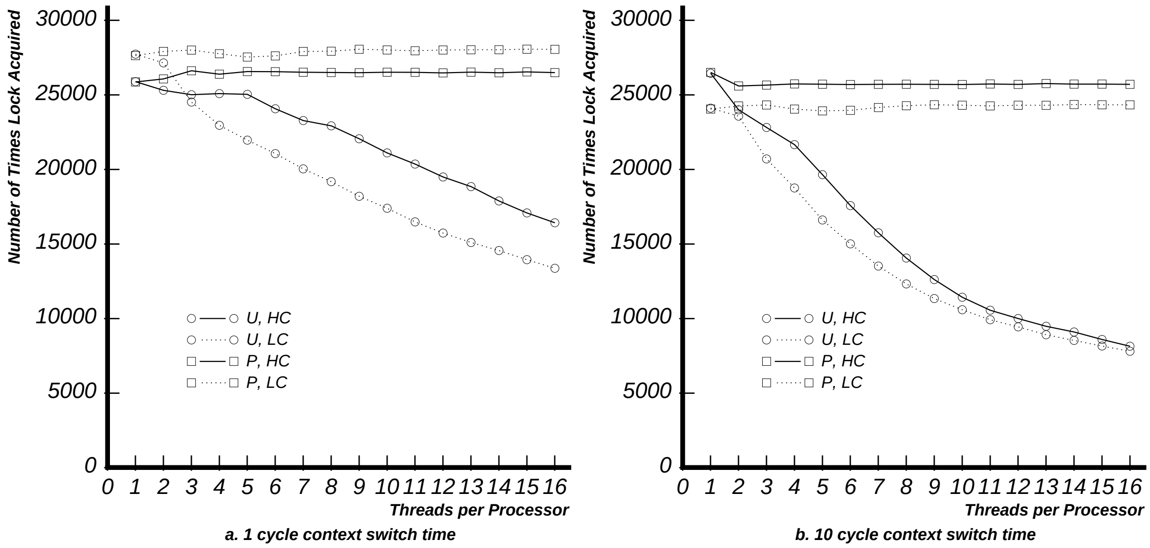


Figure 6.3: TTSET lock acquisitions. **ALL** scenario with context switch times of 1 cycle and 10 cycles.Unprioritized (U) and Prioritized (P) cases are shown with both High Contention (HC) and Low Contention (LC).

Conclusion

These three scenarios show that the prioritizing of both loaded and unloaded threads is important for the performance of the TTSET lock. In particular:

- Using the priority at the software level prevents a thread that owns the lock from swapping out when a quantum expires.
- Using the priority at the hardware level allows the thread owning the lock to quickly execute its critical section and release the lock.
- The TTSET benchmark is not sensitive to the thread switch time because the thread switch time is only a small fraction of the OS quantum.
- Using thread priorities reduces the sensitivity of the test to context switch time by eliminating most unnecessary context switches.

6.3 Barrier Synchronization

Barrier synchronization is another important synchronization primitive. Mellor-Crummey and Scott [71] provide a good overview of different possible implementations of barriers in shared memory multiprocessors. A simple barrier uses a centralized shared counter. Each thread increments this counter as it arrives at the barrier, and then spins on a flag. When the last processor arrives at the barrier, it resets the flag, thus releasing all the spinning processors. This simple barrier works well for a small number of processors, but for larger numbers of processors causes a large amount of contention on the shared counter. More scalable algorithms distribute the barrier so that processors do not spin on a single variable. A software combining tree barrier [111] uses a k-ary tree structure, with the threads assigned to the leaves of the tree. Each group of k threads first perform a simple shared counter barrier. When the last thread in a group arrives at barrier, it proceeds up the tree and another simple barrier is performed for each group of k leaves. This continues on up the tree to the root, at which point all the processors have reached the barrier. To release the processors, the thread that arrives at the root resets the flag on which its children are spinning, and this propagates down the tree releasing all the threads. Other variations on this distributed, scalable barrier theme include the Mellor-Crummey and Scott tree barrier and tournament barriers [71]. These optimize the combining tree barrier idea, by including a shorter critical path through the tree, and by guaranteeing that spinning is done only on locally allocated variables.

We implemented a barrier benchmark using a shared memory combining tree. In this benchmark, a number of threads are spawned on each processor, and these threads repeatedly perform a barrier synchronization. The first level of the combining tree has a fan-in equal to the number of threads on each processor[1]. The threads on each processor first perform a local combine, and then the last thread to combine on each local processor participates in a global barrier using a radix-4 combining tree. The simulation uses 64 processors, with a fully associative cache, so that only cache invalidation traffic affects performance.

The following variations on the barrier benchmark are run:

- **Unprioritized**: Threads all have the same priority. When threads are waiting to be released they repeatedly poll the node in the combining tree at which they are stalled until it changes and releases them. A failed poll results in a context switch. If there are more threads than contexts, then there is also a thread swap with an unloaded thread.
- **Prioritized**: When a thread arrives at the barrier and it is not the last thread in the leaf group, it decreases its priority in preparation for the next phase of the computation, and begins to spin. The last thread to arrive at a leaf node maintains its priority and proceeds up the combining tree. Thus on each node, only the thread that is participating in the non-local barrier tree is actually using any cycles — it can either be spinning at an intermediate node of the combining tree, or it can be proceeding up or down the combining tree. Once a thread going back down the combining tree reaches a leaf of the tree, it decreases its priority to the priority of the other spinning leaf threads, and they can all proceed to the next phase of the computation. Note

[1] If there is only a single thread per processor, then this first level is eliminated.

that the prioritization required for other tree-like barriers including the tournament barriers and the MCS tree barriers [71], is qualitatively similar to the prioritization of the combining tree barrier.

One prioritization scheme that does not work very well is to increase the priority of the last thread that arrives at the barrier on each processor. This would allow a single thread on each processor to participate in the non-local portion of the combining tree without interference from the others. The problem with this scheme is that it does not differentiate between threads spinning at the barrier, and threads that are still doing useful work to get to the barrier. Making this differentiation is important to prevent spinning threads from stealing cycles from threads that have not yet reached the barrier.

6.3.1 Results

Figure 6.4 shows the average barrier wait times for the three different scenarios, **SINGLE**, **ALL**, and **LIMITED**, where the barrier wait time is the time spent by a thread waiting at the barrier. Figures 6.5 and 6.6 show the sensitivity of the benchmark to the thread swap time and the context switch time respectively.

Results show that the threads prioritization improves performance by eliminating unnecessary thread swaps and context switches. Threads critical to the completion of the barrier are given priority, and a minimum of time is spent on non-critical spinning threads. Eliminating unnecessary thread swaps and context switches also reduces the sensitivity to the thread save/restore time and to the context switch time. These results are discussed in detail below.

SINGLE

With unprioritized threads, performance of the barrier decreases as the number of threads increases due to two factors. First, each thread that participates in the barrier must swap into the context in order to reach the barrier. Second, when a thread that is spinning at an intermediate node of the combining tree does an unsuccessful poll, the scheduler swaps out this thread, and successively loads in all the other spinning threads on the local node. It does this because it does not differentiate between the locally spinning threads and thus treats them all fairly. In the prioritized case, the time to perform the barrier increases due to the larger number of threads, but once the local barrier has been completed and one thread has been chosen to represent the node in the global barrier, this thread never swaps out regardless of how often the polling is unsuccessful. As a result, the second component which contributed to poor performance in the unprioritized case is eliminated. For 4 threads per processor performance improves by 18%, and for 16 threads per processor performance improves 42%.

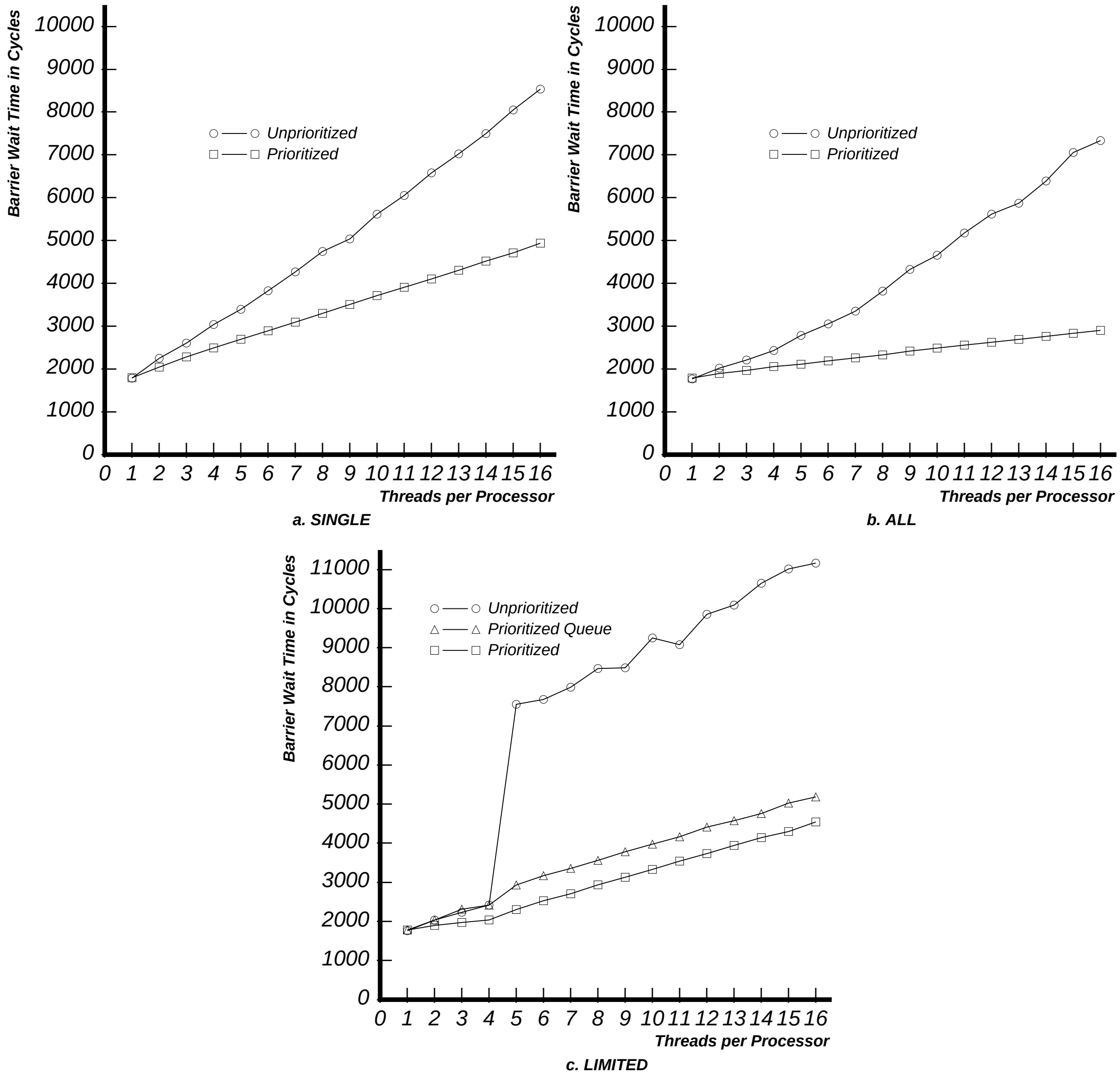


Figure 6.4: Average barrier wait time for 64 processors. **SINGLE**, **ALL**, and **LIMITED** scenarios. The *Prioritized Queue* case in the **LIMITED** scenario prioritizes the software scheduling queue, but does round-robin scheduling of the hardware contexts.

Note that for this benchmark we did not investigate 2-phase algorithms [48, 67] in which a thread spins a certain number of times, and then if it is still unsuccessful, suspends and is woken up later. Doing this makes the benchmark more complex because we must record which threads have to be woken up and which processors they are on, as well as signal these threads to wake up. Performance of this more complex approach is likely to be worse than the unprioritized case with a small number of threads, and better when there are a large number of threads and the thread swapping cost becomes more important. It will be worse than the prioritized case in all cases because of the extra overhead and the extra spinning.

ALL

The unprioritized **ALL** scenario suffers from a similar problem to the unprioritized **SINGLE** scenario, except that no thread swapping is necessary since all threads are loaded, only context switching. Although a context switch is much cheaper than a full thread swap, the context switches happen more often in the **ALL** case than thread swaps in the **SINGLE** case because they occur not only on failed synchronization tests, but also on cache misses. Each time the thread participating in the global barrier misses in the cache or does an unsuccessful polling operation, the processor runs through all the other contexts before returning to the critical context. It is important to note that the time to switch between the contexts is more than simply the number of cycles to switch between hardware contexts, in this case 5 cycles. This is because once the actual context switch takes place, the new thread issues some number of instructions, until it either misses in the cache, or tests its flag unsuccessfully and context switches. The prioritized scheduling eliminates unnecessary context switching during the global barrier with performance improving by 15% for 4 threads, and by 60% for 16 threads.

LIMITED

The case of having more threads than contexts with multiple contexts can potentially suffer from the worst of both the **SINGLE**, and the **ALL** scenarios. With unprioritized threads, each time a non-critical spinning thread runs, it not only checks its flag but also does a thread swap if there are other threads on the scheduling queue. Thus the time between when the critical thread context switches to the time it is again the executing thread is increased by the time to run through all the other loaded threads, where each is checking its flag and then swapping itself with some other spinning thread on the software scheduling queue. Note that this in effect represents a worst case scenario in terms of the amount of thread swapping that is done when threads are unprioritized. The more complex 2-phase algorithms mentioned previously will certainly perform better because they eliminate many unnecessary thread swaps, although they will again be worse than the prioritized case because of the extra complexity and extra spinning overhead. Figure 6.4c also shows the case when the thread priorities are used only by the software scheduler, and not the hardware scheduler. In this case the hardware scheduler does round-robin scheduling of the loaded threads rather than

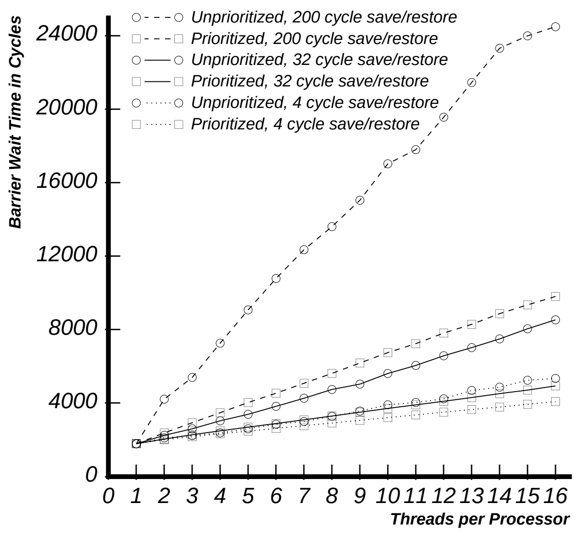


Figure 6.5: Average barrier wait time. **SINGLE** scenario with register save/restore times of 4, 32, and 200 cycles.

priority scheduling and thus does some amount of unnecessary context switching. With 16 threads, software thread prioritization without hardware thread prioritization reduces the barrier wait time by 54%, whereas doing both software and hardware thread prioritization reduces the wait time by 59%.

Sensitivity to Thread Swap Time

Figure 6.5 shows the runtime for the **SINGLE** scenario with different thread swap times. In addition to the curves shown previously for a register save/restore cost of 32 cycles, results are also shown for a save/restore time of 4 cycles and 200 cycles. Since both the prioritized and the unprioritized cases must do some thread swaps to perform the barrier, both are sensitive to the increase in thread swap time. However, since the unprioritized case does many unnecessary thread swaps, it is much more sensitive than the prioritized case. With 16 threads per processor, in going from a 4 cycle save/restore time to a 200 cycle save/restore time, the performance decreases by a factor of 4.6 in the unprioritized case, and by a factor of 2.4 in the prioritized case. As expected, the prioritization has a larger impact when the context switch cost is high.

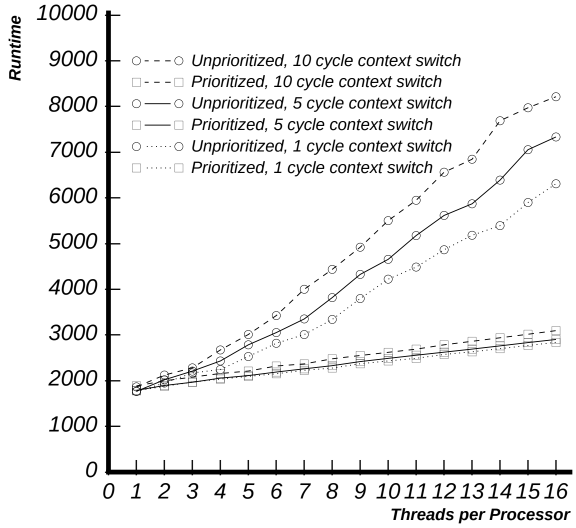


Figure 6.6: Average barrier wait time. **ALL** scenario with context switch times of 1, 5, and 10 cycles.

Sensitivity to Context Switch Time

Figure 6.6 shows the runtime for the **ALL** scenario with different context switch times. In addition to the curves shown previously for a context switch time of 5 cycles, results are also shown for context switch times of 1 and 10 cycles. From this figure we see that the unprioritized case is more sensitive to the increased context switch time than the prioritized case. In going from a 1 to 10 cycle context switch time the wait time for the unprioritized case increases by 30%, whereas for the prioritized case it increases only by 9%. This is as expected since the unprioritized case does many unnecessary context switches whereas the prioritized case does not.

Conclusion

These three scenarios show that the prioritizing of both loaded and unloaded threads is important for the performance of the barrier synchronization. Prioritizing unloaded threads in the thread queue is important because it guarantees that threads that still have to participate in the barrier are loaded, and it eliminates unnecessary thread swapping. Prioritizing the loaded threads themselves guarantees that lower priority spinning threads do not steal cycles from the higher priority threads, so that a critical thread can proceed as soon as it is able. Prioritizing threads also substantially reduces the effect of both increased thread swap time and increased context switch time.

6.4 Queue Locks

A queue lock is a mutual exclusion mechanism that is appropriate for high contention locks [71, 8]. Each thread inserts itself onto a queue, and then spins on its own flag so as to not generate the hot spots and excess network traffic that can be generated by simpler Test_and_Set style locks. Our implementation of queue locks is inspired by the MCS lock of Mellor-Crummey and Scott [71]. The MCS lock has the advantage that the flag on which each thread spins while waiting in the queue is locally allocated and generates no global traffic while the thread is spinning.

Scheduling threads waiting for a lock raises a number of issues. Once a lock is acquired, the thread owning the lock should not swap out [112, 67]. This is because all other threads waiting for the lock will be unable to make progress until the lock is released, and so performance can be seriously degraded. In the case of the queue lock, there is an additional factor to be considered. The order in which threads are going to acquire the lock is determined by the order in which threads are inserted into the queue lock. The priority of a spinning thread should reflect the position of the thread in the queue, so that when there are multiple spinning threads on a processor, the processor gives priority to the thread earlier in the queue.

The synthetic queue lock benchmark consists of an equal number of threads on each processor that are trying to obtain a lock. Each thread repeatedly obtains the lock, runs a critical section, releases the lock, and then runs a non-critical section. We consider both a high contention and a low contention case. In the high contention case, the critical section is about 100 cycles, and the non-critical section varies between 50 and 150 cycles, with a uniform distribution. In the low contention case, the critical section is the same, but the non-critical section varies between 5000 and 15000 cycles with a uniform distribution. The total number of locks acquired over a sample period of 10^6 cycles was taken as the figure of merit. It should be noted that the latency tolerance properties of having multiple threads, as well as possible cache interference between threads are not measured in this benchmark, but rather only the effects of the thread scheduling. The simulation uses 16 processors, with a fully associative cache so that only invalidation traffic occurs.

The following variations on the queue lock benchmark are run:

- **Priority1**: The threads repeatedly poll a local variable to determine if they are at the head of the queue. A failed poll results in a context switch. If there are more threads than contexts, then there is also a thread swap with an unloaded thread. When a thread acquires the lock, it increases its priority so that it does not swap out, and when it releases the lock it decreases its priority.
- **Priority2**: If a thread owns the lock or is trying to determine its position in the queue, it has the highest priority. If the thread is trying to insert itself into the queue,

it has the next highest priority[2]. Threads that are spinning in the queue have a priority based on their position in the queue[3]. Thus whenever a processor has several threads waiting in the queue, it will give priority to the thread that will next acquire the lock. Note that this requires the addition of a count field to the data structure in order to keep track of the position in the queue, and slightly more complicated lock acquisition code.

- **Signaling1**: In this version, after inserting itself into the queue a thread suspends itself. A suspended thread is put into a suspended thread data structure until it is explicitly woken up. When the thread currently owning the lock releases the lock, it sends a message to wake up the next thread. This requires that the location and the ID of each thread be available in the queue data structure. A thread owning the lock has high priority.
- **Signaling2**: This version combines signaling and spinning. When a thread first acquires the lock, it sends a message to the next thread in the queue to increase its priority. It releases the lock by writing the shared memory location on which the next thread in the queue is spinning. By doing this the next thread will get advanced warning that it is about to receive the lock, allowing the processor to load the thread if it is not already loaded.

6.4.1 Results

The results of the simulations for the three different scenarios, **SINGLE**, **ALL**, and **LIMITED** are shown in Figures 6.7, 6.8, and 6.9. Figures 6.10, 6.11, and 6.12 show the sensitivity of the benchmark to the thread swap time and the context swap time respectively.

The results show that prioritizing threads improves performance not only by giving high priority to a thread that owns the lock so that the critical section is executed quickly as in the TTSET benchmark, but also by making sure that the next thread in the lock's queue gets the lock quickly once it is released. The results also show that the sensitivity to thread save/restore time varies depending on the prioritization scheme. If the save/restore time is in the critical path between when the lock is released and when it is next acquired, then the results are sensitive to the save/restore time, otherwise they are not. The **Priority1** case is also sensitive to the context switch time because it can context switch many times before it reaches the next thread in the lock's queue. These results are discussed in more detail below.

[2]There is a subtle issue here having to do with a thread trying to release the lock while the next element in the queue is in the midst of inserting itself into the queue. The thread owning the lock has to drop its priority temporarily to allow the insertion to take place before it can release the lock.

[3]Note that the priority only needs to be calculated once during insertion into the queue, and does not have to be recomputed each time the lock is released.

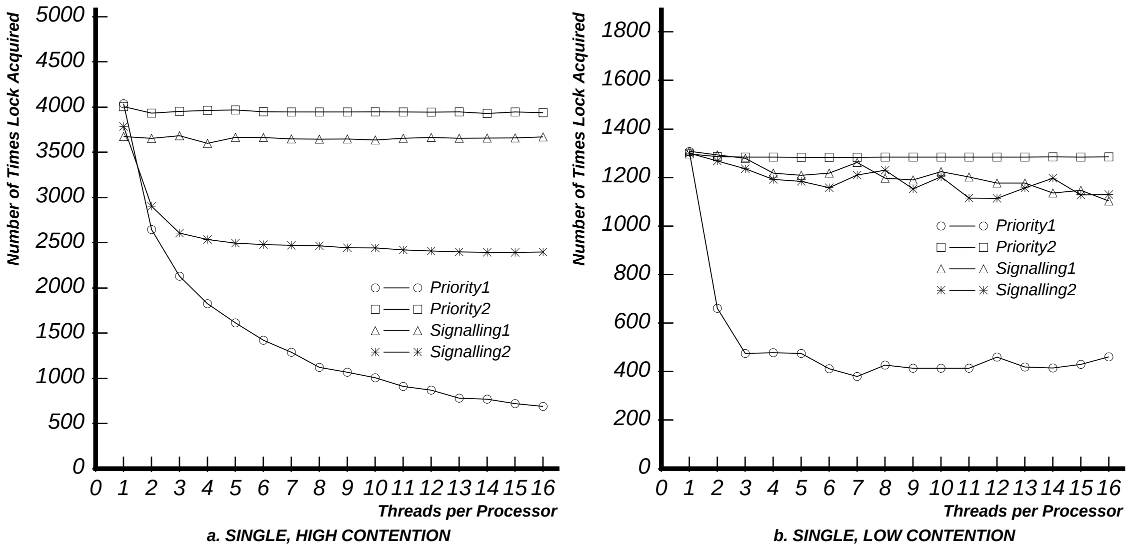


Figure 6.7: Queue Lock acquisitions. **SINGLE** scenario with high and low lock contention.

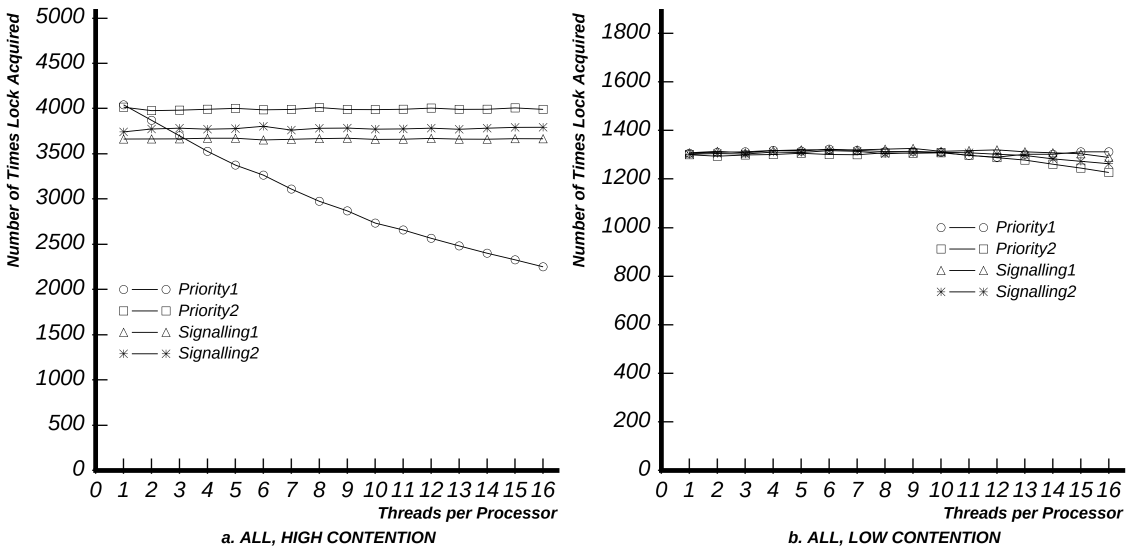


Figure 6.8: Queue Lock acquisitions. **ALL** scenario with high and low lock contention.

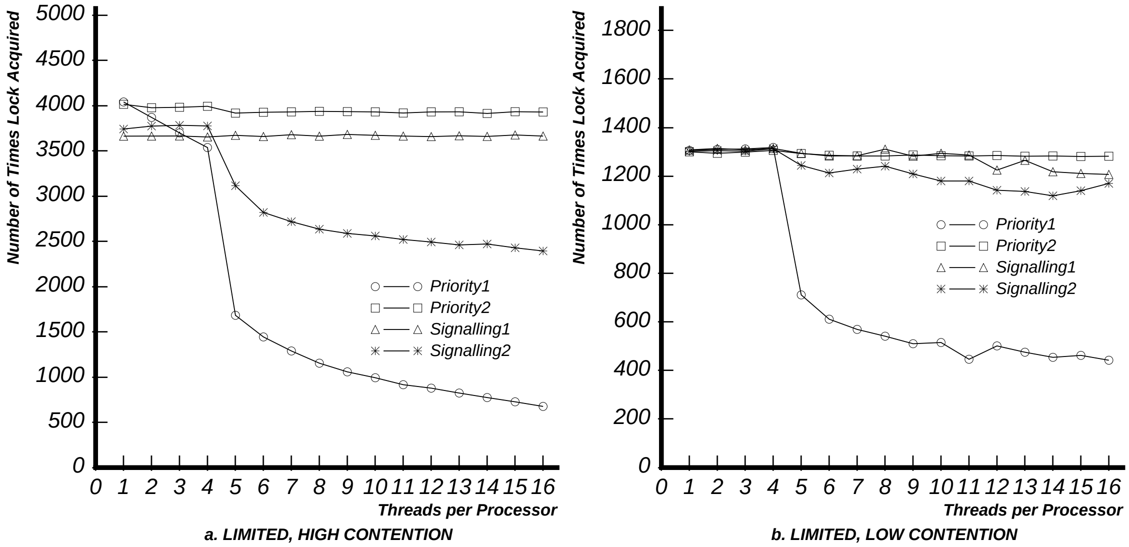


Figure 6.9: Queue Lock acquisitions. **LIMITED** scenario with high and low lock contention.

SINGLE

Figures 6.7a and 6.7b show the results for the **SINGLE** scenario. The main problem is making sure that the next thread to acquire the lock does so in a timely fashion. The **Priority1** scheme does not solve this problem because the next thread in the queue is still likely to be in the software scheduling queue of another processor and the next acquisition of the lock will be delayed until this thread is loaded. In both the high and low contention cases the performance drops significantly as the number of threads per processor goes from 1 to 16, by factors of 5.8 and 2.8 respectively. The **Priority2** scheme deals with this problem by prioritizing spinning threads such that their priority depends on their position in the queue. On any given processor, the next thread that is to acquire the lock is loaded and spins waiting for the lock to be released, which results in much better performance. For 16 threads, the **Priority2** case runtime performance is 83% and 64% better than the **Priority1** case, for the high and low contention cases respectively. The **Signaling1** scheme also performs consistently. Passing the lock requires a message send and the loading of the receiving thread's state into a hardware context. This can require a full thread swap if the receiving processor is already busy. It is never the case that the next thread is ready and waiting to acquire the lock. Because of this, **Signaling1** performs 8% worse than **Priority2** in the high contention case, and 14% worse in the low contention case when there are 16 threads per processor and passing the lock nearly always requires a full thread swap. **Signaling2** performance drops off as the number of threads increases because the overhead of passing a lock has now increased. The length of the critical section increases because the thread owning the lock must both send a message, and change the shared memory variable on which the remote thread is spinning. On the remote node, expensive re-scheduling operations take place based on the new priorities. In the low contention case the performance of **Signaling1**, and **Signaling2** become similar because both typically require a thread swap operation when the next thread is signaled to acquire the lock.

ALL

Figures 6.8a and 6.8b show the results for the **ALL** scenario. In the high contention case, **Priority1** performance suffers because of unnecessary context switching when it should wait for a critical reference to be satisfied. Performance drops by 44% in going from 1 to 16 threads per processor. **Priority2** performs best because it keeps all threads loaded and executes them in the correct order. **Signaling2** performs better than **Signaling1** because it never suspends a thread. In the low contention case with a low number of threads, performance is no longer dominated by the performance of the lock, and all 4 scenarios perform similarly up to 9 threads per processor.

LIMITED

Figures 6.9a and 6.9b show the results for the **LIMITED** scenario. In the high contention case, **Priority1** performance drops by a factor of 6 in going from 1 to 16 threads per processor, with a dramatic drop occurring once there are more threads than contexts due to all the thread swapping done on failed polling operations. The performance of the other three cases is much the same as in the **ALL** scenario when there are 4 or less threads per processor, and much the same as in the **SINGLE** scenario when there are more than 4 threads. Having more contexts helps the **Signaling1** scenario in the low contention case, because it is more likely that a context will be free when a thread's state has to be loaded in order for it to acquire the lock.

Sensitivity to Thread Swap Time

Figures 6.10 and 6.11 shows the runtime for the **SINGLE** and **ALL** scenarios respectively, using the high contention test with save/restore times of 4 and 200 cycles.

In the **SINGLE** scenario, a change in thread swap time influences all scenarios except **Priority2**. It does not affect **Priority2** because the thread that is going to next acquire the lock is always loaded well before the lock is released. The thread swap time has a large effect on the **Signaling1** scenario because threads always suspend when they enter the queue, and always have to be reloaded once they are next in line in the queue. When the save/restore cost is just 4 cycles it performs slightly better than the **Priority2** case, but when the save/restore cost is increased to 200 cycles, its performance is about 42%

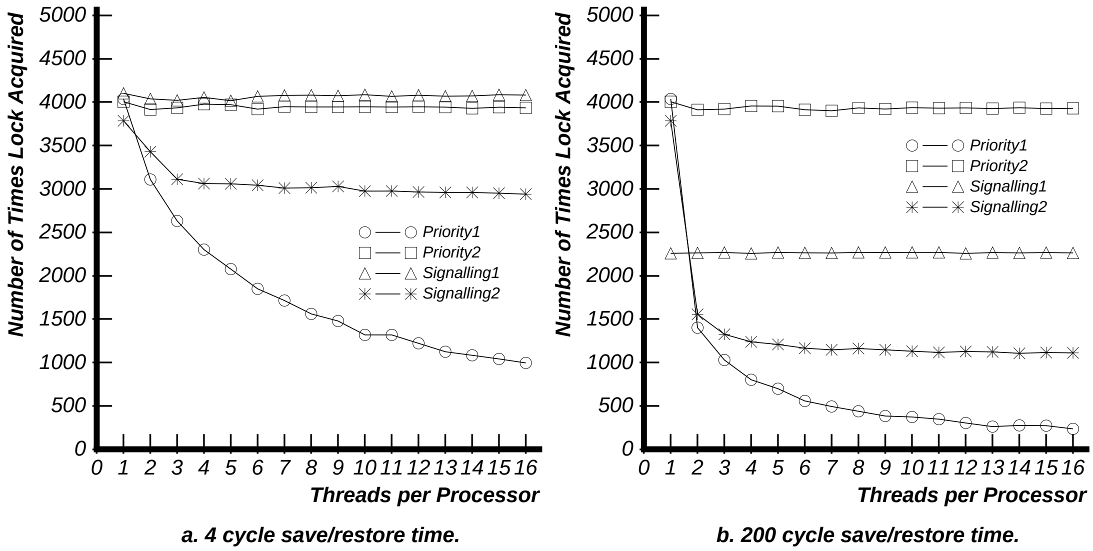


Figure 6.10: Queue lock acquisitions. **SINGLE** scenario with high contention and register save/restore times of 4 and 200 cycles.

worse than the **Priority2** case. The performance of **Priority1** and **Signaling2** suffers the most from the increased thread swap time. For 16 threads per processor, when the thread save/restore time increases from 4 to 200 cycles, performance drops by 76% and 62% respectively.

In the **ALL** scenario, only the **Signaling1** performance decreases with increased save/restore time since it is still suspends and wakes up threads. **Priority1** and **Priority2** and **Signaling2** never do a thread swap operation since there are enough contexts to have all the threads loaded.

Sensitivity to Context Switch Time

Figure 6.12 shows the runtime for the **ALL** using the high contention test with different context switch times. **Priority2**, **Signaling1**, and **Signaling2** all suffer from 3% to 7% when the context switch time goes from 1 to 10 cycles. This is because when the context switch time increases the penalty for cache misses and inter processor interrupts increases. The **Priority1** case suffers by as much as 14% because of the extra context switches that are done.

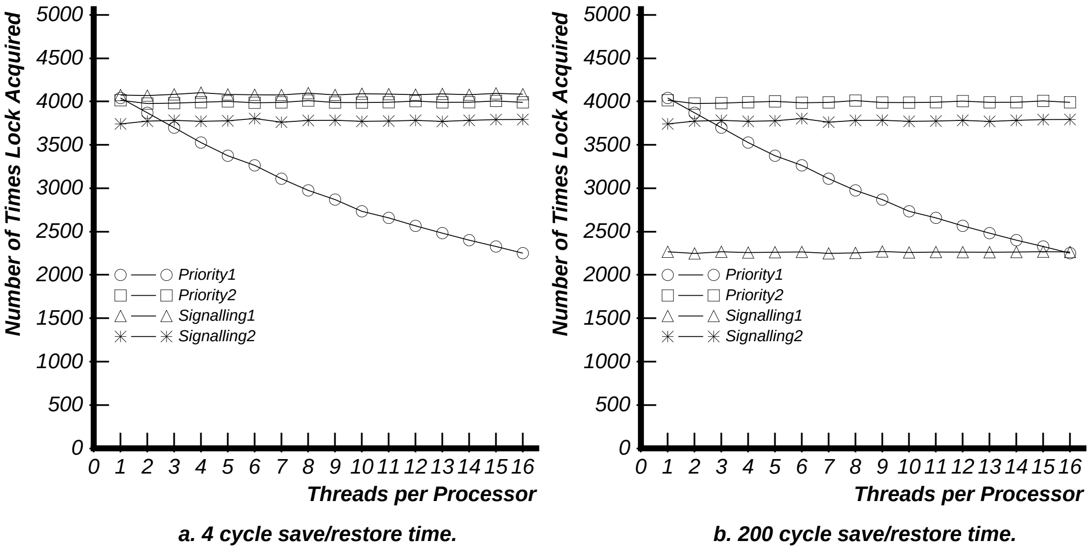


Figure 6.11: Queue lock acquisitions. **ALL** scenario with register save/restore times of 4 cycles and 200 cycles.

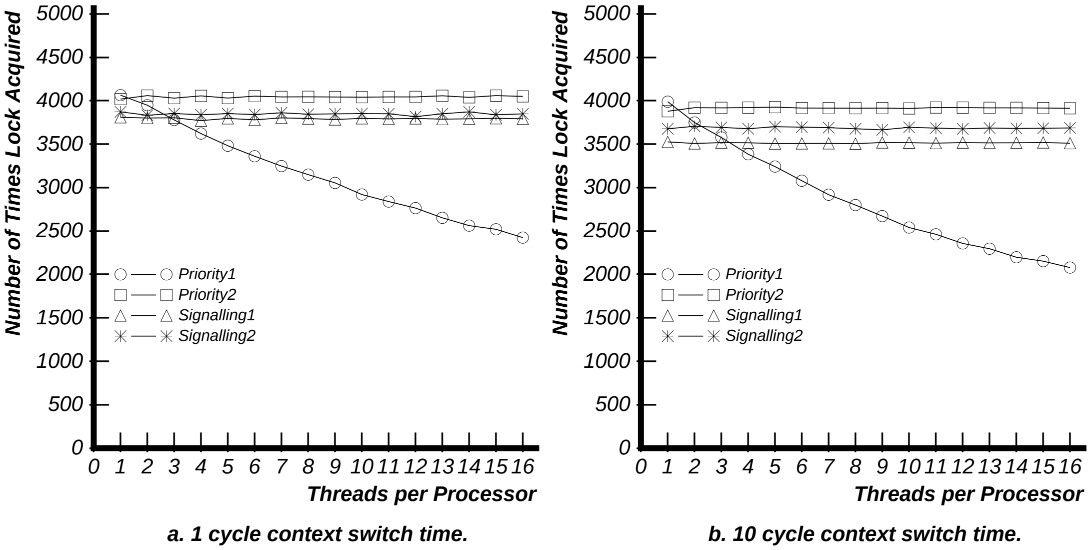


Figure 6.12: Queue lock acquisitions. **ALL** scenario with context switch times of 1 cycle and 10 cycles.

Conclusion

Several general observations can be drawn from this queue lock study. First, some sort of prioritization is helpful to make sure that threads acquire the lock in reasonable time, without having to resort to polling threads by continuously swapping them in and out of the loaded set to determine the next thread in the queue. Associating a priority with each thread based on its position in the queue is one approach to solving the problem. It allows the next thread that is to acquire the lock to be loaded and ready to accept the lock. Note however that determining a prioritizing of threads that is both correct and performs well is trickier than one would expect. Using a signaling mechanism to wake up the next thread in the queue is also a reasonable approach, but incurs the extra overhead of swapping threads in and out of contexts. Also, the relative performance of the three cases that prioritize effectively, **Priority2**, **Signaling1**, and **Signaling2**, depends on the assumptions made about thread swapping time, the length of the critical section, and the cost of message sends. In particular, if the cost of waking up a suspended thread is assumed to be higher than just the minimum time to load the context from the cache (due to cache misses for instance), the relative advantage of **Priority2** over **Signaling1** will increase. Also, if the cost of loading a thread is sufficiently high, and the length of the critical section is sufficiently long then the **Signaling2** will perform better than **Signaling1**.

6.5 Summary

In the context of multithreaded, multiple-context parallel processors, thread prioritization can be successfully used to prevent the performance of spin-waiting synchronization operations from degrading when the number of threads and/or the number of contexts is increased. Specifically, software thread prioritization that decides which threads should be loaded and which should be unloaded prevents critical threads from swapping out and becoming unloaded. Hardware prioritization of the loaded threads prevents spinning threads from needlessly consuming cycles, and allows critical threads to proceed as quickly as possible.

Three synthetic benchmarks were studied: A TTSET benchmark, a barrier synchronization benchmark, and a queue lock benchmark. In all cases performance suffered when threads were unprioritized, and did not suffer when threads were prioritized. Prioritization also reduced the sensitivity to the thread swap cost and the context switch cost because many unnecessary thread swaps and context switches are avoided.

Chapter 7

Scheduling for Good Cache Performance

Cache memory plays a key role in improving the performance of modern processors by significantly reducing the average latency of memory requests. The increasing ratio of processor speed to DRAM speed makes the cache even more critical, since loads and stores that miss in the cache require a larger number of cycles before they can be satisfied. Multiple-context processors allow the latency of cache misses to be tolerated, but can also lead to reduced cache hit rates due to *negative cache effects*: the different working sets interfere with each other causing misses that would not normally occur if only a single thread was executing [2, 105, 37, 81]. It is also possible for the threads in the multiple contexts to have significant overlap in their working sets, leading to *positive cache effects*. In this scenario, one context can make a reference that brings data into the cache, and when other loaded threads refer to the same data it is already in the cache. In this way different contexts prefetch data for each other. This data sharing also reduces invalidation misses since threads using the same data may be on the same processor.

Several studies conclude that negative cache effects dominate over positive cache effects [105, 37]. These studies however made little or no effort to schedule threads so that the working sets of the different running threads are overlapped. Thekkath and Eggers [96] studied the effect of thread placement on cache performance and runtime and conclude that sharing-based placement has no positive impact. One or more factors contributed to this being true: the threads accessed shared data in a sequential manner referring several times to the data before it is invalidated, the shared data was uniformly shared across the processors so that no placement of threads was clearly superior, and the shared data references were an insignificant part of the overall number of references. This study does not make any special effort to define threads in a way that will lead to positive cache effects, and does not attempt to closely coordinate their execution so that they are using shared data at the same time. Another study by Thekkath and Eggers [95] concludes that multiple-contexts are much more effective when the application has threads that have been optimized for locality. Again however, the threads are optimized individually, and not as a whole.

In this chapter we present a number of techniques for improving the cache performance of multiple-context parallel processors, in particular *data sharing* and *favored thread execution*. Data sharing makes the working sets of the loaded threads overlap as much as possible, by tightly coupling their execution so that they use much of the same data, at approximately the same time. This technique is particularly useful in the context of blocked algorithms where there is considerable sharing of data between loop iterations. Favored thread execution requires assigning a priority to the threads so that in the case that there are more contexts than necessary to tolerate latency, the processor favors the execution of high priority threads. This can lead to better cache behavior because the cache favors the working sets of the high priority threads.

We show a number of simple experiments that illustrate the benefits of data sharing and favored thread execution. These experiments concentrate on parallel loops for which cache performance is critical. We show that depending on how threads are defined and how work is distributed to the threads, they can have significant overlap of their working sets, or almost no overlap. We show that distributing work to the different contexts dynamically a single iteration at a time rather than statically provides good load balance between the contexts, and guarantees that latency tolerance is provided throughout most of the computation. Using favored thread execution, the minimum number of threads required to tolerate latency is selected to execute at any given time. This minimizes the hit rate degradation as the latency increases. We also show that though favored thread execution can improve hit rate, it can still in some cases have worse runtime than round-robin execution. This is caused by the load imbalance that occurs when some threads finish well ahead of others, leaving those few remaining threads without any means to tolerate latency. This effect can be minimized however by distributing work dynamically in small chunks. The results also show that favored execution has a much bigger effect on performance when the memory bandwidth is limited, because fewer threads are required to saturate the available bandwidth, and the penalty for additional cache misses is much higher.

Performance improvements depend on the number of contexts, the cache parameters, the memory latency, and the memory throughput. For the range of parameters and cache sizes simulated, multiple context versions of the benchmarks that use both techniques yield cache hit rates that are 25 to 50 percentage points higher than versions of the benchmarks that did not. Runtime improvement due to the improved cache performance depends on the memory throughput available. With high memory throughput the effect of low cache hit rate is not so important, and improvements range from almost none to about 16% with 16 contexts. If memory throughput is limited, than the improved cache performance has a big impact, with runtime improvements up to 50% with 16 contexts.

7.1 Data Sharing

Data sharing requires that the threads executing in the different contexts use common data, so as to maximize positive cache effects, and minimize negative cache effects. We will examine these effects principally in the context of parallel loop constructs. The key observation in this context is that the most data sharing typically occurs between successive iterations of a loop i.e. threads that are working on successive iterations of a loop are likely to share data. In the case of nested loops, there is typically one loop across which the most sharing occurs, and if different threads execute interleaved iterations of this loop then significant sharing will occur. The discussion of data sharing in the next section in the context of a blocked matrix multiply algorithm serves to clarify these points.

7.1.1 Blocked Algorithms

Blocked algorithms attempt to exploit data locality by operating on small blocks of data that fit into the cache, so that data loaded into the cache is reused [61, 109]. A simple example is matrix multiply that computes the matrix $Z = XY$, shown in unblocked form in Figure 7.1, and in blocked form in Figure 7.2.

To understand the effects of blocking, consider each loop of the code starting from the innermost loop. In the unblocked case, the only reuse over the iterations of the innermost **k** loop is the register allocated X array element. In the next innermost loop, the **j** loop, an entire row of the Z matrix is reused on each iteration. Provided there is sufficient room in the cache, this will lead to reuse of this data among the **j** loop iterations. If there is not enough room in the cache, then the entire Z row will have to be read in at each **j** iteration. In the outermost **i** loop, the entire NxN Y matrix is reused on each iteration. Provided the cache is large enough, there is the potential to reuse the Y matrix data. If the cache is not large enough, the entire Y matrix is likely to have to be read in at each **i** iteration.

Now consider the blocked case. The three innermost loops are the same as the unblocked case, except that we change the bounds so that for reuse in the **j** loop, only B (the blocking factor) elements of the Z row have to fit into the cache in order to have reuse, although it should be noted that due to the change in the number of **j** iterations, we use the data only B times rather than N times as in the unblocked case. Similarly, in the **i** loop, we reuse a BXB portion of the Y matrix N times provided it fits in the cache. Table 7.1 shows the amount of data the processor must bring into the cache for the blocked and unblocked algorithm, depending on the amount of reuse that can be exploited.

A couple of important notes should be made about this blocked algorithm. First, there is a tradeoff between data reuse, the blocking factor B, and loop overhead. To get the reuse, B must be chosen small enough so that the data fits in the cache, but large enough so that the data is reused as many times as possible, and so that the loop overhead is reduced.

```
for (i = 0 ; i < N ; i++)
  for (j = 0 ; j < N ; j++) {
   r = X[i][j]; /* register allocated */
   for (k = 0 ; k < N ; k++)
     Z[i][k] += r*Y[j][k];
  }
```

Figure 7.1: Straightforward matrix multiply code.

```
for (jj = 0 ; jj < N ; jj += B)
  for (kk = 0 ; kk < N ; kk += B)
    for(i = 0; i < N ; i++)
      for (j = jj ; j < MIN(jj+B,N) ; j++) {
        r = X[i][j]; /* register allocated */
        for (k = kk ; k < MIN(kk+B,N) ; k++)
          Z[i][k] += r*Y[j][k];
      }
```

Figure 7.2: Blocked matrix multiply code.

Reuse Pattern	Words Fetched Into Cache	
	Unblocked	Blocked
No data reused	$2N^3 + N^2$	$2N^3 + N^2$
Z data reused	$N^3 + 2N^2$	$N^3 + \frac{N^3}{B} + N^2$
Z and Y data reused	$3N^2$	$\frac{2N^3}{B} + N^2$

Table 7.1: Data that must be fetched into the cache depending on reuse patterns.

Lam, Rothberg, and Wolf [61] found that it is important to consider the amount of cache interference that occurs, which is highly sensitive to the matrix size, the blocking factor, and the stride. They found that the blocking factor B that leads to the least cache interference depends heavily on the matrix size and should be tailored accordingly.

7.1.2 Reuse Patterns in Blocked Algorithms

Wolf and Lam identify a number of different types of reuse in blocked algorithms [109]. These include:

1. self-temporal reuse: A reference in a loop accesses the same location in different iterations.
2. self-spatial reuse: A reference accesses the same cache line in different iterations.
3. group-temporal reuse: Different references access the same location.
4. group-spatial reuse: Different references access the same cache line.

Looking at the different references in the blocked matrix multiply code, and assuming that there are 2 double words per line, and that we store the arrays in row major order, we can identify the types of reuse that we can exploit for each reference. Z[i][k] has self-spatial reuse in the **k** loop since successive iterations use successive elements in the row. It has self-temporal reuse in the **j** loop because the same portion of the Z row is reused at each iteration. Similarly, Y[k][j] has self-spatial reuse in the **k** loop, and self-temporal reuse in the **i** loop. Finally, X[i][j] has self-spatial reuse in the **j** loop, and self-temporal reuse in the **kk** loop.

In a blocked algorithm, there is typically one loop in which the most reuse takes place. Consider the blocked matrix multiply loops beginning with the innermost loop, and with a blocking factor B. In the **k** loop, we can exploit only the self-spatial reuse of the Z and Y references. In the **j** loop, self-temporal reuse of the Z reference occurs as well. In the **i** loop, we can exploit the self-temporal reuse of the Y reference. Finally, each new iteration of the outermost loops **kk** and **jj** requires that we bring in a new set of data into the cache. Thus, assuming that the cache can hold a BxB Y matrix block, as well as B elements of the Z matrix, the loop in which most of the reuse occurs is the **i** loop. This fact can be used to maximize positive cache effects as described in the next section.

7.1.3 Loop Distribution to Achieve Positive Cache Effects

In order to exploit positive cache effects we want to have loop iterations that use common data execute in different contexts at the same time. In the matrix multiply example, the ideal loop to distribute across the multiple contexts is the **i** loop because these iterations are independent in the sense that they update (write) different sections of the Z matrix, and because threads executing different **i** iterations will share significant amounts of data in the cache. Since the **i** iterations are independent, several iterations of the **i** loop can proceed at once in different hardware contexts. To get good reuse the thread should hold the BxB Y matrix block that all the iterations are using, as well as B elements of the Z array for each context.

Distributing loops other than the **i** loop leads to a number of different problems. Distributing the **k** loop iterations to different contexts incurs a large amount of overhead because each iteration performs only a single multiply accumulate. Distributing the **j** loop has higher granularity but is problematic because different iterations update the same Z matrix locations, and some form of synchronization is required[1]. Distributing the **kk** or **jj** loops to the different contexts means that the different contexts have no overlap at all between their working sets. Each context requires a separate BxB Y matrix block, and a separate set of B elements of the Z array. This can lead to significant negative cache effects, and a resulting degradation in performance.

Efficient Local Loop Distribution

The simple technique we use for distributing multiple iterations to different contexts on a single processor is to have each thread running in a context dynamically acquire the next iteration by using an atomic Fetch-and-Increment instruction. Distributing the loop iterations one at a time leads to several good effects in the multiple-context processor:

- Different contexts are working on closely spaced iterations that tend to share data. As will be shown in the benchmarks later in this chapter, this can lead to better spatial and temporal locality in the cache as compared to schemes where the threads are not working on closely spaced iterations at the same time.
- Distributing work a single iteration at a time means that contexts will be load balanced and provide maximum latency tolerance for each other. If work is distributed statically in big chunks, there can be a load imbalance if some threads have lots of work to do, while others have very little.

Our implementation of local loop distribution to multiple-contexts is similar to a multiple processor loop distribution scheme described by Weiss, Morgan, and Fang [106]. The loop is scheduled using a shared structure that contains two values, one shared counter for acquiring an iteration, and one shared counter for signaling the end of an iteration. A thread acquires an iteration by atomically incrementing the acquire variable, executes the iteration, and

[1] This could be accomplished with Full/Empty bits, locks, or a floating point Fetch-and-Add instruction.

then signals the end of the iteration by incrementing the end counter. An easy optimization to this scheme counts the number of contexts that have completed, rather than the number of iterations that have completed: once there are no more iterations to acquire, each context increments a counter to signal that it has finished execution. This effectively represents a software barrier performed by all the threads. If the loop being distributed across the contexts is contained within other loops, then the last processor updates the outer loop indices, resets the acquire and the end counters, before releasing the barrier.

To give a concrete example, the case of the matrix multiply would work as follows. Each context atomically increments a shared variable to determine the **i** iteration that it is to execute. If no more **i** iterations are left, the thread atomically increments the end counter, and enters a local barrier. The last thread to finish an **i** iteration updates the **jj** and **kk** iteration variables in preparation for the next **i** loop, resets the shared counters, and then releases the barrier. Note that hardware prioritization of threads plays an important role in optimizing this local barrier: when there are no more iterations to be executed, a thread arrives at the barrier and drops its priority, so that threads that are spinning at the barrier do not steal cycles from threads that are still executing iterations.

Comparison to Multiprocessor Loop Distribution Techniques

The problem of distributing iterations to the threads running in the contexts of a multiple-context processor has a number of similarities and differences with the more general problem of distributing loops to multiple processors that have been discussed in the literature [79, 68, 99, 70]. The similarities are that the distribution can be done statically or dynamically, and there can be a load imbalance problem. The major difference is that fine-grain loop distribution is much cheaper with multiple contexts that share a cache, than with multiple contexts that do not.

Multiprocessor loop distribution studies compare static loop distribution in which iterations are statically assigned to multiple processors, and dynamic distribution using a shared iteration counter. All the studies of multiprocessor loop distribution lead to the same basic results:

- The static distribution of loops can lead to serious load imbalance when the amount of work in each iteration is unknown or variable. Dynamic distribution schemes achieve much better load balance.
- Dynamic distribution schemes suffer from a bottleneck on the iteration variable as the number of processors increases. To relieve this bottleneck various schemes are possible. More than a single iteration can be obtained at each iteration variable access [79, 68, 99]. Deciding how many iterations should be acquired requires a tradeoff between reducing the contention on the iteration variable, and achieving good load balance between the processors. Alternatively, one can start by statically allocating

iterations to processors and then do load balancing if a load imbalance occurs [70].

Similarly to the distribution of loops to multiprocessors, the distribution of loops to multiple contexts can be done either statically or dynamically. A static distribution to the multiple contexts can lead to a load imbalance. This imbalance can cause threads with less work to finish well before others, so that not all the contexts will be running at the same time. Although the processor will never be completely idle until all the contexts have finished executing their work, there can be significant periods of time where there are not enough contexts running to completely tolerate latency.

The main difference between distributing iterations to multiple processors and distributing iterations to multiple contexts is the cost of dynamic loop distribution: it is much cheaper to use a Fetch-and-Increment instruction to acquire an iteration in the multiple-context case than in the multiprocessor case. This is because all the contexts share a cache and no global operations have to take place to invalidate remote copies of the iteration variable. Typically the iteration variable will always be available in the cache. Complicated schemes to avoid contention on iteration variables are not needed, and further this fine-grain distribution of threads has the data sharing advantages mentioned in the previous section.

Note that the loop distribution to multiple contexts we have described is done at a purely local level, and thus requires a method for distributing work at a global level to the multiple processors. In some cases we can distribute the global work statically. Alternatively, we can distribute a loop to the processors using any one of the multiple processor loop distribution schemes, and then redistribute iterations locally to the multiple contexts using the local scheme.

7.1.4 Data Prefetching and Data Pipelining Effects

In addition to the reuse of data that occurs when multiple contexts share data, two other effects can impact performance. First, a reference in one context often acts as a prefetching mechanism for another context. This can increase the cache hit rate and decrease the latency penalty. This is a form of implicit prefetching, which avoids the complications and overhead associated with doing explicit prefetching.

Second, even if the working set of data does not fit in the cache, data can be pipelined through the cache and be used by each context as it executes a single iteration. Consider again the blocked matrix multiply example, and consider a blocking factor B for which the BxB portion of the Y matrix does *not* fit in the cache. The different iterations running in the different contexts all request the Y data in the same order. Elements of the Y block are brought gradually into the cache, and are used by each context in turn. We achieve a reuse factor of C, where C is the number of contexts, where normally we would not achieve any reuse because the data block was too big. These effects are observable in the matrix multiply example as described next.

16 X 16				
Cache Size	Single Context		Four Contexts	
	Speedup	HR	Speedup	HR
64 bytes	1.00	0.65	1.35	0.46
512 bytes	1.17	0.77	1.55	0.69
1 Kbytes	1.22	0.80	1.74	0.89
2 Kbytes	1.24	0.81	1.77	0.92
4 Kbytes	1.65	0.96	1.81	0.96

Table 7.2: 16X16 single processor matrix multiply using a fully-associative cache. Speedup is given relative to the single context case with a 64 byte cache.

Example Hit Rates for Matrix Multiply

We can easily calculate analytically the effect of data reuse on the hit rate for the matrix multiply example. We consider a fully-associative cache, and assume two double words per cache line, a 16x16 block, and consider only the matrix data references. The worst hit rate, assuming that we exploit only the temporal and spatial locality in one cache line, is 65%. If we exploit the self-temporal locality of the Z references, then the hit rate increases to 81%. The best hit rate is when we exploit the self-temporal locality of both the Z and the Y references, and is 97%. Finally, if we assume we can exploit the self-temporal locality of the Z references but not the self-temporal locality of the Y references, and that the **i** loop is being distributed to 4 contexts and that these contexts are exploiting the data pipelining effect (i.e each Y value is used by all 4 contexts before being removed from the cache), then we calculate the hit rate to be 93%.

Table 7.2 shows the results for a 16x16 matrix multiply for 1 and 4 contexts, with various cache sizes. In this case it is the **i** loop that is being distributed across the contexts. Initially, for small cache sizes, negative cache effects dominates, and the cache hit rate for the multiple context case is worse than the single context case. Note that performance is still better for the multiple context case, because of the latency tolerance provided by the multiple contexts. For moderate size caches, the cache hit rate is actually better for the multiple context version of the code than the single context version due to the data pipelining that occurs. With a large cache the hit rate of both examples are very similar, and because the hit rate is so high in both cases, the benefits provided by the latency tolerance of the cache are small.

7.2 Favored Thread Execution

Favored thread execution requires prioritizing threads so that they are executed preferentially in a certain order. The main benefit of this prioritization is that the cache will have a tendency to contain more of the working set of the higher priority threads, and less of the working set of the lower priority threads. Consider for instance the case where there are 4 threads executing in 4 contexts. If only 2 and sometimes 3 threads are necessary to hide the processor latency, and the cache is not big enough to contain the working sets of all 4 threads, then the prioritization can make it so that the working set of the 2 high priority threads are loaded, some portion of the third thread's working set is loaded, and virtually none of the working set of the fourth, and lowest priority thread.

The big advantage of this scheme is that the number of loaded threads that are using processor cycles can be dynamically chosen to be the minimum required to tolerate the observed latency. If the average latency increases, lower priority threads will begin executing to tolerate the additional latency. If the average latency decreases, then some of the lower priority threads will stop receiving processor cycles. Choosing the minimum number of loaded threads to execute means that the minimum required number of working sets will be in the cache, leading to overall better hit rates.

There are also a number of disadvantages of prioritizing threads in this way including:

- Insufficient loaded threads: Favored thread execution is most effective when there are more than enough contexts to fully tolerate latency. If there are not, then all the contexts will be executing in an attempt to tolerate the long latencies, and so all their working sets will want to be loaded. The only effective way of influencing the cache ratio in this case may be to limit the number of loaded threads.
- Load balancing: Favored thread execution may exacerbate the load balancing problem mentioned in the previous section. Consider for example the case that 3 contexts are required to tolerate latency, and 4 threads are created with decreasing priority. The lowest priority will not begin to execute a significant number of cycles until at least one of the other threads completes. If all the three other threads finish at approximately the same time, the one remaining thread will find itself executing alone, without any other threads available to tolerate latency.

We can solve the insufficient loaded thread problem and the load balancing problem by distributing work in small enough chunks so that many threads can be loaded, and no one thread has an unduly large piece of work that will cause it to run a long time beyond when other threads have no more work. Distributing work in small chunks has extra overhead, which is usually compensated for by better cache performance and better latency tolerance.

7.3 Experiments

In this section we quantify the effects of data sharing and favored execution by examining in detail three benchmarks: matrix multiply, Successive Over-Relaxation (SOR), and sparse-matrix vector multiply. We wrote different versions of each code that show different levels of data sharing and favored execution, and simulate them on a single processor with multiple contexts, varying the latency and memory bandwidth.

The results confirm that defining threads so that they share data in the cache, and closely coordinating their execution, leads to better cache hit rates. Runtime performance improves as well, especially when memory bandwidth is limited and cache performance is critical. Favored execution can also improve performance especially when latencies are short, memory throughput is low, and the data sets of threads do not share much data. When these conditions hold, favored execution is the most successful in reducing the amount of data that has to be in the cache. However when latencies are long and work is statically distributed in big chunks, favored execution can make the load balancing problem worse by causing high priority threads to finish early, leaving low priority threads without any means of tolerating the long latencies.

For these benchmarks the default memory latency is 20 cycles, and the default memory throughput is 1 request every 4 cycles. We simulate the long latencies that might occur with slower DRAM and in multiple processors by increasing the memory latency of the single processor to 80 and 160 cycles. We also vary the memory throughput available, and consider what happens when the memory throughput is a much lower 1 request per 20 cycles.

7.3.1 Matrix Multiply

We simulate the following versions of the matrix multiply benchmark:

1. **mm**: Single context version that runs the straightforward blocked matrix multiply code.
2. **mm_i**: Multiple context version that does fine-grain distribution of the **i** loop across multiple contexts.
3. **mm_kk**: Multiple context version that does fine-grain distribution of the **kk** loop across multiple contexts. Note that in this case, essentially no data reuse is exploited between the different contexts. When this is done for large block sizes, there can be fewer iterations than contexts.
4. **mm_kk_i**: Multiple context version that does fine-grain distribution of both the **kk** and the **i** loops as a unit by coalescing the loops [79]. A single shared counter is incremented, and the value of **kk** and **i** are computed from the result. The goal of doing this is to reduce the synchronization overhead due to synchronizing the different contexts at the end of each group of **i** iterations.

Data Sharing Effects

Data sharing between contexts varies depending on the version of the code being run, and on the block and cache sizes. Figure 7.3 show the results for a blocked matrix multiply with a matrix size of 36, a blocking factor of 9, and a range of cache parameters. The multiple context versions of the benchmark all use 4 contexts. All the threads that use multiple contexts perform better than the single context case due to the latency tolerance provided. The cache hit rates for both **mm_i** and **mm_kk_i** are about the same as for **mm** despite the larger aggregate working set size because of the sharing of data between contexts. They are a bit worse for the direct-mapped cache where there is more interference between references, and a bit better for a fully-associative cache where there is less interference and the pipelining and prefetching are more effective. The differences in performance between **mm_i** and **mm_kk_i** are minor, with the extra cost of synchronization in **mm_i** offset by the extra work required to calculate **i** and **kk** from the coalesced loop index in **mm_kk_i**. The **mm_kk** benchmark which only distributes the **kk** loop has considerably worse cache performance for smaller cache sizes because the working sets of the 4 threads do not share any data. This lower hit rate translates to lower performance due to the extra context switching. However, once the cache is large enough, the hit rates for all the cases are about equal.

Systematic interference can sometimes be a problem, particularly with direct-mapped caches. Due to the relationship between the cache size, the matrix size, and the blocking factor, many values in the block map to the same cache lines so that different references within a block systematically interfere with each other [61]. For instance, Figure 7.4 shows the results for a blocked matrix multiply with a matrix size of 32 and a blocking factor of 8. In all cases the direct-mapped hit rates are less than for the 36X36 matrix using a blocking factor of 9. The problem is worse in the **mm_i** and **mm_kk_i** cases because they have several references outstanding to the same block. When the cache is fully-associative the hit rates show similar behavior to the 36X36 matrix with a blocking factor of 9. In general, we must choose blocking factors carefully to avoid this systematic cache interference [61].

In general, assuming the absence of systematic cache interference, the considerable overlap in the working sets of different iterations in the case of **mm_i** and **mm_kk** cause them to have comparable cache hit rates to the single context **mm** example, and better performance than the **mm_kk** scenario in which their is little overlap of the threads working sets.

Favored Thread Execution Effects

Using favored thread execution rather than round-robin execution of contexts improves cache hit rates and performance in the case that there are more than enough contexts to tolerate the observed latency. Figure 7.5 shows the runtime and the cache hit rate for **mm_kk_i** as the number of loaded threads increases, using a 36X36 matrix size, a blocking factor of 9, and different memory latencies and memory system throughputs. Two

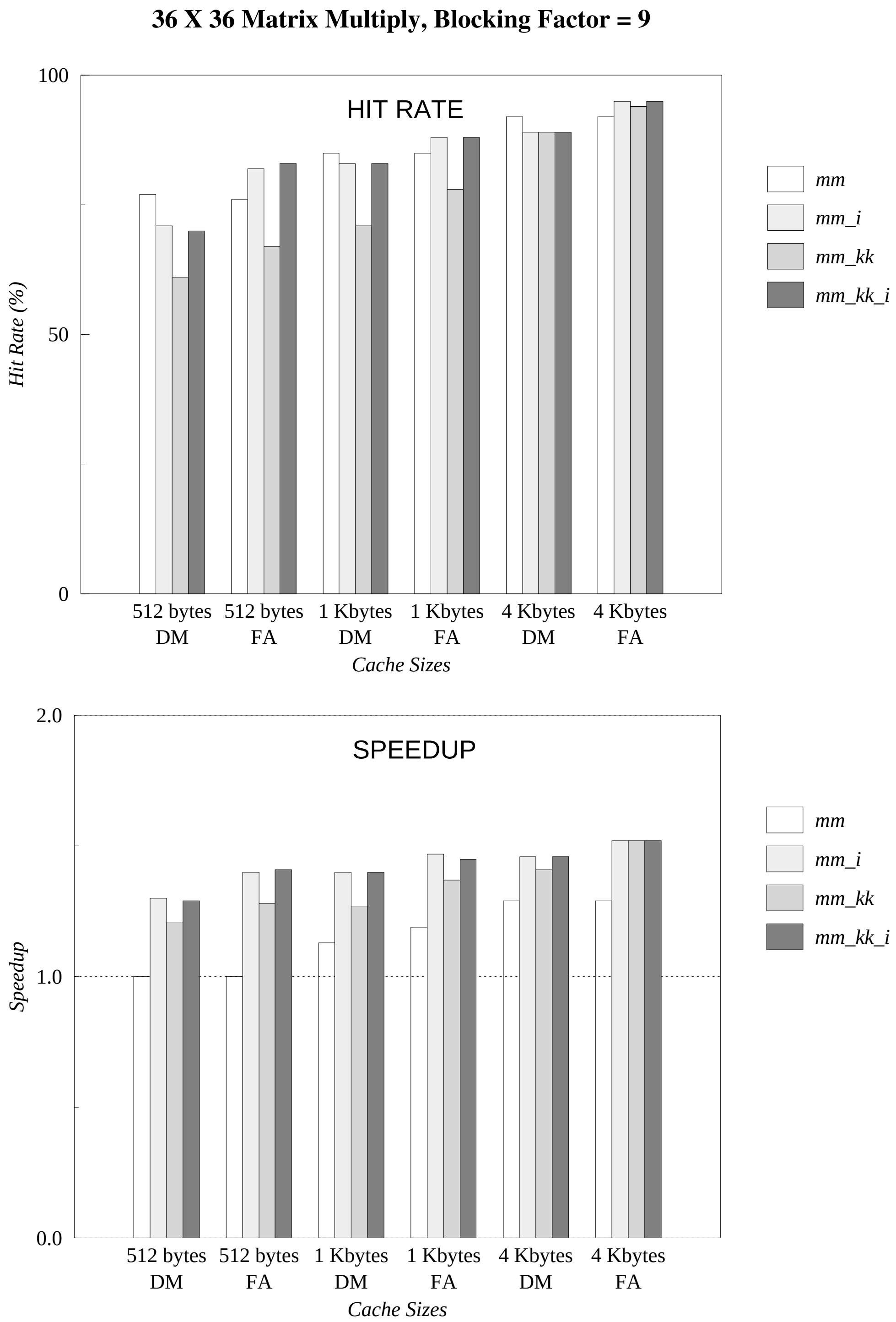


Figure 7.3: Hit rates and speedups for a 36X36 matrix multiply with a blocking factor of 9. Multiple-context versions of the code use 4 contexts. Fully-associative (FA) and direct-mapped (DM) caches are simulated.

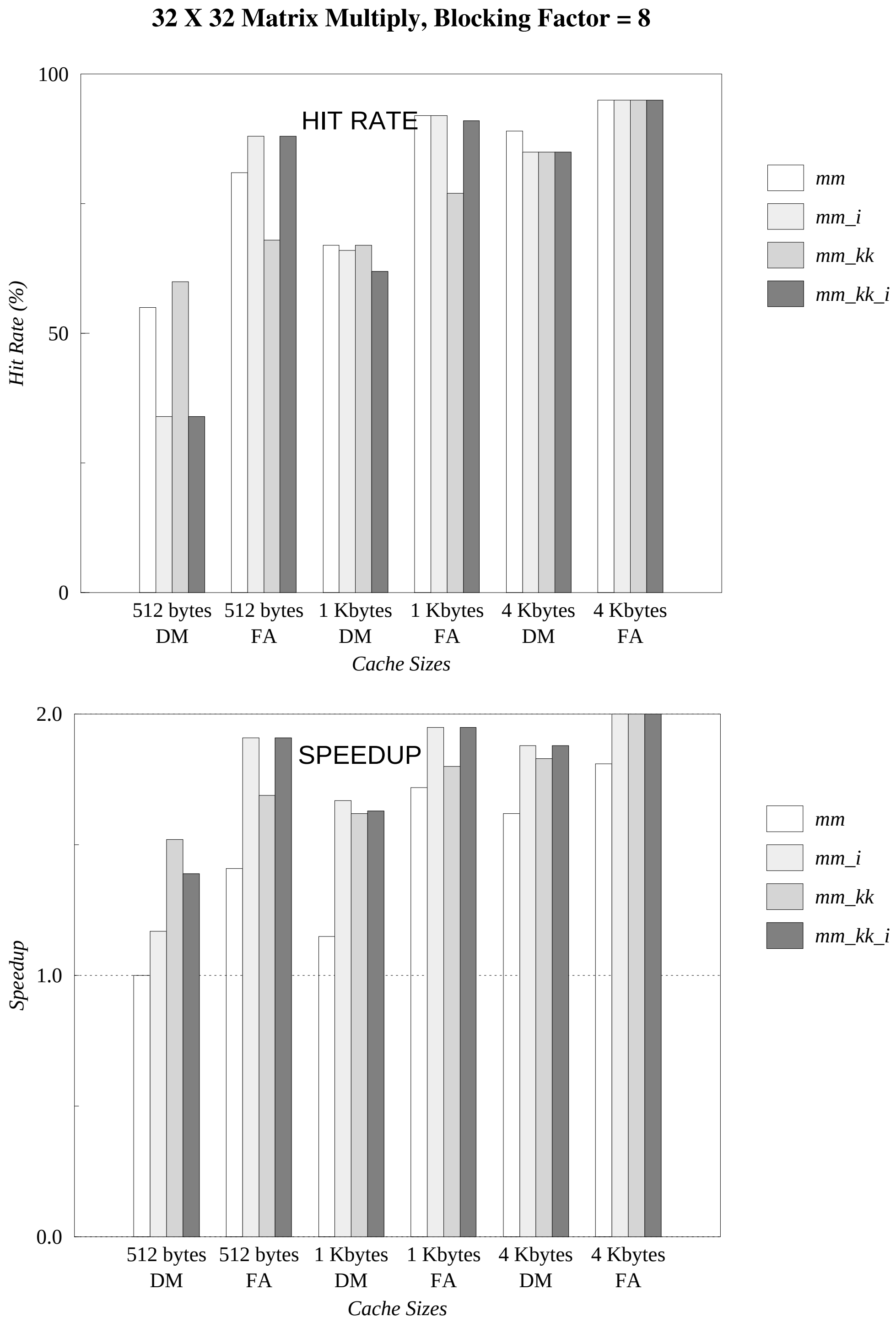


Figure 7.4: Hit rates and speedups for a 32X32 matrix multiply with a blocking factor of 8. Multiple-context versions of the code use 4 contexts. Fully-associative (FA) and direct-mapped caches (DM) are simulated. In the DM case, systematic cache interference leads to poor hit rates.

scheduling schemes are shown, one in which the multiple threads have equal priority, so that threads run in round-robin fashion, and one in which threads have a priority based on the iteration that they are calculating, so that they execute in a favored order.

With round-robin scheduling the cache hit rate decreases uniformly as the number of loaded threads increases independent of the latency. With the favored thread scheduling the hit rate decreases, but eventually reaches a stable value. This is because the favored execution dynamically chooses the minimum required number of contexts to tolerate the observed latency, and stabilizes the hit rate at the value corresponding to this number of contexts executing at once.

The impact of the improved hit rate is small for both high and low memory throughput because the hit rates are high, leading to long run lengths and an application that is not bandwidth limited. The only performance penalty is the cost of the extra context switching. The largest performance difference occurs with a memory latency of 20 cycles and 16 contexts. The favored execution scheduling increases the hit rate from 65% to 82%, and leads to a 9% reduction in runtime.

7.3.2 SOR

Figures 7.6 and 7.7 show an unblocked version and a blocked version respectively, of code for 2D red/black successive over-relaxation (SOR). This SOR code divides the domain into red and black points layed out in a checkerboard pattern, and at each iteration updates first the red points, and then the black points [29].

Looking at the different references in the SOR code, and assuming that there are 2 double words per cache line, and that we store the arrays in row major order, we can identify the types of reuse exploited by each reference. A[i][j], A[i][j-1], and A[i][j+1] have group-temporal and group-spatial reuse within the **j** loop, such that each iteration requires only one new element from the **i** row. All the references have group-temporal reuse within the **i** loop, since we use values from the current row in the calculation of the next row.

We simulate three versions of the benchmark:

1. **sor_sing**: Single context version of the blocked code.
2. **sor_dyn**: Multiple context version with fine-grain distribution of the **i** loop to the different contexts.
3. **sor_sta**: Multiple context version with static distribution of blocks of rows to the different threads. The code does blocking within each thread if there are sufficient rows.

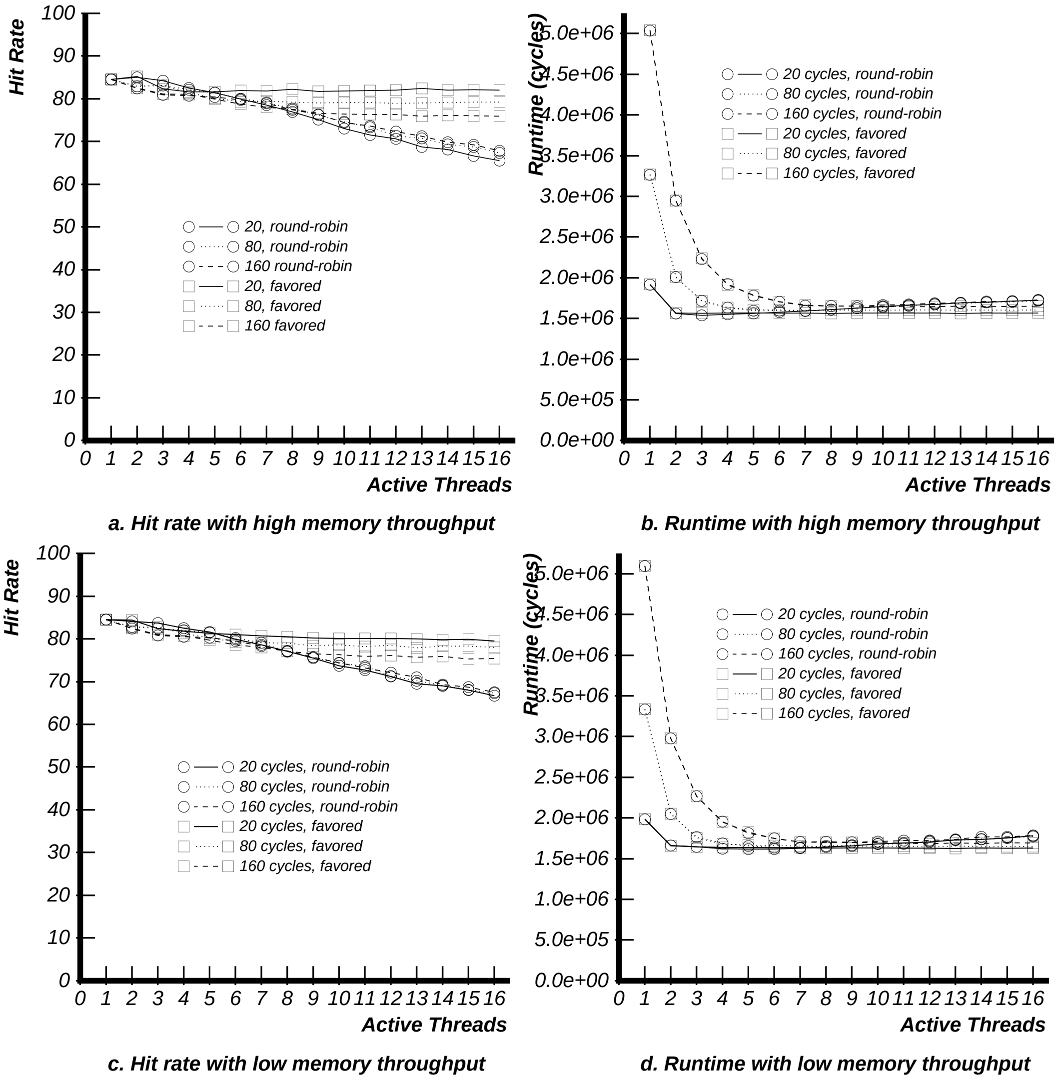


Figure 7.5: Performance of **mm_kk_i** comparing round-robin and favored execution for different memory latencies and throughputs. A 36x36 matrix multiply is done with a blocking factor of 9 and a 1Kbyte direct-mapped cache.

```
for (t = 0 ; t < T ; t++) {
 for (i = 1 ; i < N ; i++)
  for (j = (odd(i) ? 1 : 2) ; j <= N-1 ; j += 2) {
    A[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][j+1] +
                     A[i-1][j] + A[i+1][j]);
  }
 for (i = 1 ; i < N ; i++)
  for (j = (even(i) ? 1 : 2) ; j < N-1 ; j += 2) {
    A[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][j+1] +
                     A[i-1][j] + A[i+1][j]);
  }
}
```

Figure 7.6: Straightforward 2D red/black SOR code.

```
for (t = 0 ; t < T ; t++) {
 for (ii = 1 ; ii < N ; ii += B)
  for (jj = 1 ; jj < N ; jj += B)
   for (i = ii ; i < MIN(ii+B,N) ; i++)
     for (j = (odd(i) ? (jj):(jj+1)); j < MIN(jj+B,N-1) ; j+=2) {
       A[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][j+1] +
                        A[i-1][j] + A[i+1][j]);
     }
 for (ii = 1 ; ii < N ; ii += B)
  for (jj = 1 ; jj < N ; jj += B)
   for (i = ii ; i < MIN(ii+B,N) ; i++)
     for (j = (even(i) ? (jj):(jj+1)); j < MIN(jj+B,N-1) ; j+=2) {
       A[i,j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][j+1] +
                       A[i-1][j] + A[i+1][j]);
     }
}
```

Figure 7.7: Blocked 2D red/black SOR Code.

Data Sharing Effects

Figure 7.8 shows the performance of the different versions for a variety of blocking factors. In the case that there is no blocking (blocking factor = 80), the performance of **sor_sing** suffers due to poor data reuse. For each **i** iteration data from 3 rows is needed, and if these rows do not fit in the cache, there will be little reuse on the next **i** iteration. **sor_dyn** cache performance is much better in this case, due to data pipelining. Four threads progress at once and share their row data, giving a hit rate of 74% as opposed to 53% for **sor_sing**. As the blocking factor decreases, the cache hit rate of **sor_sing** and **sor_dyn** increase with the best performance at a blocking factor of 10 or 20 when most of the shared row data between **i** iterations is reused. The cache hit rate of **sor_sta** suffers because of the disjoint working sets. As a result, the hit rate increases uniformly as the blocking factor decreases and the working set sizes of the loaded threads decrease.

It is interesting to note that for the relatively small latency of 20 cycles, the best performing cases are not the ones with the best hit rate. This is due to the increased overhead that occurs with smaller block sizes. This is particularly bad in the **sor_dyn** case, since a barrier is performed after each block is calculated. The overhead can be estimated by removing the calculations from the innermost loop and rerunning the code. For instance, at a blocking factor of 80 the overhead corresponds to about 17% of the computation in the **sor_dyn** case. At a blocking factor of 5, overhead has increased by a factor of 3.7 and corresponds to 42% of the computation time. By contrast, overhead increases by only a factor of 1.7 over the same range of blocking factors for both **sor_sing** and **sor_sta**. At a blocking factor of 5, **sor_sta** performs better than **sor_dyn** because of less overhead, despite having a smaller hit rate. As the latency increases and the memory bandwidth is restricted, the effect of the hit rate becomes more important on performance, and the tradeoff of decreasing overhead versus improving hit rate will change.

Favored Thread Execution Effects

The effect of favored execution on the SOR cache hit rate and runtime varies depending on how data is divided up between the threads. It has a small effect on the cache hit rate when the threads are working on common data, and a large effect when threads are working on disjoint data. The effect on the runtime varies depending on whether the application is bandwidth limited or not. Figures 7.9 and 7.10 show the runtime and the cache hit rate for **sor_dyn** and **sor_sta** respectively, as the number of loaded threads increases, using a blocking factor of 20 and different memory latencies and throughputs. Two scheduling schemes are shown, one in which the multiple threads have equal priority, so that thread are scheduled in round-robin fashion, and one in which threads have a priority based on the iteration that they are calculating, so that threads are executed in a favored order.

For the **sor_dyn** case shown in Figure 7.9, favored thread execution has a small effect on the hit rate and on the runtime, compared to the round-robin execution. This is because

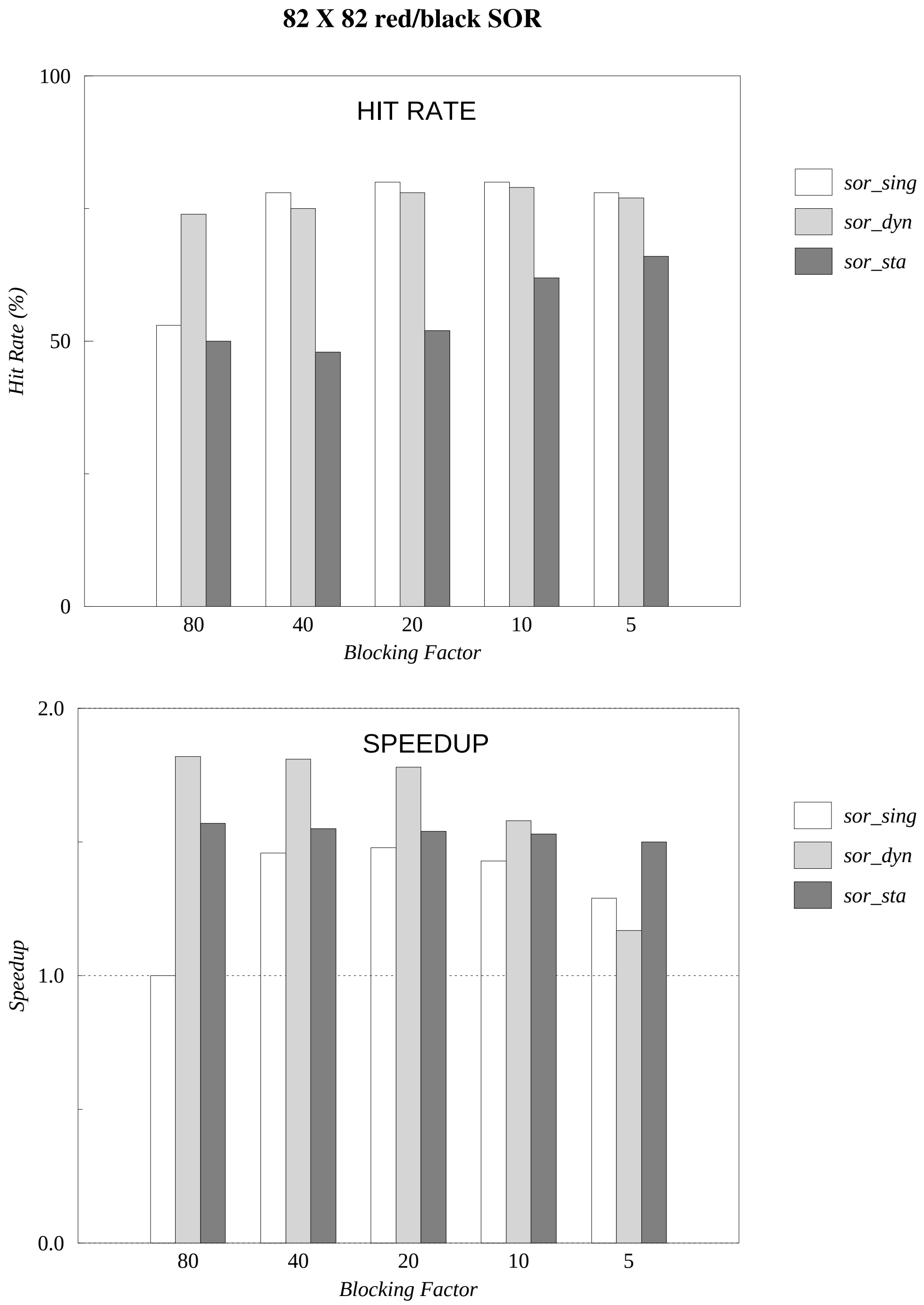


Figure 7.8: Hit rates and speedups for an 82X82 red/black SOR with a 1Kbyte direct-mapped cache. Multiple-context versions of the code use 4 contexts.

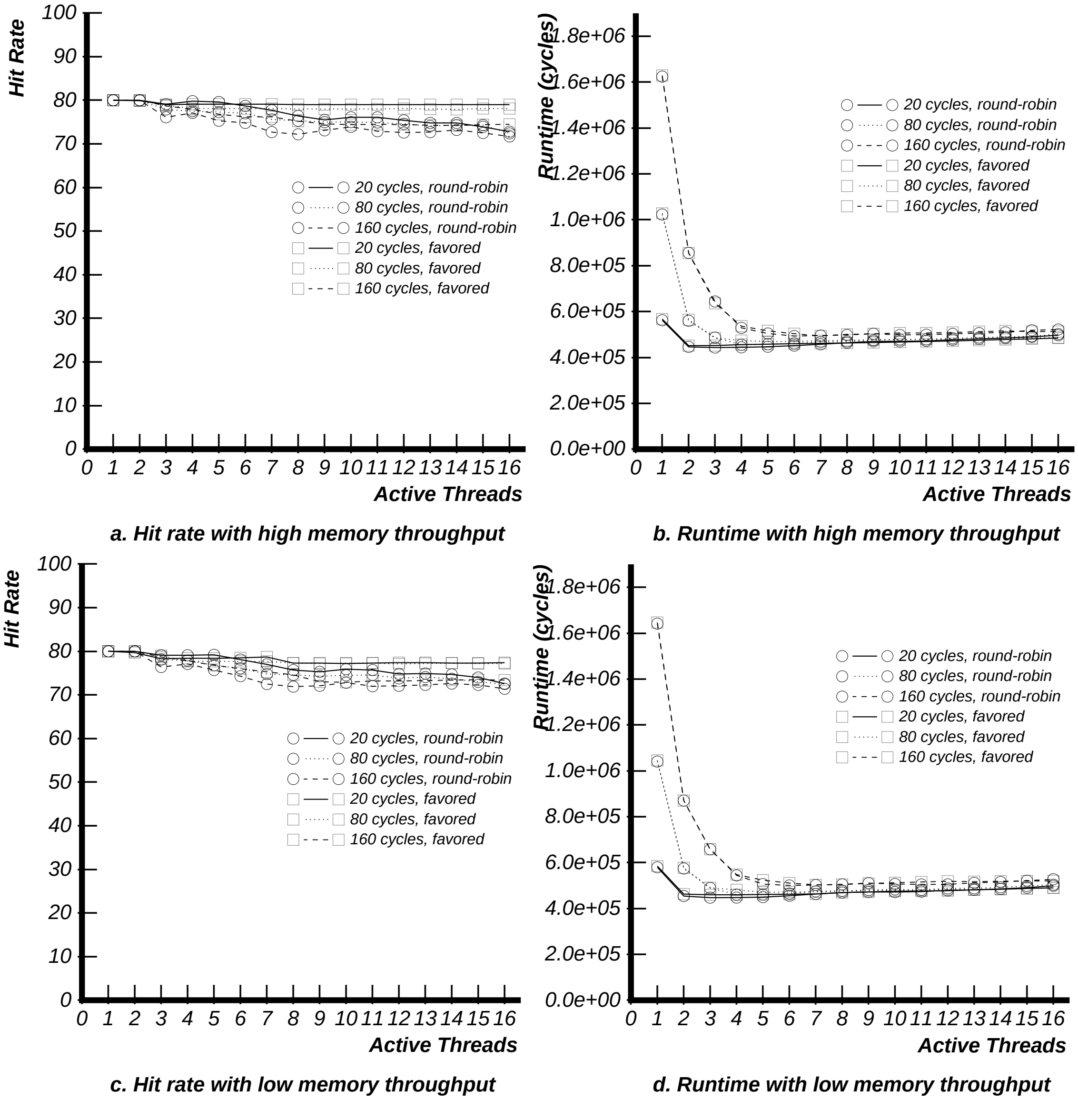


Figure 7.9: Performance of **sor_dyn** comparing round-robin and favored execution for different memory latencies and throughputs. An 82X82 SOR is done using a blocking factor of 20 and a 1Kbyte direct-mapped cache.

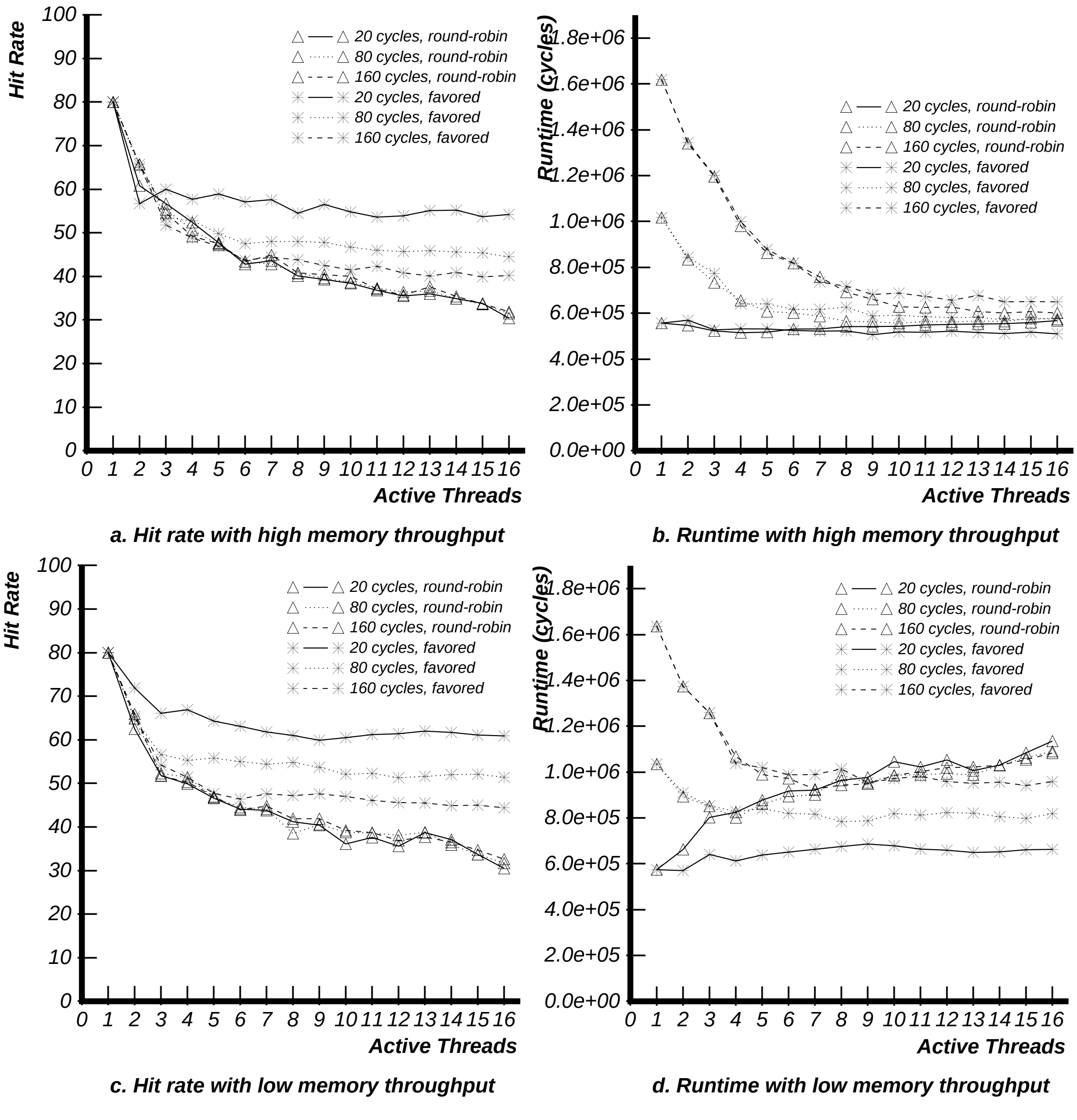


Figure 7.10: Performance of **sor_sta** comparing round-robin and favored execution for different memory latencies and throughputs. An 82X82 SOR is done using a blocking factor of 20 and a 1Kbyte direct mapped cache.

the threads are closely coordinated and are working on closely related rows and columns so that there is significant overlap in the working sets.

For the **sor_sta** case shown in Figure 7.10 the situation is much different as threads have only a limited amount of overlap in their working sets. As a result, with round-robin scheduling the cache hit rate decreases with increasing loaded threads independent of the latency. Going from 1 to 2 contexts gives the biggest drop in hit rate as 2 working sets are immediately too much for the cache to handle, though the runtime is better due to the latency tolerance provided by the multiple threads. With the favored thread scheduling, the hit rate also decreases, but then stabilizes approximately at the value corresponding to the minimum number of loaded threads required to tolerate the observed latency, just as it did in the matrix multiply example. The effect of the better hit rate on runtime depends on the memory throughput. If memory throughput is high, the only runtime penalty is due to the extra context switching and it is small. If memory throughput is low however, bandwidth becomes the limiting factor, and the penalty is not only a context switch, but also a penalty due to using more bandwidth. With low memory throughput and 16 threads, favored scheduling is better than the round-robin case by 42%, 25%, and 11% for latencies of 20, 80, and 160 cycles respectively.

Doing favored execution can lead to a load imbalance problem when work is distributed statically in large chunks and the latency is long. When threads have about the same work to do between barrier operations, and they are scheduled in round-robin fashion, they tend to arrive at the barrier at about the same time. This means that the multiple contexts effectively provide latency tolerance for each other throughout most of the computation. When favored execution is used, the favored threads tend to finish first, and a situation can arise in which only a few threads have work remaining, but not enough threads are doing work to effectively tolerate latency. The graph of the **sor_sta** runtime shows a number of cases in which the round-robin scheduling performs better than the favored scheduling due to this problem. This is the case for instance with a latency of 160 cycles, and high memory throughput. With 16 threads, a 160 cycle memory latency, and round-robin scheduling, threads arrived at the barrier in a span of about 32000 cycles. When favored scheduling was used, the threads arrived at the barrier in a span of about 324000 cycles, a span that is over a factor of 10 longer. Note that **sor_sta** is particularly susceptible to this problem because the work is divided up statically into large chunks. The **sor_dyn** does not suffer from this problem because work is dynamically acquired in small chunks, and threads always arrive at the barrier fairly close in time.

7.3.3 Sparse-Matrix Vector Multiply

Figure 7.11 shows the code for multiplying a vector with a sparse matrix. Figure 7.12 shows the compressed storage format used for the sparse matrix. This format stores the non-zero elements of the matrix in a linear data array, and uses index arrays to denote the start and the end of the each matrix row, as well as the column position of each row element.

```
for (row = 0 ; row < num_rows ;row++) {
  index_start = rowstart[row];
  index_end = rowstart[row+1];
  partial = 0.0;
  for (index = index_start; index < index_end ; index++)
    partial += x[columnpos[index]] * a[index];
  result[row] = partial;
}
```

Figure 7.11: Sparse-matrix vector multiply code.

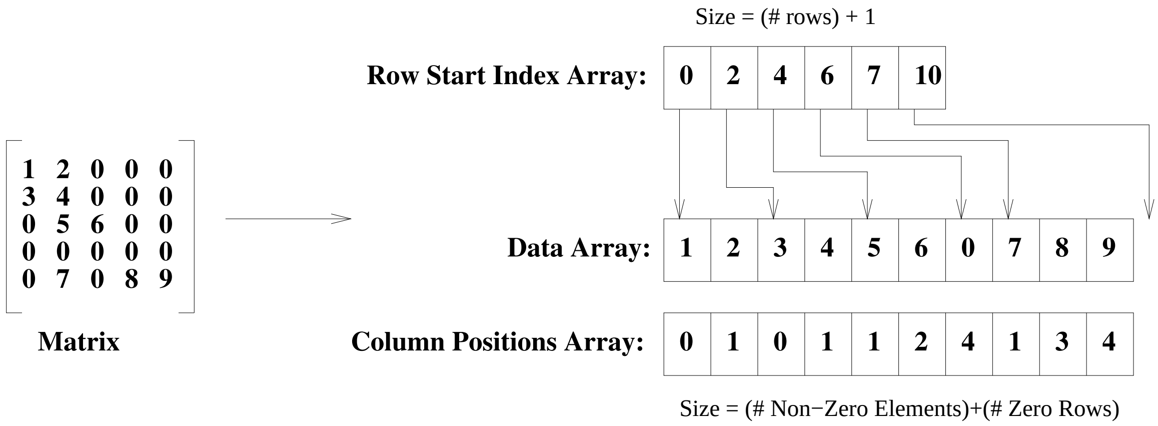


Figure 7.12: Example of sparse matrix storage format using row indexing.

The data reuse patterns in sparse matrix code are less clear than for regular dense matrix code. We can exploit self-spatial reuse in the sparse matrix references, as different values are read linearly from the index and the data arrays. There can also be self-temporal reuse of vector elements because several rows of the sparse matrix will have non-zero elements in the same columns, and will need to read the same vector elements. In typical sparse matrices, rows that have non-zero elements in the same column are close to one another in the matrix.

We simulate three different versions of the sparse matrix code:

1. **smvm_sing**: Single context version of the code.
2. **smvm_dyn**: Multiple context version with fine-grain distribution of the **row** iterations to the multiple contexts.
3. **smvm_sta**: Multiple context version that statically assigns contiguous blocks of rows to each context.

Data Sharing Effects

Results for different matrices taken from the Harwell/Boeing sparse matrix set [28], a collection of matrices taken from a variety of scientific disciplines, are shown in Figure 7.13. The general trend is that multiple context versions perform better than the single context version due to the latency tolerance provided by the multiple contexts. Furthermore, the multiple context version in which the contexts dynamically acquire row iterations performs better than the version that assigns contiguous blocks of threads to the contexts, due to improved cache performance. This improved cache performance is due to the fact that threads tend to be working on row numbers that are contiguous, these rows have a number of the same columns that have non-zero entries, and so share vector data.

Favored Thread Execution Effects

Figures 7.14 and 7.15 show the cache hit rate and the runtime for **smvm_dyn** and **smvm_sta** respectively, with an increasing number of contexts, and different memory latencies and throughputs. The sherman2 test matrix is used as a representative example. Two cases are shown, one in which threads have equal priority, so that threads are scheduled in round-robin fashion, and one in which threads have a priority based on the iteration they are calculating so that threads execute in a favored order.

The curves show much the same trends as the curves for the SOR benchmark. For the **smvm_dyn** case shown in Figure 7.14, data sharing minimizes the drop in cache performance even with round-robin scheduling. Though the favored execution minimizes the drop

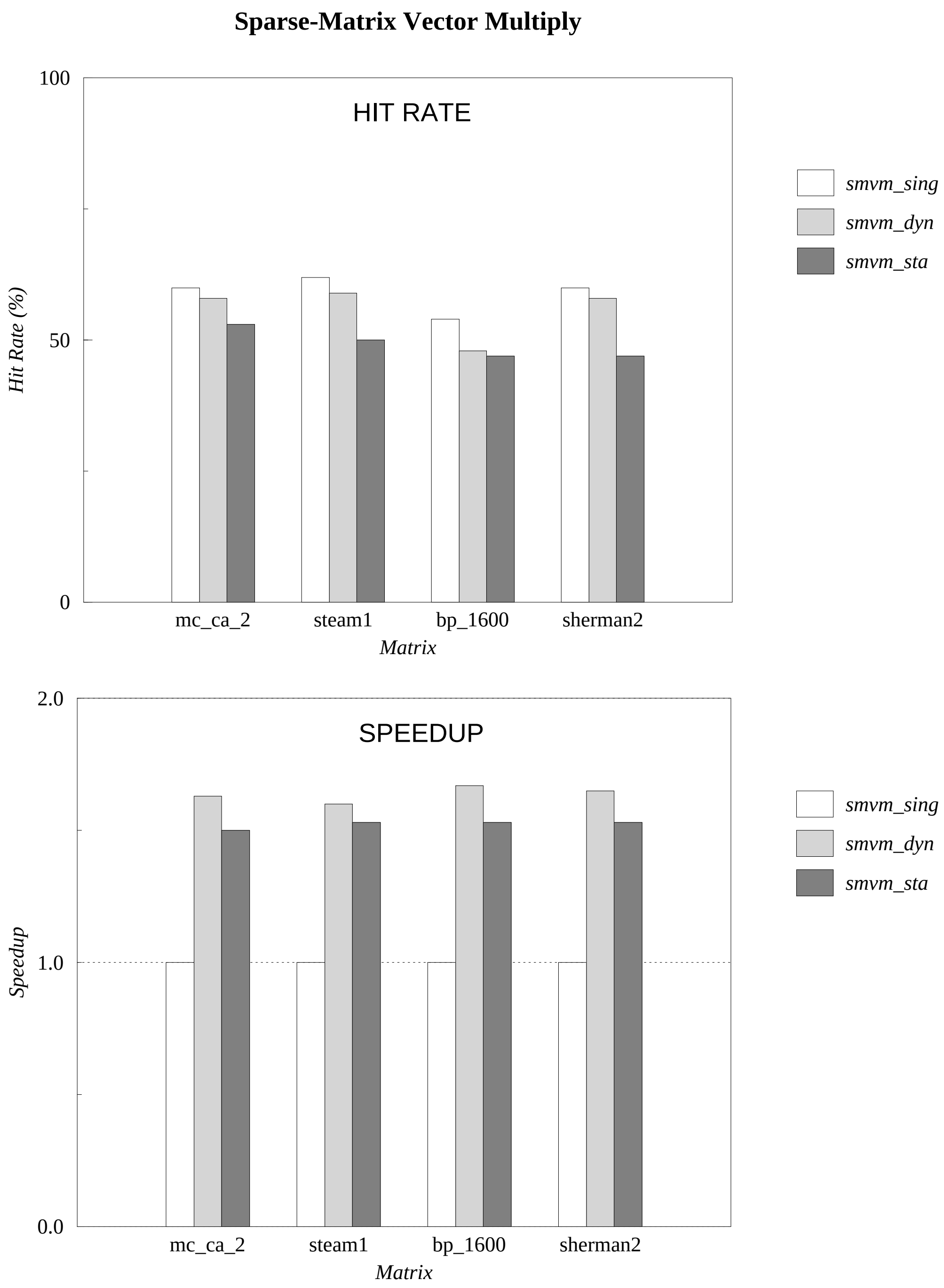


Figure 7.13: Hit rates and speedups for the Sparse-Matrix Vector Multiply using different sparse matrices. Multiple-context versions of the code use 4 contexts. A 1Kbyte direct-mapped cache is used.

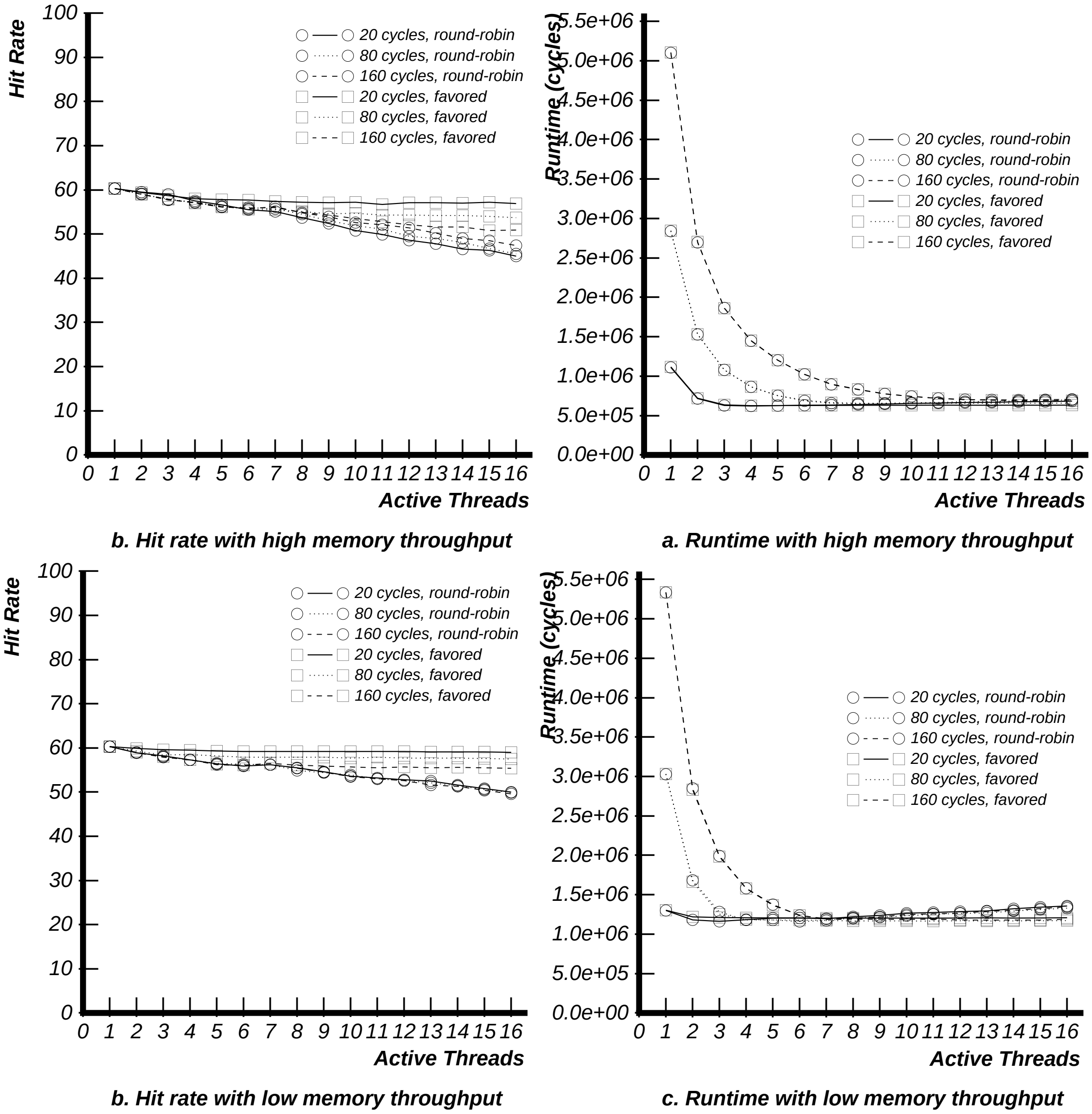


Figure 7.14: Performance of **smvm_dyn** comparing round-robin and favored execution for different memory latencies and throughputs. The sherman2 matrix is used as an example, using a 1Kbyte direct-mapped cache.

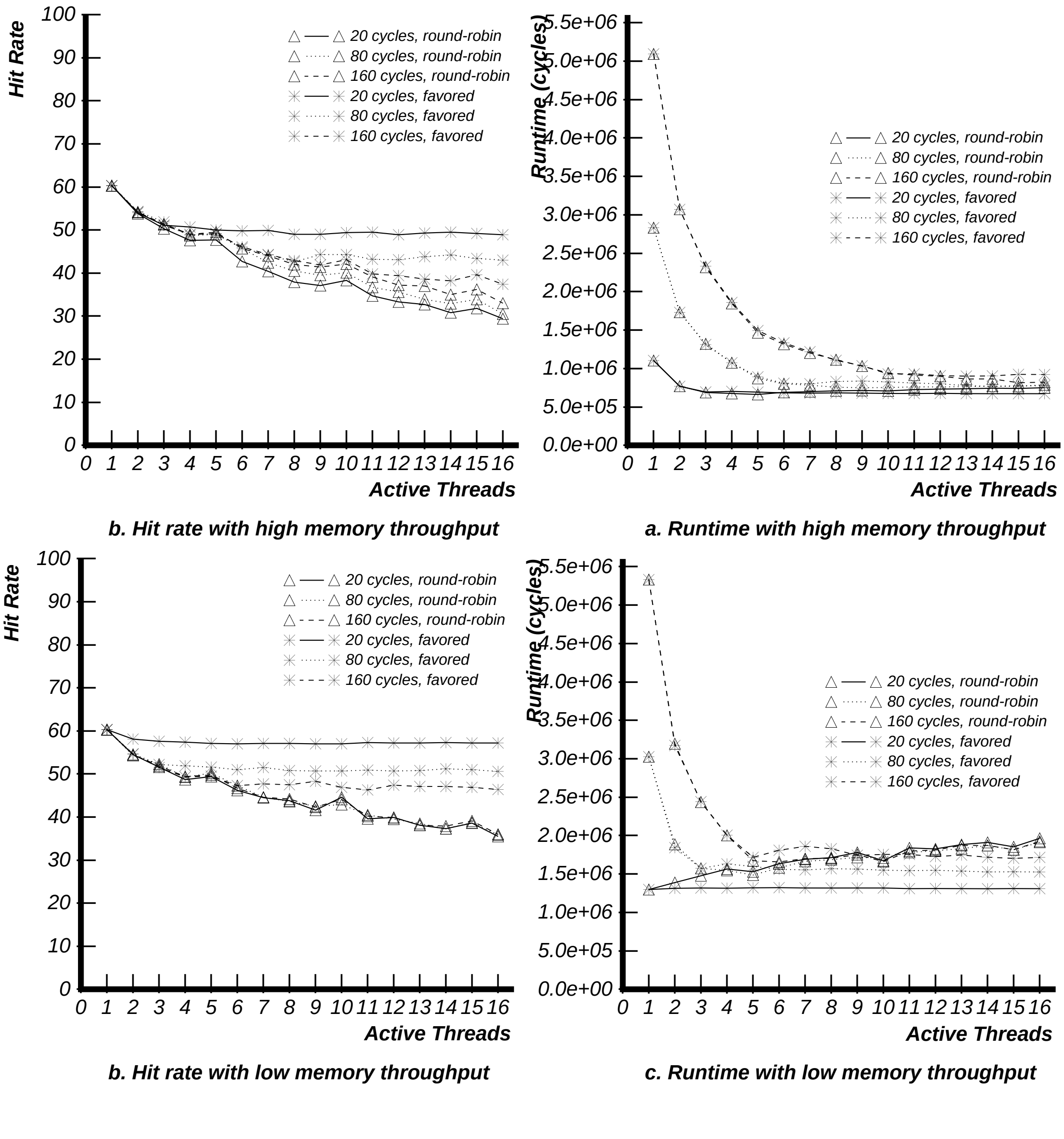


Figure 7.15: Performance of **smvm_sta** comparing round-robin and favored execution for different memory latencies and throughputs. The sherman2 matrix is used as an example, using a 1Kbyte direct-mapped cache.

in the cache hit rate, there is almost no gain in performance in the high memory throughput case, and a small gain in performance in the low memory throughput case. For 16 threads, the performance is better by about 11% independent of the latency.

For the **smvm_sta** case, data sharing is not so good, and with round-robin execution cache performance drops off dramatically. Favored execution minimizes the drop in cache hit rate. If there is more than sufficient bandwidth then the better hit rate again has little impact on performance, and the load balancing problem can cause the favored execution to perform worse than round-robin execution. If bandwidth is limited, favored execution has a significant impact on performance. With 16 threads, performance was better by 33%, 20%, and 11% for latencies of 20, 80, and 160 cycles respectively.

7.4 Summary

In this chapter we examined the effects of multithreading on cache performance using a multiple-context processor. Previous studies show that cache performance suffers as the number of threads executing increases, and that this can limit the benefits of multithreading. We present two techniques that limit the negative effects of multithreading on the cache:

- Data sharing: We define threads so that they share common data, and we closely coordinate thread execution so that they use common data at approximately the same time. When the threads share data they can both prefetch data for each other, and data can be pipelined through the cache and be used by each thread in turn. A key component of maximizing data sharing is the dynamic distribution of work in small chunks to the different threads so that they remain closely synchronized. In the context of loop iterations, we can do this very efficiently since the contexts share a cache, and we can use a simple Fetch-and-Increment to distribute loop iterations at a very fine grain.
- Favored Execution: We use thread prioritization to make the threads execute in a preferred order. In this way the minimum number of threads needed to tolerate latency can be executing, thus minimizing the number of working sets that are in the cache, and maximizing the hit rate. Favored execution allows the number of threads executing to adjust dynamically to the observed latency, provided there are more than enough threads to tolerate latency.

Several simple experiments illustrate the benefits of data sharing and favored execution as well as the following interesting points:

- Thread definition is very important. For the different applications it is possible to define threads so that they have considerable working set overlap, or so that there is much less working set overlap.

- Dynamic distribution of iterations is very important because it minimizes the load balancing problem, and allows the close coordination of threads. The versions of the benchmarks that distribute the work in small chunks avoid the load balancing problem in which several threads finish their execution and leave other threads to execute for long periods without the benefit of latency tolerance. Also, when work is dynamically distributed to the threads they work on closely spaced iterations which leads to better data sharing.

- Barrier overhead can still be an issue. If a local barrier is being done between groups of iterations (to exploit blocking for instance) then the amount of work between barriers has to be large enough to compensate for this extra overhead. Since this overhead is quite small due to the contexts sharing a cache, it only becomes an issue when barriers are very frequent as in the **sor_dyn** case with a blocking factor of 5.

- Favored execution succeeds in choosing the minimum number of threads required to tolerate the observed latency. As the latency increases an increasing number of threads execute, but only just enough to tolerate the latency. As a result, although the cache performance decreases with increased latency because more contexts are executing, the hit rate stabilizes once enough contexts are available to fully tolerate latency.

- Favored execution has a greater impact when the threads have less overlap in their working sets because each additional thread that is executing adds a greater amount of data that has to be in the cache than if the threads have more overlap.

- Favored execution can make the load balancing problem worse. It does this by causing some threads to finish well before others, leaving insufficient threads to tolerate latency.

- The effect of favored execution on runtime depends on the memory bandwidth available. If lots of memory bandwidth is available, then the impact of the improved cache performance will be reduced, because the only penalty for an extra cache miss is the cost of a context switch. If memory bandwidth is limited, favored execution can have a large impact on performance since the increased hit rate reduces bandwidth requirements.

The performance improvements due to data sharing and favored execution depend on the number of contexts, the memory latency, and the memory throughput. For our three simple benchmarks, versions of the code written to exploit data sharing have better cache hit rates and runtimes than versions that are not. With 4 contexts, round-robin scheduling, and a 1Kbyte direct-mapped cache, the cache hit rate was better by 1 to 26 percentage points. Favored execution is also useful in improving cache hit rate, especially when there are many threads that do not share much data. For 16 threads, favored execution had better hit rates than round-robin execution by 7 to 24 percentage points. The runtime improvement due to better cache hit rates depends on the cost of a context switch, and the memory throughput. Combining both data sharing and favored execution, runtime improvements range up to 16% for the high memory throughput case, and up to 50% for the low memory throughput case.

Chapter 8

Critical Path Scheduling

In this chapter we concentrate on showing how thread prioritization can be used to guide the scheduling based on the critical path. This may be an actual critical path determined from a static program graph, or the path that the user or compiler heuristically chooses as being the most important. Many programs can be analyzed by the compiler to determine which threads are most critical. In some dynamic cases where it is not possible to determine a good schedule at compile time, it is possible to determine a schedule at runtime using a pre-processing step which examines the task graph. We use a number of simple benchmarks to study how prioritization affects performance.

The experiments show how the effect of the prioritization depends very much on the characteristics of the problem. In particular, the problem must be critical path limited, and there must be sufficient parallelism. For this type of problem, performance of the prioritized cases is as much as 37% better than the unprioritized case. When the problem is not critical path limited, or there is insufficient parallelism, prioritization has a negligible effect.

We also examine the effects of memory latency and memory throughput. The most important parameter is the memory latency. As the latency increases, hardware prioritization is less effective in improving performance because more of the contexts are stalled at any given time, and the choice of which context to execute next is reduced. Memory throughput affects the overall performance if the application is memory bandwidth limited or there is a memory bottleneck. The impact of prioritization tends to increase when the application is memory bandwidth limited.

8.1 Benchmarks

In this chapter we use both regular and irregular problems that benefit from scheduling threads based on an estimate of the critical path.

The first two benchmarks are a dense triangular solve and a sparse triangular solve that evaluate the benefits of hardware prioritization of the contexts. The number of threads generated is always less then the number of contexts. The dense triangular solve is a regular problem with a well defined DAG and it is easy to statically distribute the work across processors. The sparse triangular solve has a much more irregular DAG that depends on the sparsity pattern of the matrix, and that can only be determined at runtime. This represents a class of problem which can benefit from an *inspector-executor* approach [82] in which the *inspector* analyzes the dependencies at runtime, and based on this analysis, places and schedules tasks on the processors. The *executor* then executes these tasks. Provided the same pattern of computation is performed many times, the cost of doing the scheduling analysis can be amortized over many computations. Since we are not primarily interested in the load-balancing aspect of these problems, we use a specific approach to load balancing, and then concern ourselves with the effects of prioritizing threads and contexts.

The next two examples are dense matrix Lower-Upper Decomposition (LUD) and sparse matrix LUD that evaluate the benefits of both hardware and software prioritization. In these examples we adopt a more dynamic approach to task generation, and there can be more threads than contexts. As with the two triangular solve algorithms, the dense LUD is a regular problem which can be statically load balanced, and the sparse LUD is an irregular problem that benefits from runtime DAG analysis and scheduling.

These 4 benchmarks are described in more detail below.

8.1.1 Dense Triangular Solve

A dense triangular solve finds a vector x such that $Tx = b$, where T is an upper or lower triangular matrix. For instance, given the LUD of a matrix A such that $Ax = LUx = b$, we can solve for the vector x by first doing a forward substitution step to find the vector y such that $Ly = b$, and then a backward substitution step $Ux = y$ to find x.

Figure 8.1 shows the code for the forward substitution. Each element of the result vector $x[i]$ depends on all the previous elements $x[j]$ where $j < i$. Assuming we create a thread to calculate each element, the prioritization is such that earlier i iterations have higher priority.

The benchmark does forward substitution using a 256x256 triangular matrix that is distributed in blocks of rows across the processors. Processors are responsible for calculating

```
for (i = 0 ; i < N ; i++) {
  result = b[i] ;
  for ( j = 0 ; j < i ; j++) {
    result -=  x[j] * T[i][j] ;
  }
  x[i] = result/T[i][i] ;
}
```

Figure 8.1: Serial dense triangular solve code.

interleaved elements of the result vector. There is some inherent load imbalance both in the problem itself, and in the way of dividing up the work across the processors. The load imbalance within the problem comes from the fact that there is a different amount of work required for calculating the final value of each element, and the calculation of these elements is distributed in an interleaved fashion across the processors. Each processor spawns a certain number of threads into its different contexts, and each thread acquires an element $x[i]$ to update by incrementing a counter local to each processor. Each processor has its own copy of the x vector and when a processor writes an element to the x vector it explicitly sends messages to update the vectors on all the other processors. Fine-grain synchronization using Full/Empty bits is used to make sure that a processor does not use an element of the x vector before it has been calculated. A thread that tries to read an empty location does a context switch and attempts to read the location again at a later time. Three different prioritization schemes are used:

1. Unprioritized: Threads are run in round-robin order.
2. Write Prioritized: Threads writing a final value to the x vector are given high priority.
3. Level Prioritized: Threads are prioritized based on the index of the element that they are calculating. Lower indices have higher priority.

The benchmark uses 16 processors. Each processor has an 8Kbyte, 4-way set-associative, 16 bytes per line cache.

8.1.2 Sparse Triangular Solve

The sparse triangular solve benchmark does forward elimination using a sparse triangular matrix. The overall calculation is the same as for dense triangular solve, except that it exploits the sparse nature of the matrix to eliminate unnecessary computations associated with the 0 elements of the matrix. The sparse matrix format is the same as used in Chapter 7, and is shown again in Figure 8.2 for convenience. Figure 8.3 shows the code for the

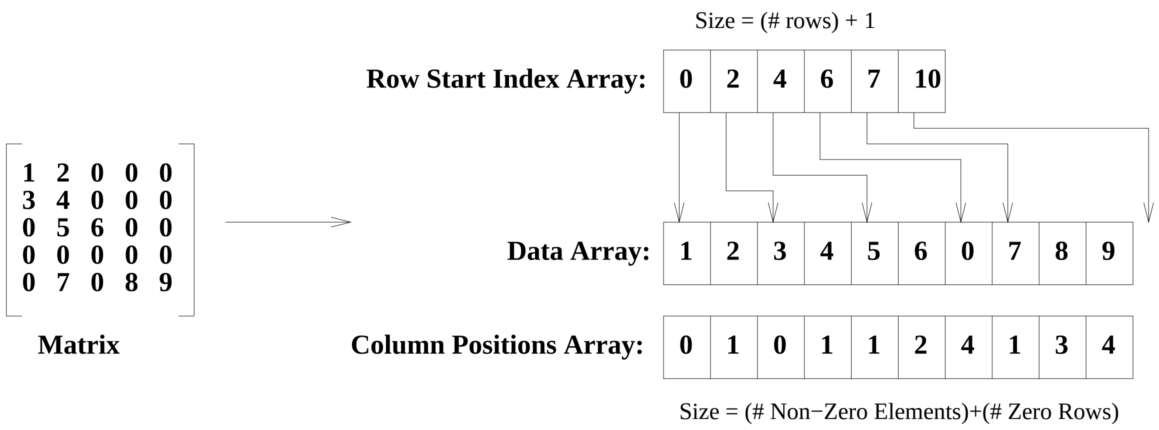


Figure 8.2: Example of sparse matrix storage format using row indexing.

```
for (i = 0 ; i < N ; i++) {
  result = b[i] ;
  for ( j = rowstart(i) ; j < rowstart(i+1) ; j++) {
    result -=  x[j] * T[columnpos[j]] ;
  }
  x[i] = result/T[columnpos[i]] ;
}
```

Figure 8.3: Serial sparse triangular solve.

sparse forward elimination. Each element of the result vector $x[i]$ depends on only some of the previous elements $x[j]$ where $j < i$. Depending on the pattern of the sparse matrix, the dependencies can be quite complex.

It is important to perform some amount of pre-scheduling and load balancing [82, 19]. In the prescheduling phase, the depth of each element $x[i]$ is calculated. The depth of element i is the length of the longest dependency chain of x elements that begin with $x[i]$. The elements are sorted in order of decreasing depth and assigned round-robin to the processors. Each processor will calculate its x vector elements starting with those of greatest depth.

The code is parallelized by having each processor spawn a given number of threads into different contexts, and each thread acquires an element $x[i]$ to update by incrementing a counter local to each processor and accessing the next element in the pre-calculated schedule. Fine-grain synchronization with Full/Empty bits and spinning is used to prevent threads from using elements of the x vector before they have been calculated.

We used two different prioritization schemes, and various sparse input sparse matrices:

Matrix	Description	n	nonzeros
bcspwr07	Power Network	1612	3718
mat6	Circuit Simulation	687	6449
adjac25	Adjacency Matrix	625	22797

Table 8.1: Sparse matrices used in the benchmarks.

1. Unprioritized: Threads are run in round-robin order.
2. Level Prioritized: Threads are prioritized based on their distance from the outputs of the DAG that describes the dependencies between $x[i]$ elements. Threads that are further from the outputs have higher priority.

We used the three different sparse matrices described in Table 8.1 as input to the benchmark, representing very different sparsity patterns. *bcspwr07* is a very sparse matrix from the Harwell-Boeing Sparse Matrix suite and comes from the sparse matrix representation of a power network. *mat6* is from the circuit simulation domain, and represents the fill-in pattern for a direct sparse LUD solve as will be discussed in more detail in section 8.1.4. *adjac25* comes from a grid adjacency matrix, and again represents the fill-in pattern that occurs during direct sparse LUD.

It should be noted that the ordering of thread execution is different from the approach taken by Saltz et. al. in [82]. In their case they divide the computation into wavefronts, where each wavefront consists of those tasks that can be calculated independently assuming that all previous wavefronts have completed. Barriers can be performed between each wavefront calculation phase. Our use of fine-grain synchronization allows the emphasis to be placed on scheduling the critical path. Also, our benchmark depends on the multiple contexts to tolerate latency rather than taking a data driven approach as done by Chong et. al. [19].

Note that only the runtime of the compute phase is measured, not the time for the scheduling, which is done serially. The benchmark uses 16 processors. Each processor has an 8Kbyte, 4-way set-associative, 16 bytes per line cache.

8.1.3 Dense LUD

One way of solving a system of linear equations $Ax = b$ is to first find the LUD of the A matrix, followed by forward and backward substitution steps to find the vector x [49]. The decomposition phase of the algorithm is $O(n^3)$ where n is the size of the matrix, and represents the major portion of the computation for large problems. This phase of the algorithm uses Gaussian elimination to find the lower triangular matrix L and upper triangular matrix U such that $A = LU$.

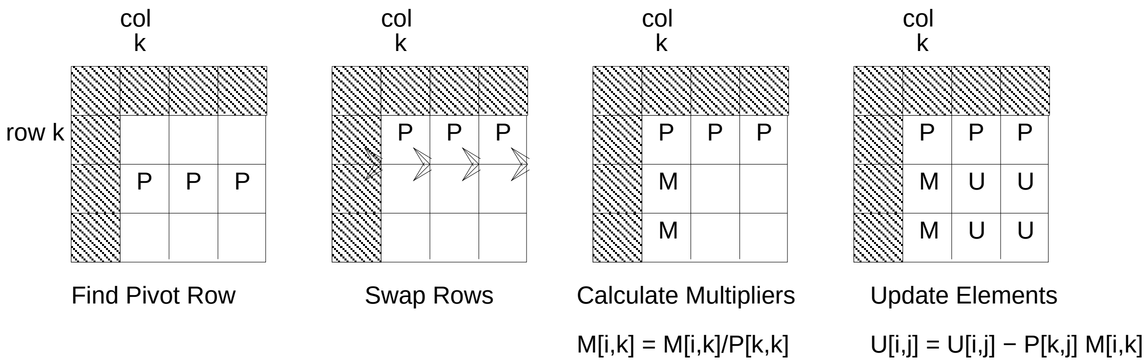


Figure 8.4: LUD with partial pivoting.

The algorithm with partial pivoting is shown pictorially in Figure 8.4. At each step of the computation one row and one column of the final LUD matrix is determined. Each iteration requires the following 4 steps to be taken:

1. Search all elements in the leftmost column of the current submatrix for the element with the largest absolute value. This element is the pivot and its row is the pivot row.
2. Switch all the elements of the pivot row and the topmost row of the current submatrix.
3. Calculate the multiplier column by dividing all the elements below the pivot by the pivot.
4. Update all elements in the new submatrix which excludes the topmost row and leftmost column of the current submatrix, by subtracting the product of the multiplier corresponding to the element's row and the element in the pivot row from the same column.

The benchmark does a 64X64 LUD with matrices distributed in a column interleaved fashion across the processors. Processors are responsible for calculating all values related to the columns that they own. Note that there is some inherent load imbalance both in the problem itself, and in the way of dividing up the work across the processors. The load imbalance within the problem comes from the fact that there is a different amount of work required for calculating the final value of each column and that the work for each column is statically allocated across the processors. Two different versions of the benchmark were used[1]:

[1]Though a simple LUD example has been chosen here for illustration purposes, it should be noted that there is generally enough easily exploitable parallelism in LUD to achieve good performance without resorting to fine-grain synchronization. The real benefits of fine-grain synchronization are only obvious in more complex wavefront computations such as the preconditioned conjugate gradient computation discussed by Yeung and Agarwal [110].

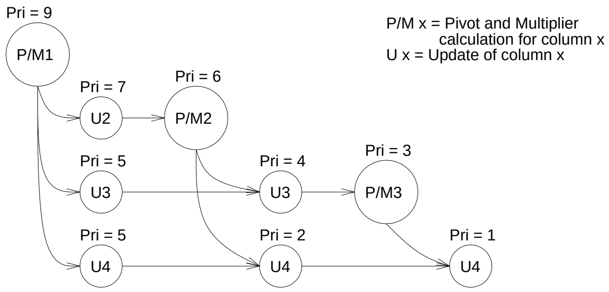


Figure 8.5: Critical path prioritization of LUD tasks for a 4 column problem.

1. Unprioritized: The fine-grain synchronization version of the program uses the fact that it is not necessary to wait for all the processors to finish before calculating the next pivot and multiplier column. At each stage, each processor generates one thread to update each column it owns. Also, the processor responsible for generating the next multiplier column generates a thread to do so. Thus stages of the computation related to different multiplier columns can proceed at the same time. Threads are scheduled in FIFO manner on the scheduling queue, and loaded threads are run in round-robin fashion. If a synchronization fault occurs, a new thread is switched in if one is available.

2. Prioritized: In this version the threads are prioritized so that at any given stage the thread that is updating the first column is given higher priority than the threads calculating the other columns, as is the thread that is responsible for generating the next multiplier column for use in the next stage. In this way, two stages of the computation are nearly always loaded, and the calculation of the next multiplier column is effectively overlapped with the updating of the submatrix. An example of this prioritization is shown in Figure 8.5 for a simple 4 column problem.

The benchmark uses 16 processors. Each processor has a small 1Kbyte, 4-way set-associative, 16 bytes per line cache.

8.1.4 Sparse LUD

The sparse LUD benchmark performs LUD on a sparse matrix. Instead of the $O(n^3)$ operations required by the dense algorithm, $O(n^\alpha)$ operations are required, where α depends on the sparsity of the matrix. For instance, for a grid problem α is equal to 1.5, and typically smaller for other circuit simulation problems.

Telichevesky [94] provides a good overview of the different steps involved in the sparse LUD which are the following:

1. Reordering: interchange rows and columns of the matrix to minimize the number of *fill-ins*. A fill-in is a given element a_{ij} that was originally 0, but becomes non-zero during the decomposition. A given source row will create a fill-in in any target row that has a 0 in the same column that the source row has a non-zero. Although computationally expensive, this step can typically be performed on symbolic data once at the beginning of the computation.

2. Load balancing and scheduling: load balance the work across the processors and their scheduling priority. Because the work per row can vary significantly, and because the dependencies between rows can be very irregular, the simple row interleaved load-balancing done in the dense LUD case does not perform well. Telichevesky [94] uses a simple load balancing scheme that estimates the amount of work associated with a row, and then assigns the rows to the processors in round-robin order. Furthermore, a priority is associated with each row update based on *remaining completion time*, or the minimum time for completion of all the tasks that depend on the current row update.

3. Data structure creation: Create data structures which allow easy access to the required matrix elements both along rows and along columns.

4. Decomposition: Using the special data structures, perform the decomposition.

The benchmark assumes a re-ordered matrix, as well as an assignment of rows to processors based on a load balancing heuristic similar to the one used by Telichevesky. The data structures used to represent the matrix are shown in Figure 8.6. An Overlapped Scattered Array (OSA) is used to represent the sparse array. OSA is a vector representation of a sparse matrix in which the distance between two non-zero elements in the same row is preserved, and no pair of non-zero elements occupy the same physical location in the vector. An *offset* array indicates the starting position of each row within the OSA vector. To easily identify target rows for a given source row, and non-zero entries for a row update, a special *diag* data structure is used. This data structure has an entry for each diagonal element, and along with the *r_in_c* and the *c_in_r* vectors, identifies all rows with non-zero elements below the diagonal, and all columns with non-zero elements to the right of the diagonal. Specifically, each *diag* entry contains 4 values: the first is the number of elements below the diagonal, the second is an offset into *r_in_c* identifying where the row numbers of the elements in the diagonals column are stored, the third is the number of elements to the right of the diagonal, and the fourth is the offset into *c_in_r* identifying where the column numbers of the elements in the diagonals row are stored. Using this data structure, the serial code for the LUD is shown in Figure 8.7.

The code is parallelized by assigning rows to processors, and having each processor spawn a thread for each *target row* that it is responsible for updating. Each one of these threads updates their target row by taking each *source row* that has to update the target row, and performing the appropriate calculation. The source rows that have to update a target row are all those rows for which the target row has a non-zero element to the left of the diagonal. An additional data structure, *lc_in_r* keeps track of all non-zero elements to the left of the

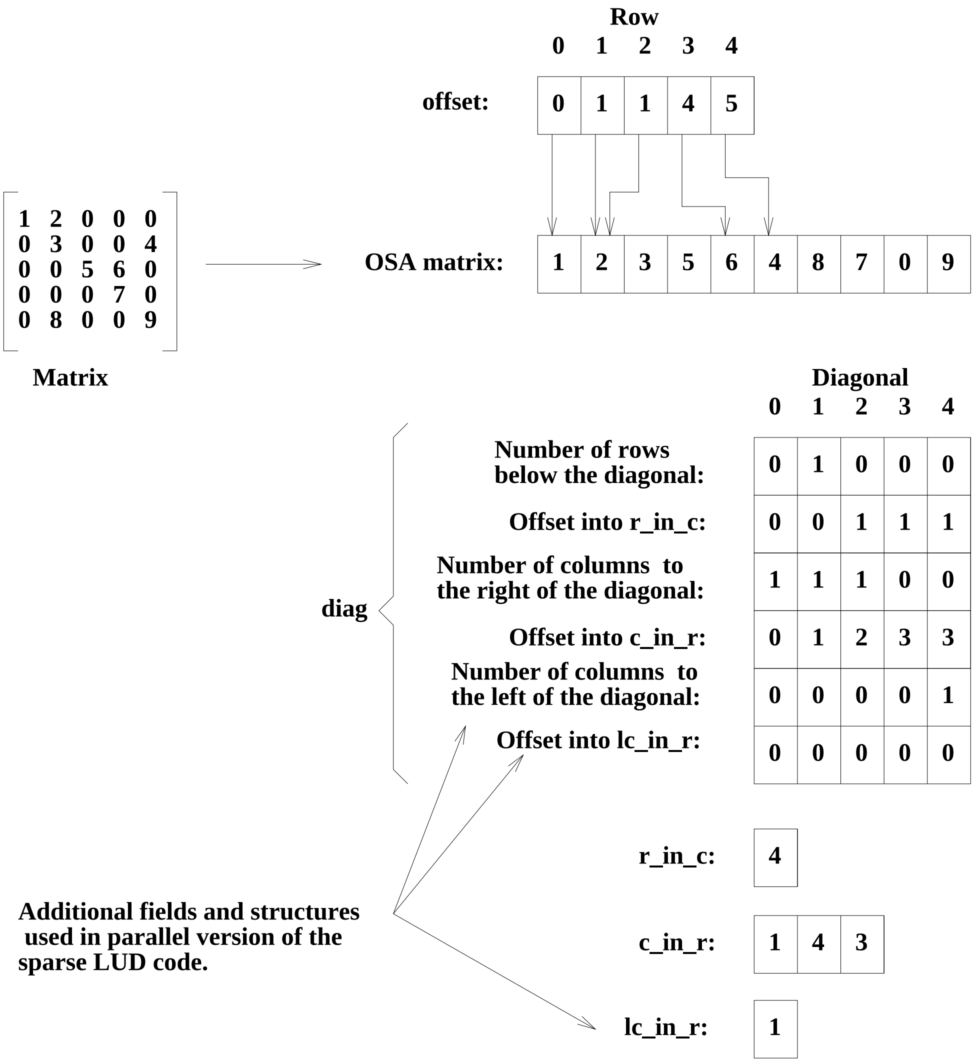


Figure 8.6: Data structures for the sparse LUD representation.

```
for(i=0; i<nrows; i++) {
  diag_val = 1.0/matrix[offset[i]+i];
  elim_row_num = diag[4*i];
  elim_row_index = diag[4*i+1];
  elim_col_num = diag[4*i+2];
  elim_col_index = diag[4*i+3];
  for(j=elim_row_num; j > 0; j--, elim_row_index++) {
    index_i = i;
    index_j = r_in_c[elim_row_index];
    /*  Normalize elements in the i column */
    matrix[offset[index_j] + i] *= diag_val;
    for(k=elim_col_num; k > 0; k--, elim_col_index++) {
      index_k = c_in_r[elim_col_index];
      matrix[offset[index_j] + index_k] -=
         matrix[offset[index_j] + index_i] *
         matrix[offset[index_i] + index_k];
    }
  }
}
```

Figure 8.7: Serial sparse LUD code.

diagonal of any given row so that the appropriate source rows can easily be identified. Each thread starts with the lowest numbered source row, and proceeds to the highest numbered source row[2]. Fine-grain synchronization is used to detect whether the next source row is available yet, and if it is not, then the thread suspends and is woken up when the desired source row becomes available. Two different prioritization schemes were used:

1. Unprioritized: Threads are scheduled in FIFO order in the scheduling queue, and loaded threads are run in round-robin order.
2. Prioritized by Remaining Work: Threads are prioritized based on estimate of the remaining work. Remaining work corresponds to the work that the thread itself has to do, plus the length of the longest chain of work that depend on this row being completed.

Note that we measure only the runtime for the decomposition step in this benchmark, as the other steps run serially. The benchmark uses 16 processors. Each processor has an 8Kbyte, 4-way set-associative, 16 bytes per line cache.

[2]This represents a more restrictive DAG than the true data dependency DAG because in some cases the order in which a row is updated by two different source rows can be interchanged. Detecting this dependency requires a more detailed analysis of the DAG however, and was not done in our case.

8.2 Results

8.2.1 Dense Triangular Solve

Figure 8.8 shows runtimes of the triangular solve for different latencies and memory throughputs. This benchmark is critical path limited since at any given time the thread responsible for calculating the next element of the x vector is the most important, and any delay in this calculation decreases the overall performance. Because this specific problem is so highly critical path limited, threads spend most of their time spinning and waiting for the next element x to be calculated, and there is little advantage to having more than 3 or 4 contexts even at high latencies. With more contexts, it is essential to prevent the spinning threads from slowing down the writing of the arriving x element. It is also essential to allow the thread calculating the x_i with the lowest index value to proceed first since the other processors use this value first. As seen in all the figures, giving high priority to the writing of the x vector array elements improves performance slightly over the unprioritized case for large number of contexts. The completely prioritized case further improves performance by allowing the highest priority thread to proceed first when a new x_i has been written.

There are a number of anomalous points on the unprioritized curve, for instance the sudden increase in runtime in the case of an 80 cycle memory latency and 15 loaded threads. This is due to the round-robin scheduling being particularly bad during several portions of the computation i.e., the needed element is written just as the processor is context switching out of the thread that would otherwise be the critical thread, and the processor executes all the other threads before coming back to the critical one. It is also interesting to note that in the unprioritized case the incremental decrease in performance due to having an additional loaded thread per processor decreases with increasing threads. This is because although each processor may begin with 16 loaded threads (since there are 16 processors, and 256 elements to calculate in the x array, 16 is the maximum number of threads that a processor can have), this number rapidly decreases as elements of the x array are calculated. Thus the performance when starting with 16 threads per processor is much the same as when starting with 14 or 15 threads per processor.

The best performance improvement due to prioritization ranges from 27% for a 20 cycle latency, to 37% for a 160 cycle latency. Since the application is not bandwidth limited, decreasing the memory throughput has only a small effect on performance and on the improvements due to prioritization.

8.2.2 Sparse Triangular Solve

Figures 8.9 through 8.10 show the runtimes of the different versions of the sparse triangular solve running on different sparse matrices. The effect of the prioritization varies from a performance improvement of 20% for the *adjac25* matrix, to virtually no gain for the

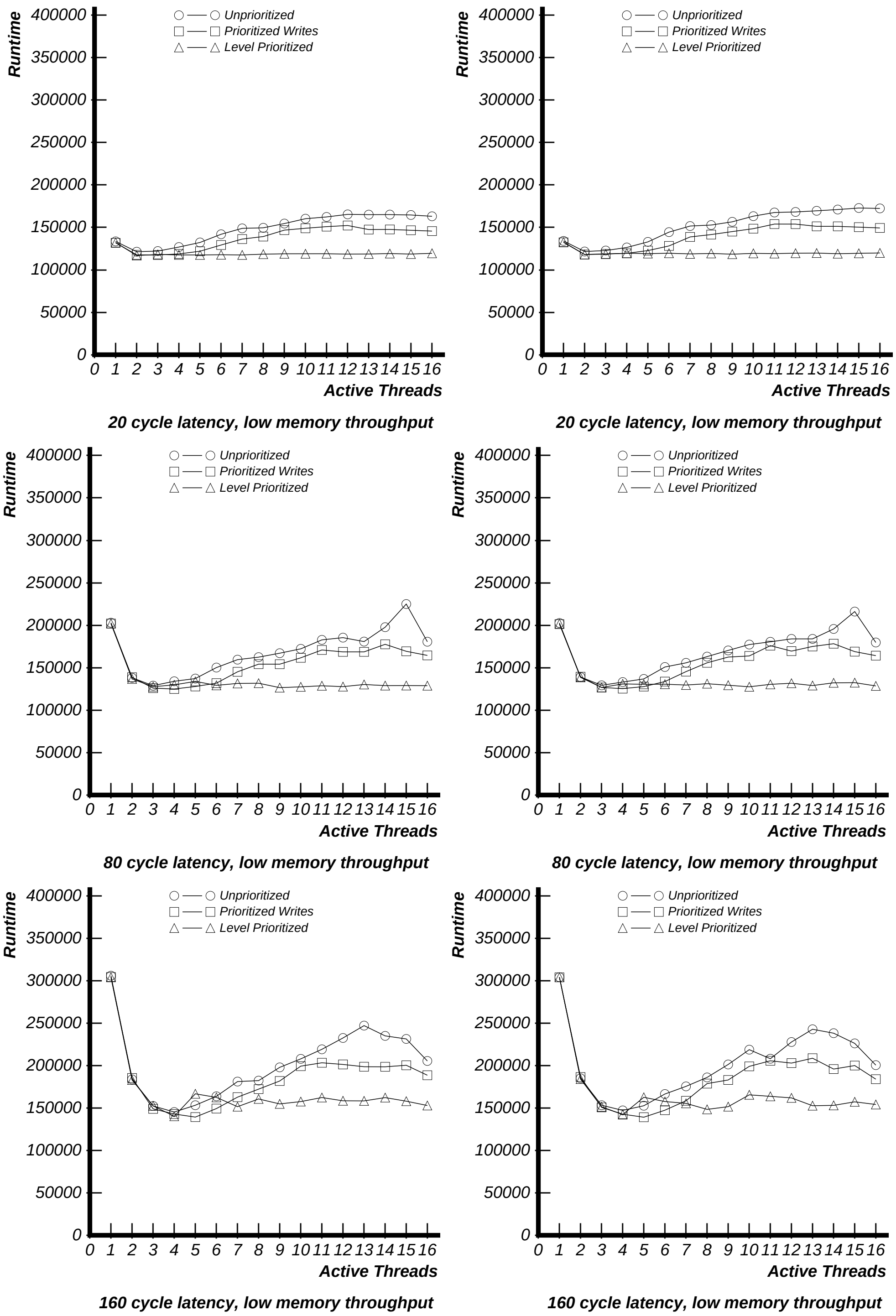


Figure 8.8: Performance of the dense triangular solve for different memory latencies and throughputs.

bcspwr07, and even a slight decrease in performance for *mat6* matrix. This difference is due to their different sparsity patterns, and is explained below.

Figure 8.9 shows the runtime of the sparse triangular solve using the *adjac25* matrix. Prioritization prevents performance from degrading as the number of contexts increases. The calculation is such that there are only a few of the x array elements that can be calculated immediately, and towards the end of the computation the matrix becomes quite dense and the dependency pattern between threads is much the same as for the dense triangular solve. Thus the dependency chains are quite long, and it is important to correctly schedule the critical path. For high memory throughput the prioritized version was better than the unprioritized version by 18% for a 20 cycle latency and high memory throughput, and by 8% for a 160 cycle memory latency. This *adjac25* case is also the case in which limiting the memory bandwidth has the most effect because of the memory contention on certain modules when many threads are trying to read the same x vector element. Overall runtime increased by up to 20%. Prioritizing the threads became slightly more important because of this: the prioritized version was better than the unprioritized version by 20% for a latency of 20 cycles, and by 14% for a latency of 160 cycles.

Figure 8.10 shows the runtime of the triangular solve using the *bcspwr07* matrix, and shows very little difference between the prioritized and unprioritized cases. The *bcspwr07* matrix is a very sparse matrix, and the dependency chains are very short. For the *bcspwr07* matrix there are 1612 result elements, and the maximum depth is 14. 1096 of the 1612 have a depth of 3 or less. Only 23 elements have a depth greater or equal to 10. This computation resembles most a bunch of largely independent threads with limited interdependency. As a result, this computation benefits the most from the multiple contexts showing performance improvements all the way up to 8 to 10 contexts for the case of a 160 cycle memory latency. On the other hand, this problem benefits the least from the detailed prioritization since the threads are largely independent. The decreased memory bandwidth also has little effect since memory references are well distributed.

Figure 8.11 shows the runtime of the sparse triangular solve using the *mat6* matrix. In this case the prioritized version in some cases even performs slightly worse than the unprioritized version. The computation proceeds by first having many short threads run first, and then tails off towards the end of the computation with a very few long running threads. The initial threads are short and numerous because the graph is sparse, and many elements of the x array depend on only a few of the other elements. Only a few threads depend on many elements, and these are calculated last. Note that having only a few long running threads is bad because there is not enough threads to effectively tolerate latency. Generally, the higher the level of an element, the smaller the number of accumulate operations that have to be done to calculate its value i.e. the level is also a good indicator of the number of other elements an element depends on. The *mat6* matrix has many elements that have a high level number, and only a few with a low level number. There are a total of 687 elements and a maximum depth of 38. 536 of the elements have a depth of 28 or greater. Only 42 elements have a depth of 10 or less. Prioritization does not help at the beginning because most of the running threads are of the same priority. Prioritization does not help

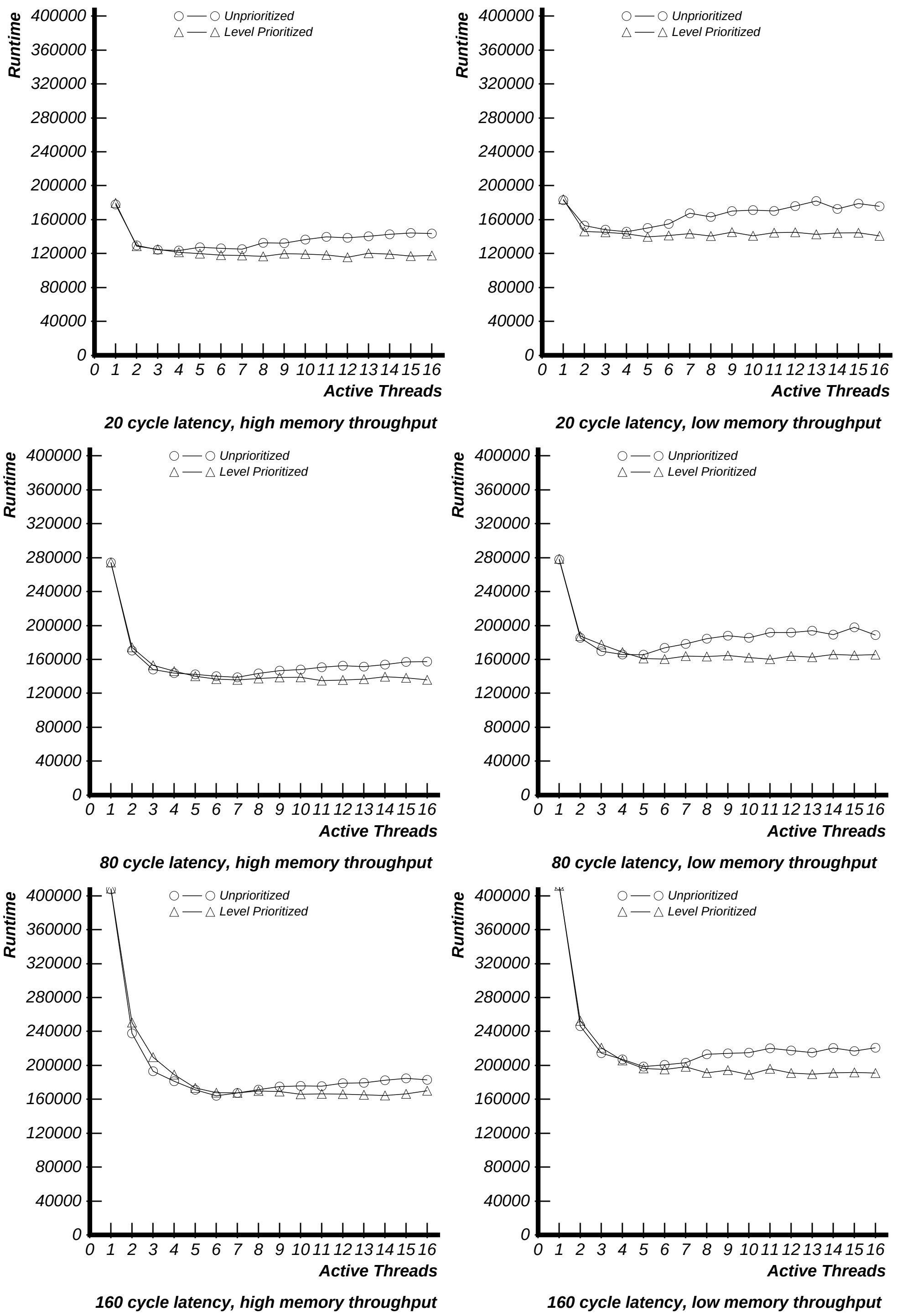


Figure 8.9: Performance of the sparse triangular solve using the *adjac25* matrix, for different memory latencies and throughputs.

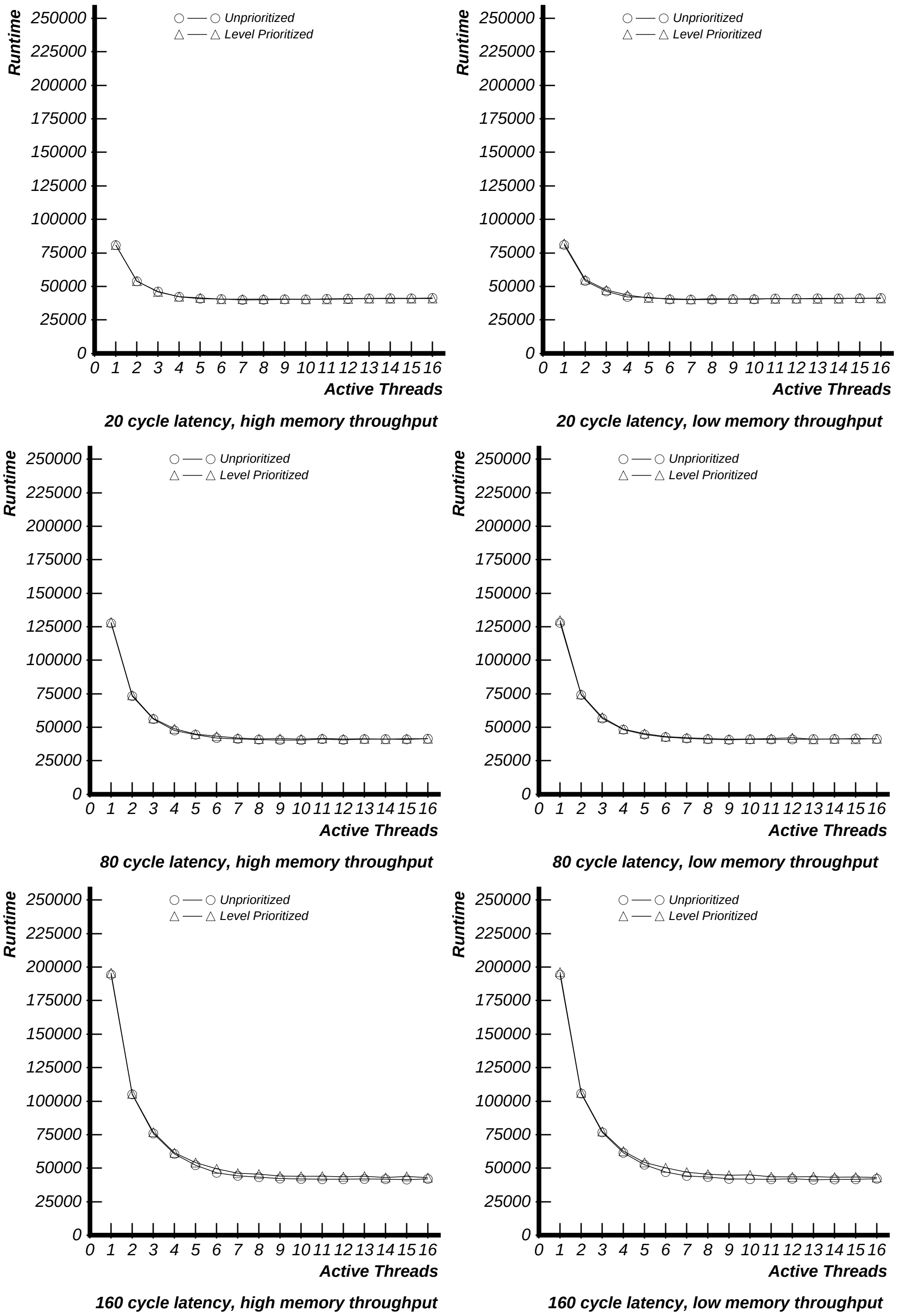


Figure 8.10: Performance of the sparse triangular solve using the *bcspwr07* matrix, for different memory latencies and throughputs.

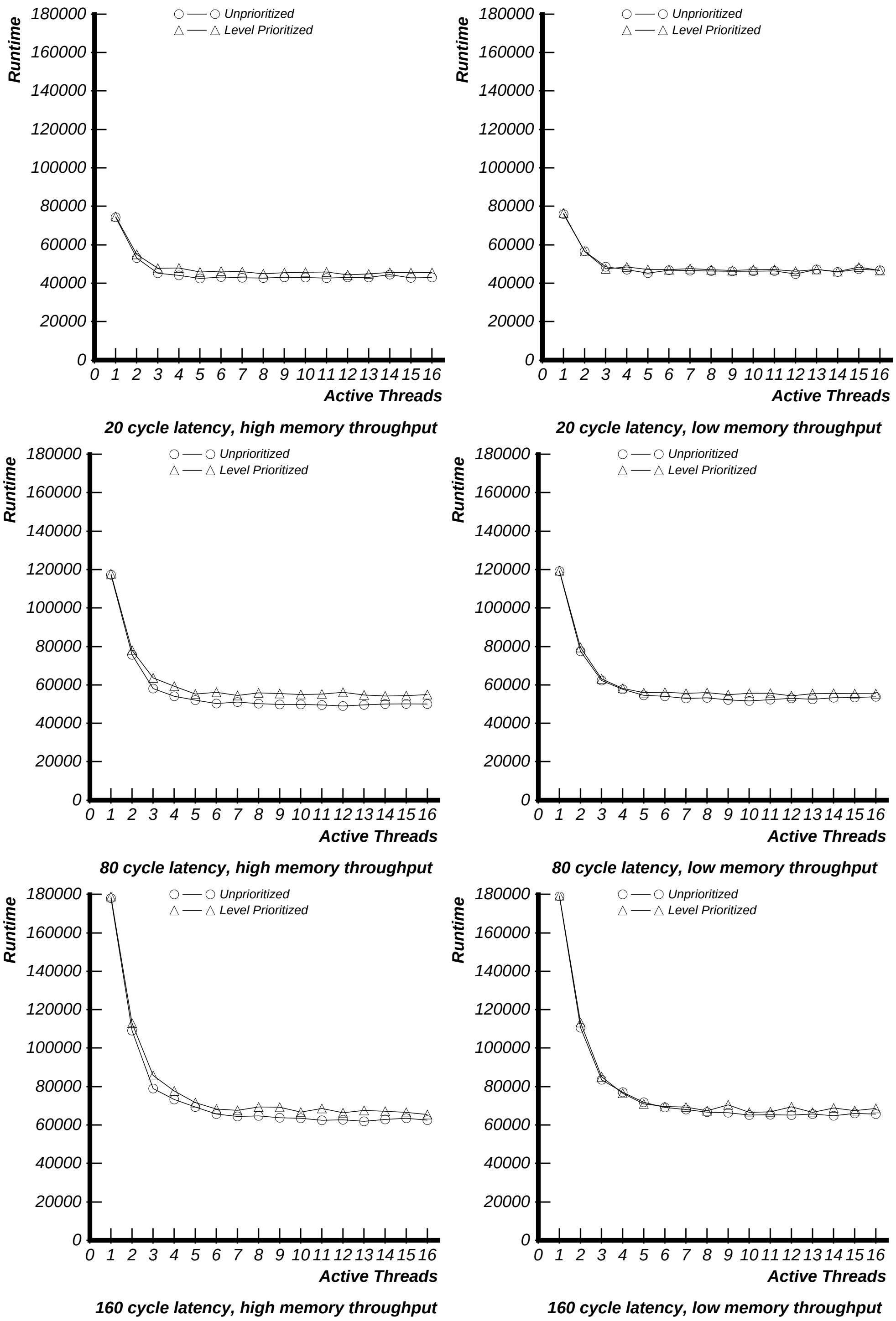


Figure 8.11: Performance of the sparse triangular solve using the *mat6* matrix, for different memory latencies and throughputs.

at the end of the computation because only a few threads are running. Further, because the prioritized version concentrates on executing some of the short threads, it can hurt latency tolerance, and can actually perform worse than the unprioritized version.

The problems that arise in the *mat6* case are artifacts of how we have defined and prioritized threads. Specifically, a given element of the x array is calculated sequentially by a single thread, and the prioritization does not take into account the number of accumulates that are required to calculate each element. What is required is to divide the accumulation operations for certain elements across several threads, so as to increase parallelism towards the end of the computation and allow better latency tolerance. This requires more complicated code since now a single element is being updated by several threads, but it allows the multiple contexts to be used more effectively. In the limit, each accumulate operation can be considered and scheduled independently.

The results of running the sparse triangular solve using these three different matrices illustrates a range of effects of both the number of contexts and the prioritization. For the *adjac25* matrix, the computation can be critical path limited, and it benefits from the hardware prioritization of threads. The *bcspwr07* case benefits from multiple contexts, but not from the prioritization since its task graph is very short and synchronization is minimal. The *mat6* matrix generates a thread pattern that causes prioritization to be ineffective, and also negatively affects the ability to tolerate latency towards the end of the computation.

8.2.3 Dense LUD

Figure 8.12 shows the running time of the LUD benchmark for a varying number of contexts. The unprioritized case performs worse because it does not give special treatment to the threads responsible for generating the next multiplier column. When it spawns threads to update the submatrix it begins to execute these threads, and they occupy all the contexts. The critical thread responsible for finding the pivot and the multiplier column is typically sitting in the thread queue, and waits until a context is free before it executes. This delays the calculation of the next pivot column to the point that it cannot be completely overlapped with the update phase. The prioritized version prioritizes the threads in such a way that generating the next multiplier column is given priority over updating the current submatrix. As a result, the update threads for the next stage are generated before the current update stage has completed. Figure 8.13 shows the lifelines for both cases with 1 context per processor. The unprioritized version shows significant idle periods as the multiplier column is calculated. In the prioritized case on the other hand, these idle times are reduced. Except at the very last portion of the calculation when the size of the remaining matrix becomes very small, the processor with the most work to do, in this case processor 15, is kept busy most of the time. Thus, we see that the processors are as busy as expected given the inherent load imbalance of the computation.

For a single context, only software prioritization comes into play. In this case the perfor-

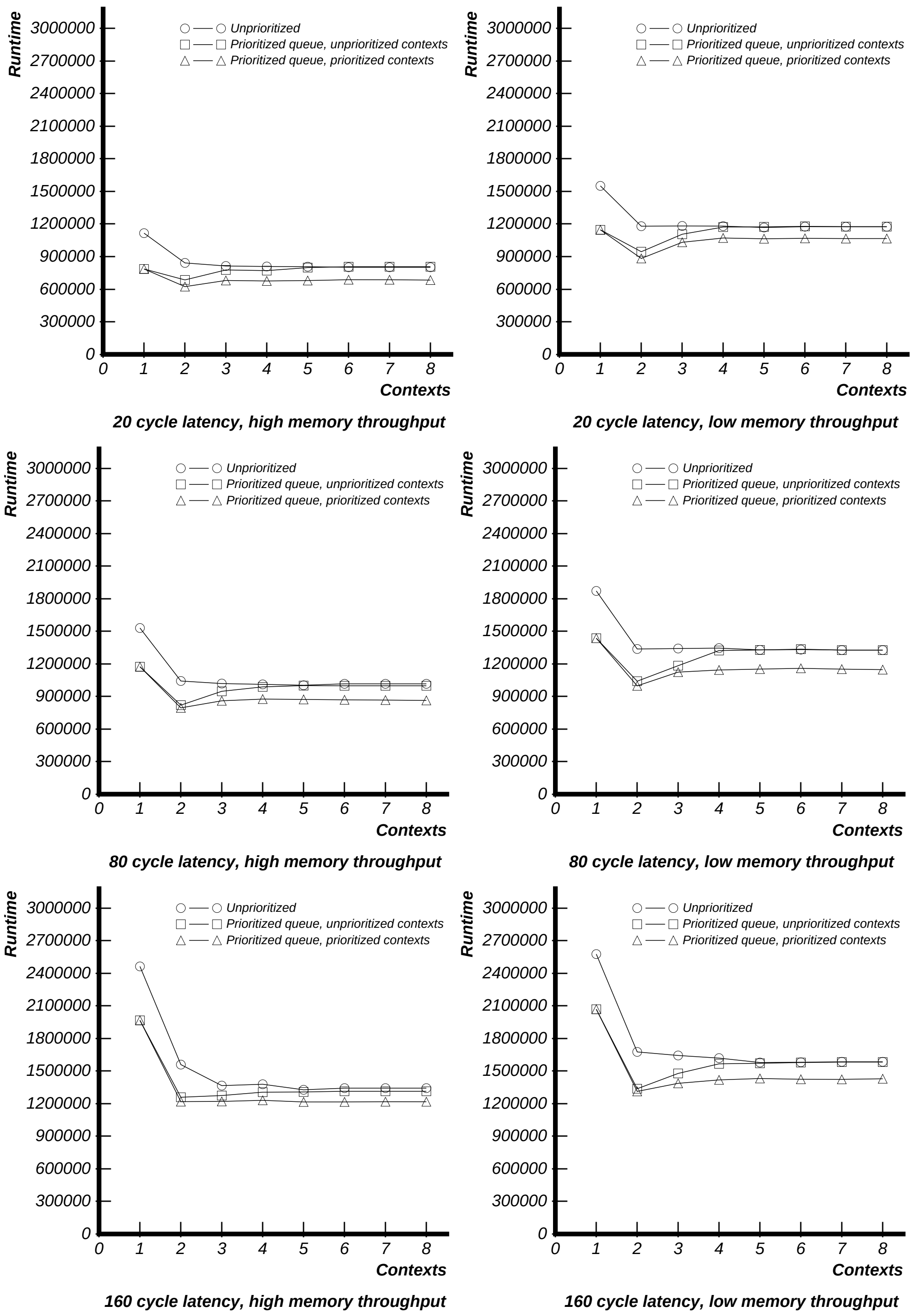


Figure 8.12: Performance of the 64X64 LUD benchmark for different memory latencies and throughputs.

mance improvement ranges from 30% for a latency of 20 cycles, to 20% for a latency of 160 cycles. Once there are 4 contexts or more the performance improvement is due to hardware prioritization since there are rarely any threads in the software queue. A third curve shown in Figure 8.12 that prioritizes the software queue, but uses round-robin scheduling for the contexts shows the effect of hardware prioritization. For 1 or 2 contexts the round-robin scheduling performs approximately the same as the fully prioritized case. However, performance becomes similar to the unprioritized case when the number of contexts increases to 4. Hardware prioritization leads to performance improvements ranging from 15% for a 20 cycle latency, to 9% for a 160 cycle latency. Improvements decrease with increasing latency since more contexts are needed to tolerate latency, and on any given context switch many contexts will be stalled.

Finally, it is interesting to note the effect of decreased memory controller throughput. Because at each stage of the computation one column of the row is read by all the processors to update their columns, the processor owning this row is a bottleneck. Reducing the memory controller throughput has an effect on performance, with runtimes increasing by about up to 45% for a latency of 20 cycles, 20% for a latency of 80 cycles, and only 4% for a latency of 160 cycles. The effect decreases as the latency increases because the increase in latency due to the memory bottleneck is a smaller percentage of the overall delay. The effect of prioritization remains approximately the same despite the decreased throughput.

8.2.4 Sparse LUD

Figures 8.14 and 8.15 show the results for the Sparse LUD benchmark for a varying number of contexts, using the *mat6* and *adjac25* matrices respectively. The *mat6* shows modest 5% to 16% improvement for 1 to 4 contexts, but the improvement falls of once there are many contexts. The *adjac25* case shows similar improvements of 6% to 15%, but only in the case that memory throughput is limited. Also, for a single context the prioritized case can be slightly worse due to worse cache performance, without the benefit of latency tolerance.

The results of this benchmark are rather inconclusive as to the benefits of prioritization. One of the main reasons for this is that the application is not critical path limited, and there are lots of threads with the same importance. There are however a couple of interesting points to note.

The performance improvement due to hardware prioritization falls off as the number of contexts increases, rather than increasing as has been the case in nearly all previous benchmarks. There are two factors that contribute to this. First, as the number of contexts increases, many threads can make progress, and a critical thread is less likely to be stuck in the software queue not making any progress. Second, because we are only prioritizing the *issue* of memory requests to the local memory or to remote memory modules, the prioritizing becomes ineffective when there are many non-local references and lots of memory and network traffic. Thus a request sent by a critical thread does not receive special treatment

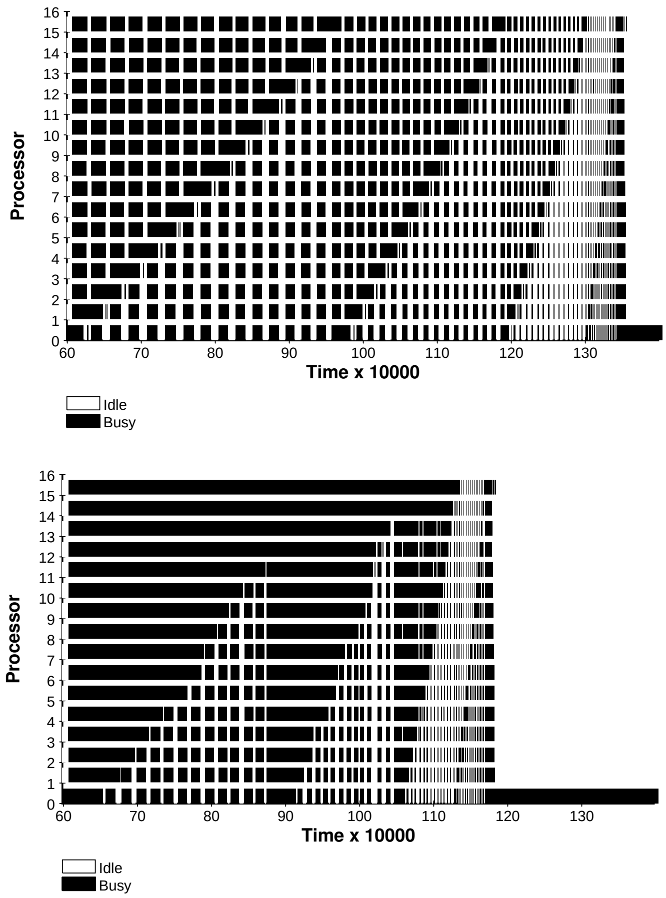


Figure 8.13: Processor lifelines for different versions of the LUD benchmark, 16 processors, 1 context per processor. a. Unprioritized. b. Prioritized.

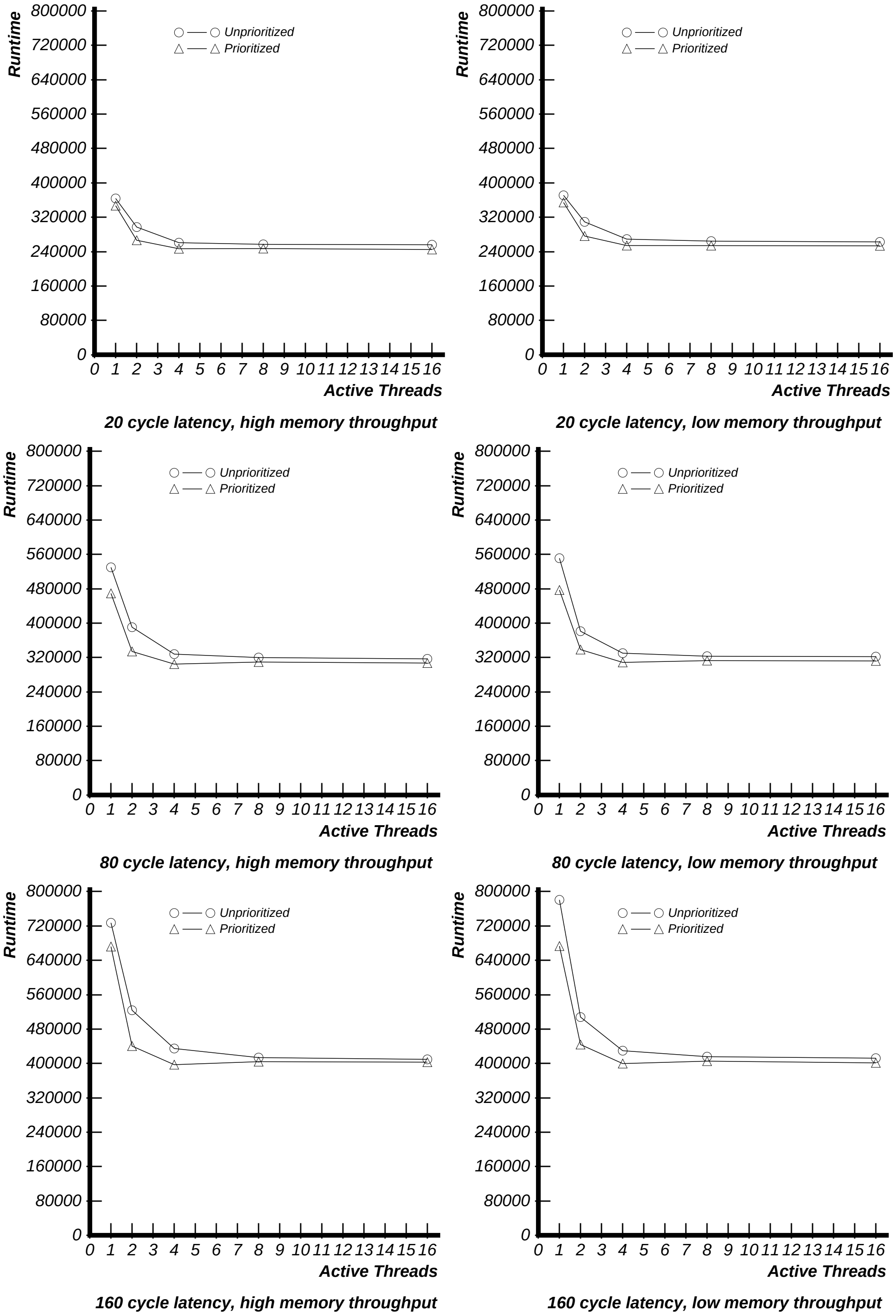


Figure 8.14: Performance of the sparse LUD benchmark using the *mat6* matrix, for different memory latencies and throughputs.

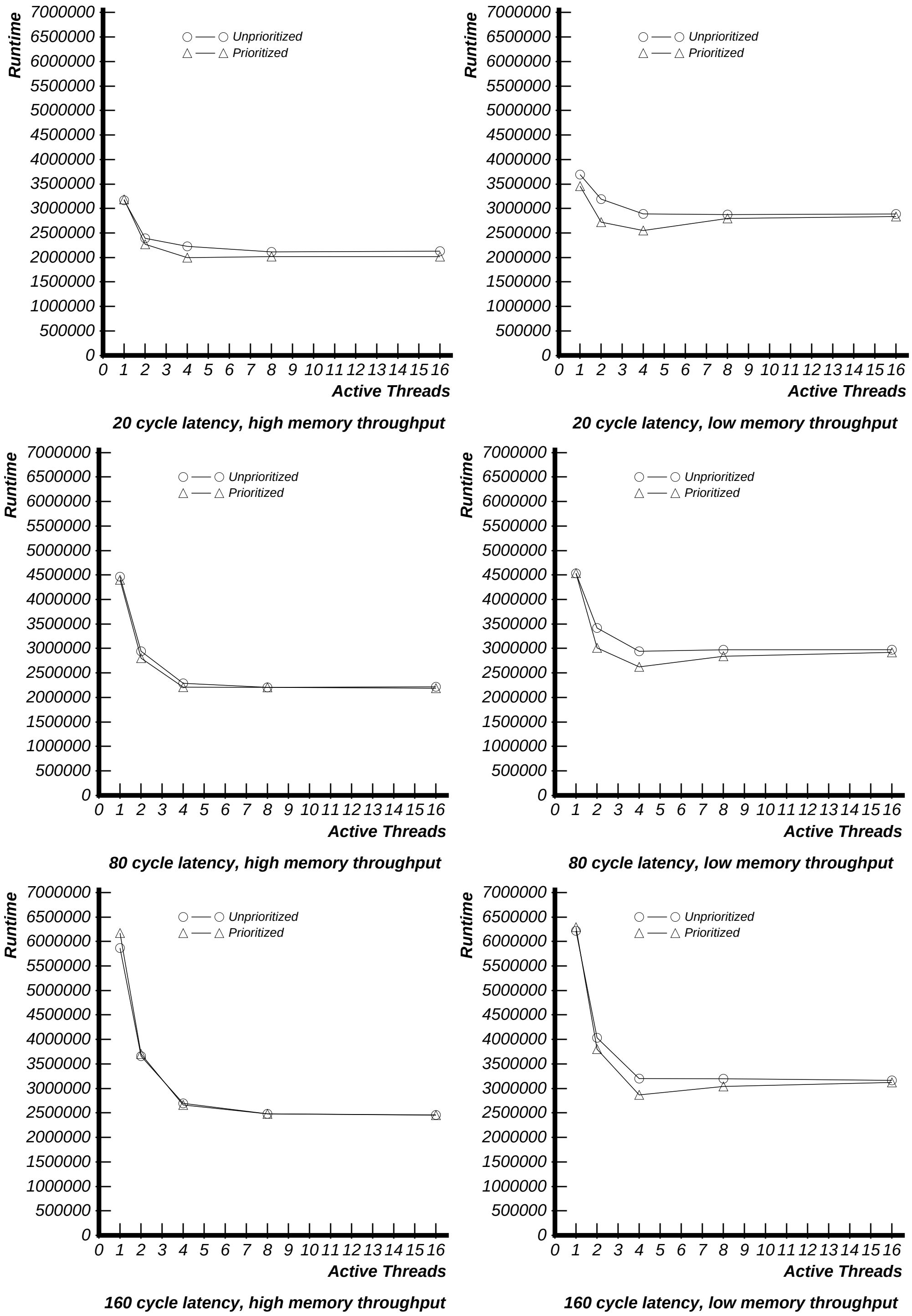


Figure 8.15: Performance of the sparse LUD benchmark using the *adjac25* matrix, for different memory latencies and throughputs.

in the network nor at the remote memory modules, and can easily be delayed by requests from less critical threads. The prioritization of messages in the network and the of incoming remote requests represents another level at which prioritization might have a possible impact, though this is not studied in this thesis.

Another interesting point to note is that the priority is based on a heuristic that does not take into account data placement and complex cache interactions. In particular, the OSA representation of the matrix is distributed across the processors without regards to which processors are going to be using which rows of the matrix. This data structure leads to many non-local references, and in particular many non-local write operations, which can have long and unpredictable delays. There are also numerous instances of false sharing. As a result, one path that has many fewer operations than another path can easily take much longer to execute that a path with fewer operations just by having better cache performance. It is clear that in order to correctly assign priorities to threads based on the critical path, the means of estimating relative path length must be accurate. In particular, it requires algorithms with well behaved cache behavior and data structures that do not lead to wildly different thread running times based on bad cache behavior.

8.3 Summary

The results from this chapter show that thread prioritization can have a large impact on runtime performance for certain problems, with runtime performance improvements ranging up to 37%. A number of characteristics of the program determine whether prioritization will have an impact on performance:

- Critical path characteristics: The problem must be critical path limited in order for prioritization to make a difference. Programs with lots of mostly independent, short threads, do not benefit much from prioritization.
- Available parallelism: If there is not enough parallelism, then the choice of threads is reduced, and the prioritization will have no effect. Note that insufficient parallelism also reduces the ability of multiple contexts to tolerate latency. Insufficient parallelism is sometimes a product of how the program is expressed, as in the Sparse Triangular Solve, rather than a lack of parallelism in the problem. This can often be solved with finer grain partitioning.
- Unpredictable memory reference patterns and execution times: If there are lots of threads all making remote references, then the effectiveness of prioritization of the processor pipelines and memory request issuing can be made ineffective by FIFO queuing in the network and at remote memory modules. Also, if the memory behavior causes the thread runtime to be very unpredictable, it is hard to assign priorities correctly to threads and thus the prioritization is less effective.

Finally, system characteristics can, but do not necessarily affect the impact of the prioritization:

- Long memory latencies: Long memory latencies typically decrease the impact of hardware prioritization because more contexts are needed to tolerate latency, and on any given context switch less contexts are ready to run.
- Memory throughput: Decreases memory throughput decreases overall performance for those applications that are bandwidth limited, or for which there is a memory bottleneck. Prioritization tends to be more important when the application is memory bandwidth limited.

Chapter 9

Reducing Software Scheduling Overhead

Thread scheduling overhead is the cost associated with inserting and deleting threads in scheduling queues, storing and retrieving thread arguments from memory, and allocating stack space for the thread. Reducing this overhead is important, and can be done by providing hardware support to do scheduling without software intervention, by minimizing the movement of data back and forth to memory, and by minimizing scheduling queue operations.

In this chapter we discuss the use of specialized hardware for reducing the overhead of scheduling message handlers (a thread scheduled in response to a message), as well as more general threads. Reducing the overhead of scheduling message handlers is important since they are often very short, and scheduling overhead can be a significant part of their runtime. Although specialized hardware works well for scheduling message handlers, it is not appropriate for general thread scheduling because it lacks flexibility. However, if the processor provides support for spawning a thread directly into another available context, it can reduce software scheduling overhead by avoiding the cost of software queue manipulation.

Section 9.1 discusses message handler scheduling and how hardware support can be used to reduce scheduling overhead to just a few cycles, provided these message handlers meet certain requirements that allow them to avoid deadlock. Section 9.2 discusses the more general requirements of thread scheduling that make software scheduling desirable despite the extra overhead involved. Finally, section 9.3 illustrates how multiple contexts can be used to reduce overhead of software scheduling using a radix sort example. Multiple contexts can reduce scheduling overhead by allowing the processor to avoid the cost of inserting and deleting threads from the software queue.

9.1 Message Handler Scheduling

In order to implement high performance message passing, a parallel processing node must have a network interface that allows a processor to both send and receive messages efficiently. For each message received a node must execute a message handler[1] corresponding to the type of message being received. A message handler can either directly execute the action required, or it can create and schedule a separate thread to execute the action at a later time. Thus each message requires a scheduling decision.

Executing the code directly in the message handler rather than scheduling a separate thread is preferable as it eliminates much of the overhead associated with creating and scheduling a thread. Very often the action required is a simple write operation, or a write, an increment, and a test in the case that the handler is delivering an argument and checking whether all the arguments have arrived [75, 21]. It is much quicker and more efficient to simply execute the operations, rather than going through the overhead of scheduling a task to do them — the cost of scheduling such a task can be many times the cost of doing the operations themselves. Furthermore, numerous hardware proposals allow very efficient message handling and reduce the overhead of handling messages to just a few cycles [23, 4, 78, 44]. Unfortunately, there are a number of situations in which a task may not be able to run to completion quickly. This can happen if:

1. The handler is a long task which takes many cycles to execute.
2. The handler requires a synchronization action to take place and this takes a long time. For instance, it has to acquire a lock, and the lock is unavailable.
3. The handler takes an exception that takes a long time to resolve (e.g a TLB miss).
4. The handler needs to use the network resources, and they are unavailable. In the case of shared memory, a write or read of shared memory can require the network, and can take a long time due to the required remote communication. In general, the task may need to send messages and the network output port may be busy.

Long running message handlers can cause network congestion and even network deadlock. If messages do not run to completion quickly, then messages can get backed up into the network and severely hurt network as well as overall performance. More seriously, deadlock may occur if a message handler has to send a message, since the availability of the output network interface for sending a message usually depends on all the processors continuously emptying their input network interfaces.

Using software scheduling in conjunction with hardware scheduling can eliminate many of the disadvantages of hardware scheduling on its own. For instance, the *active message*

[1] The message handler is usually in software, but can also be hardwired.

system [100] adopts a software convention that says that all handlers must be short and must run to completion. In particular, if a handler must send out a reply message, then it must not busy-wait if the outgoing message channel is unavailable. With *Optimistic active messages* [103], methods are run directly as handlers rather than going through the overhead of creating a thread, and if the method is unable to run to completion quickly then it is aborted and a separate thread is created to do the computation. The code is compiled specially to detect conditions in which an abort is required.

9.2 General Thread Scheduling

It may be tempting to try and use the hardware message handler scheduling mechanisms as a more general means to schedule threads. Taking the J-Machine [23] as an example, it is very efficient for a node to create another thread by sending a message to itself. The message is automatically sent, enqueued in memory, and scheduled in hardware. The overhead of creating such a thread in this way is just a few instructions. There are a number of problems with this approach, including the previously mentioned problems with running code inside message handlers, as well as the limited scheduling flexibility that can be implemented in hardware.

The same reasons that prevent all code from being run inside message handlers also prevent it from being easy to use the message scheduling hardware as a more general scheduler: threads may take a long time to run, they may require long latency synchronization operations, they make take an exception, and they may require the use of the network.

Hardware implementations limit the flexibility of the scheduling that can be done. For instance, whereas it is easy to design a small input queue to deal with some small number of incoming messages, it is more difficult to design a large queue that is more general and that will contain all the threads generated in the system. The later requires a general interface to the processor memory. Another limitation is that only FIFO or LIFO type scheduling policies are easily implementable. FIFO scheduling is particularly easy since the threads at the head of the queue can be executed as additional messages arrive at end of the queue. FIFO and LIFO scheduling is clearly inadequate or non-optimal for many problems. For instance, any sort of recursive program that schedules threads in FIFO manner will expand the call tree in breadth first fashion leading to excessive parallelism, and possible exhaustion of the memory of the machine [41, 45]. On the other hand, LIFO scheduling leads can unnecessarily restrict parallelism. With both FIFO and LIFO scheduling, critical threads can sit in the queue while threads at the head of the queue are executed.

As an example of the problems with simple scheduling schemes, consider a Traveling Salesman Problem, that finds the optimal tour in which the nodes of a graph with weighted edges can be traversed. For the purposes of this example we use a simple branch and bound algorithm. An exhaustive search is done, but the search tree is pruned by not looking at paths that have a cost that is greater than the best solution found so far. There are two

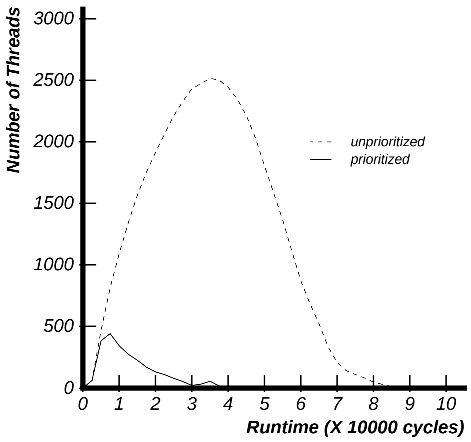


Figure 9.1: Number of active threads running an 8-city traveling salesman problem on 64 processors.

major parts to the program. The first part generates the next part of the search space to visit. New partial paths are created by adding each unvisited node to the current partial path, with a resulting increase in the partial cost of each tour. If the cost of a partial tour is higher than the best tour found so far, the path is pruned. Otherwise, a new thread is generated on a random node to continue the search using the new partial tour. Distributing threads to random nodes provides a primitive form of load balancing. The second major part of the program updates the best path found so far, and the bounds variable found on each of the processors. Whenever a tour is found that has a cost better than the current bound, the new path is saved, and the new cost bound is propagated to all the processors. Note that the bound on a processor may be out of date due to a pending update. The processor may do some extra work because its copy of the bound is stale, but the correct tour will still be found.

Figure 9.1 shows the runtime and the number of active threads as a function of time for two different scheduling strategies on a small 8 city problem. For the unprioritized case, a simple FIFO scheduling strategy is used that does not differentiate between the type of thread being run. For the prioritized case, threads were prioritized as follows: the threads that are responsible for propagating and updating the cost bound used for pruning at each processor have the highest priority, and the threads that are generating the search space have a priority equal to the number of nodes visited so far in the tour.

Two things are immediately noticeable. First, the runtime of the prioritized version is roughly half the runtime of the unprioritized version. Second, the maximum number of active threads, which corresponds to the peak memory utilization of the machine, is roughly 5 times less for the prioritized version than the unprioritized version. The improved runtime is due to two factors. First, in the prioritized case the processors typically have a more recent copy of the best cost bound found so far. The thread that updates the cost bound at each processor is run at top priority and is not sitting in the scheduling queue behind many other threads. Second, because threads that have visited more nodes have higher priority than those with fewer nodes, the search tends to proceed in depth first manner, rather than breadth first, which means that complete tours are found more quickly, bounds on the cost are found more quickly, and pruning is more effective. The improved memory utilization is also principally due to the prioritization of the search threads by the number of nodes visited so far. The computation tends to proceed in more of a depth first manner, resulting in more efficient memory utilization. It should be noted that the implementation is by no means ideal. The search thread priority could also take into account the current cost of the partial tour rather than just the depth. The policy of sending search threads to a random processor is non-optimal since it often sends work to processors who already have lots of work, and generates unnecessary network traffic — a more sophisticated load balancing technique would be helpful. However, this example illustrates the benefits of even fairly simple prioritization of threads.

The main point is that for general scheduling to be flexible it must be controllable in software. Simple FIFO and LIFO schemes that are relatively easy to implement in hardware, are not adequate for general scheduling. At best, hardware features allow certain important optimizations to take place. For instance, the direct dispatch to message handlers, and direct execution of code inside message handlers rather than creating and scheduling a thread, is an important optimization for message handlers that will run quickly and to completion, as in active messages. Having multiple hardware contexts allows further optimizations which are discussed in the next section.

9.3 Using Multiple Contexts

For certain types of computations, multiple contexts provide a unique opportunity for gaining most of the benefits of hardware scheduling and software scheduling at once. Having multiple contexts can allow potentially longer running message handlers to execute without the risk of deadlock, and without the larger overhead associated with software scheduling. Specifically, for each message one of the following actions is taken depending on the type of message:

1. The work is done directly in the handler if the thread is short and is guaranteed to run to completion.
2. If there is a free context, a thread is spawned directly into this context.
3. If there are no free contexts, a thread is inserted into the software scheduling queue and is run at a later time.

The more contexts are available for message handling, the more often a thread will be able to avoid option 3, and forego the overhead associated with running the software thread scheduler.

Two configurations are shown in Figure 9.2 that allow this to be possible. In the first configuration, each context can directly access the message queue. As long as there is more than one free context, the message handler can execute immediately without software queue overhead. Once there is only a single free context, we must be more conservative, and any thread that will not run quickly and to completion has to be scheduled in the software scheduling queue. The advantage of this system is that messages can be spawned to any context with a very small cost. The disadvantage is the complexity associated with determining when it is safe to execute the thread directly. For instance threads may be spawned on the node itself such that all the contexts become full, leaving no context to handle incoming messages. Thus the general scheduler must be aware that one context must always be available for handling messages.

In the second configuration, a single context handles messages as they arrive and carries out one of the three actions. One advantage of this is that the context aimed at handling incoming messages can be specially designed to be extremely efficient. In particular, the network interface can include a co-processor with a completely separate pipeline, and can be optimized to handle certain types of messages such as read request messages, or shared memory protocol messages [44, 75, 60]. The disadvantage is that when the message interface does decide to spawn a thread to an available context, it must copy arguments to the context before initiating execution. This is a relatively small cost to pay for the simplified network interface implementation and message handling optimizations, and this configuration is likely the more desirable of the two configurations.

9.3.1 Radix Sort Example

The radix sort algorithm sorts an array of integers one digit at a time, starting from the least-significant digit to the most significant, where a digit is represented by a field of b-bits digit [20]. One phase of the operation, the *scan phase*, requires 2^b parallel scan operations, one for each possible digit value. The data structure used is a distributed tree structure, distributed in a balanced fashion across the nodes so that each node contains a leaf node, and at most one internal node of the tree. For a single scan operation, messages first flow up the tree in a combine operation, and then back down the tree to distribute results to all the processors. For the purposes of the radix sort code, there are dependencies between the scan operations at the root node: a scan of a given index must wait at the root until all the previous indices have reached the root of the tree.

Tasks in the scan phase of radix sort are quite short, and consist in doing some small number of operations and tests, and sending off new messages to parent nodes or children nodes. The number of instructions for each task going up the tree is about 50 cycles, and for each

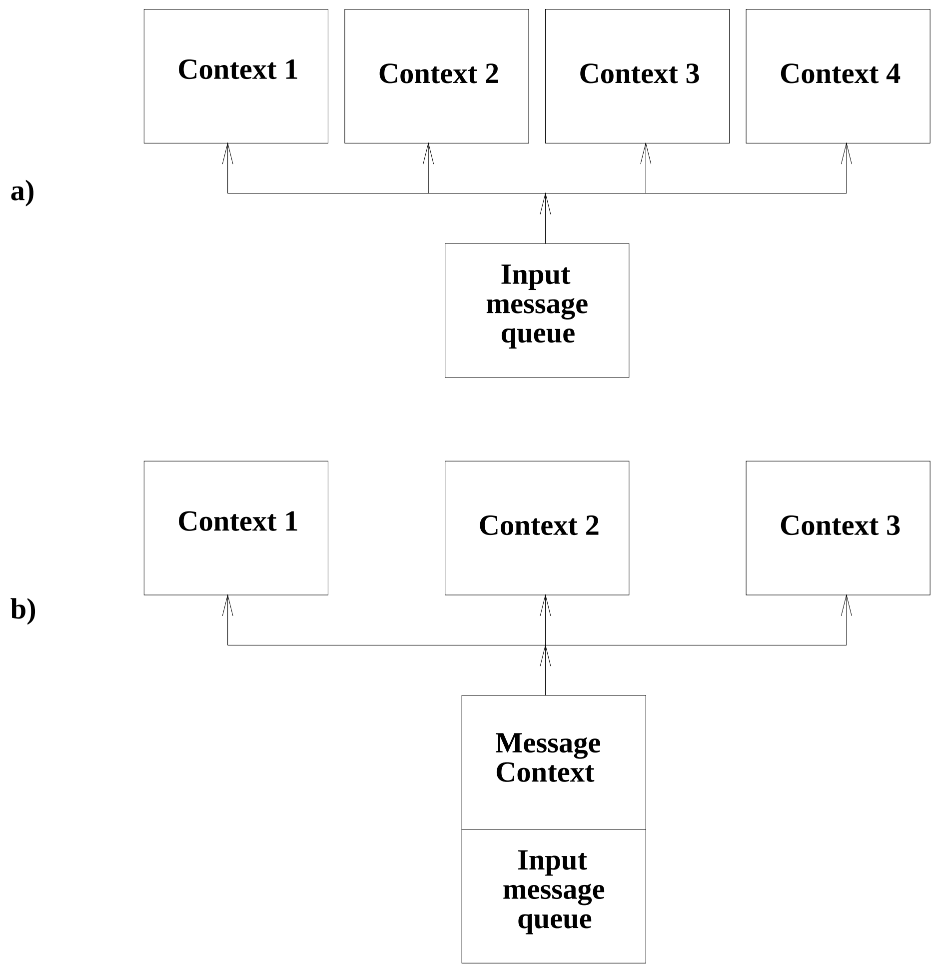


Figure 9.2: Message interface configurations. a. All contexts have equal access to the input message queue. b. One context has access to the input queue allowing certain message interface optimizations.

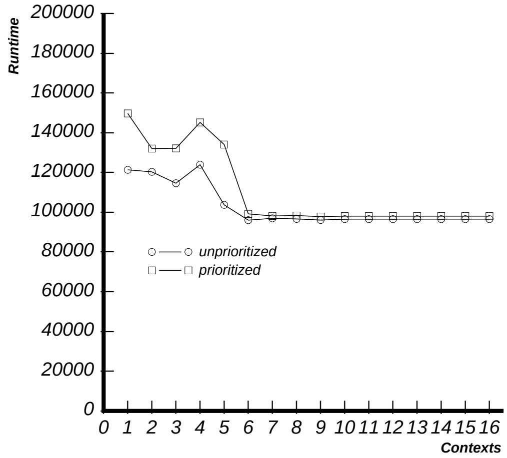


Figure 9.3: Radix sort scan phase for different numbers of contexts running on 64 processors. The digit size is six bits, requiring 64 parallel scans.

task going down the tree is about 35 cycles. Although the tasks are short, they do use the network, and thus these threads can take much longer to execute if the network interface is unavailable. if the code is implemented directly as a message handler, there is the risk of deadlock.

The scheduling of threads using multiple contexts is done as follows. A message handler reads the message from the network interface. If there is an available context, it spawns a thread directly into this context, otherwise a thread is put into the scheduling queue to be run at a later time. This corresponds to the message interface configuration of Figure 9.2b. If the processor had a free context it can spawn the thread directly into the context, and the cost is just the cycles required to copy the arguments and dispatch to the code. If no context is available, then the processor puts the thread into the scheduling queue to be run at a later time. The cost of scheduling a thread is the cost of inserting the thread into a software queue, including the copying of the arguments, later removing the thread from the queue, and reading the arguments into the context. On the order of about 50 extra cycles are required to do this software scheduling, about the same length as the minimum running time of the tasks themselves.

Figure 9.3 shows the results for an increasing number of contexts, and different scheduling schemes. For this problem FIFO scheduling achieves the best results. The prioritized version of the code attempts to ensure that the scans finish in order so as to avoid synchronization faults at the root, but this is already achieved by doing FIFO scheduling, and at a reduced scheduling cost. What is more interesting in this case is the performance increase that occurs as the number of contexts increases. Note that in this benchmark this is not due to any latency tolerance effects since shared memory is not simulated, but rather is due to avoided software scheduling costs. The software scheduling costs involved are important because the cost of scheduling is close to the cost of the task themselves. If each task was much longer, than the cost of scheduling it in software would be less significant. In the example shown, when there are sufficient contexts to avoid ever having to put a thread in the software queue, the multiple contexts lead to an 18% increase in performance in the unprioritized case, and a 34% increase in performance for the higher overhead prioritized case.

It is interesting to note that for this example, the runtime does not decrease uniformly with increasing contexts. For example, the runtime with 4 contexts is higher than the runtime with 2 or 3 contexts. This is because when the number of contexts changes, the thread generation pattern changes: some nodes may receive more messages faster, causing them to have to put more threads into their thread queue, leading to both higher scheduling overhead and imbalance in the amount of work each node has to do. Having sufficient contexts for all nodes to avoid using their software queue eliminates the problem.

9.3.2 General Problem Characteristics

In general, using multiple contexts in the scheduling of arriving messages will help performance when the tasks being scheduled are fairly small so that the scheduling overhead would be a substantial portion of their execution time, but have characteristics that make them risky to execute as active messages because of the risk of deadlock, or the possibility of backing up the network. In the case of the radix sort example, the tasks created are sending out messages into a network that is congested, and deadlock may result if the messages are sent directly from the message handlers. Furthermore, the handling of incoming messages is done faster if the message handlers can simply hand off the thread to a waiting context to execute, rather than execute it itself. This type of thread pattern occurs in many global type operations, such as scan, accumulate, broadcast, and barriers.

Finally, it is best if the number of threads on each processor remains below the number of contexts. This is because the contexts act as a thread cache, and as long as there are less threads than contexts, the processor never has to do insert or remove threads from a software queue.

9.4 Summary

This chapter briefly discusses issues related to hardware scheduling of messages and threads, and how multiple contexts can be used in certain situations to help eliminate overhead associated with the software scheduling of threads. In general, hardware scheduling does not allow the flexibility that is required for general thread scheduling. FIFO and LIFO scheduling policies that are easy to implement in hardware are generally not optimal for general thread scheduling and can lead to such problems as excessive memory utilization, and poor performance because critical threads are not given priority. For threads that are generally short but may be risky to execute directly as a message handler due to the possibility of long execution time or network resource requirements, using multiple contexts can improve performance by allowing threads to be spawned into separate contexts. Doing this eliminates most of the overhead incurred if the thread is put into a software scheduling queue, while making the thread safe to run since its execution has been decoupled from the handling of incoming messages.

Chapter 10

Conclusion

10.1 Summary

Multiple-context processors tolerate latency by rapidly switching between different threads of execution when a long latency operation takes place. This allows the processor to perform useful work in what would otherwise be idle cycles, thus increasing processor utilization and decreasing application runtime. Tolerating latency using a multiple-context processor requires a number of different scheduling decisions. First, we must decide which threads are loaded in contexts and eligible to execute instructions, and which are unloaded and waiting in a software scheduling queue. Second, we must decide which thread to execute next on each context switch. This thesis shows that both decisions are important for good performance.

Previous work on multiple-context processors considers round-robin context scheduling and uses processor utilization as a performance metric. This work has identified a number of factors that limit the performance of multiple-context processors, including network effects, and cache interference between multiple working sets. Other important performance limiting factors include the effect on critical path execution time, the effect of spin-waiting, and the effect of limited memory bandwidth. The naive sharing of processor resources among the threads due to round-robin scheduling is one of the main causes of the performance limiting problems associated with multiple-context processors.

In this thesis we propose thread prioritization as general scheduling mechanism that allows the user to easily and dynamically specify the preferred order in which threads should be executed, thus allocating processor resources more intelligently. we show that it is important to consider the effects that multiple contexts have not only on processor utilization, but also on the critical path. We show that thread prioritization is useful in addressing some of the limitations of multiple-context processors, including bad spin-waiting effects, negative cache interference effects, and critical path runtime effects.

The principal results of the thesis include the following:

- **Analytical models:** We develop analytical models that show how multiple contexts affect both processor utilization and the execution of the critical path of an application. Both processor utilization and critical path can affect overall performance. The models consider the effect of cache performance, network latency, spin-waiting synchronization, and limited memory and network bandwidth. Processor utilization suffers when there are too few contexts, but also when there are too many causing cache effects to become important. Having many contexts can lengthen the critical path execution time because a critical thread is delayed while other threads execute. Both spin-waiting and limited bandwidth reduce the effectiveness of multiple contexts.
- **Thread prioritization:** We present *thread prioritization* as a general thread scheduling mechanism which can help solve many of the problems associated with multiple-context processor thread scheduling. Thread prioritization is a temporal scheduling mechanism which helps decide when threads should run. Software prioritization allows us to decide which threads should be loaded and which should remain unloaded. Hardware prioritization allows us to choose a loaded thread on any given context switch. Hardware implementations can do context selection in a single cycle. Software implementations are more difficult to implement efficiently and can make the cost of thread selection unacceptably high.
- **Scheduling for good synchronization performance:** We show that thread prioritization can be used to substantially improve the performance of synchronization that uses spin-waiting. For simple synthetic benchmarks such as Test-and-Test_and_Set, barrier synchronization, and queue locks, runtime performance improvement range from 10% to 91% using 16 contexts.
- **Scheduling for good cache performance:** We present a number of techniques that improve the cache performance of multiple-context processors. *Data sharing* involves closely coordinating the threads running in each context so that they share common data in the cache. *Favored thread execution* uses thread prioritization to dynamically allow only the minimum number of contexts required to tolerate latency to be running. Cache performance improves because the scheduling minimizes the number of working sets in the cache. Runtime improvements range up to 50% for bandwidth limited applications using 16 contexts.
- **Scheduling for good critical path performance:** We show how thread prioritization can help schedule threads based on the critical path. If the problem is critical path limited then prioritization can have a large impact, 37% for one benchmark using 16 contexts. If the problem is not critical path limited, or there is not sufficient parallelism to keep the multiple contexts busy, prioritization will have little effect.
- **Using multiple contexts to reduce software scheduling overhead:** We show how in certain situations multiple contexts can be used to eliminate software scheduling overhead associated with safely scheduling threads in response to incoming messages.

These results show that by carefully controlling the allocation of processor resources to the different threads, including pipeline resources, bandwidth resources, and cache resources, some of the deficiencies of multiple-context processors can be overcome, thus making them an even more effective latency tolerance and performance enhancing mechanism.

10.2 Future Work

As in any PhD Thesis, answering interesting questions raises a host of related questions. A number of directions that should be explored include the effect of prioritization on a more extensive set of applications, automating the thread prioritization process, exploring other uses of thread prioritization including ways in which it could be used by the operating system, and determining how different latency tolerance techniques can be used together to offer the best possible latency tolerance.

10.2.1 Applications

Most of the applications in this thesis are micro-benchmarks that are either synthetic in nature, or represent a computationally intensive kernel of a real application. Each one is carefully chosen to have characteristics that illustrate a specific type of scheduling problem in as much isolation as possible, while at the same time representing characteristics that are found in real programs. Doing this allows us to see how effectively our thread scheduling techniques and mechanisms deal with specific scheduling problems.

Complete applications present a combination of interacting effects that may not arise in small kernels or synthetic benchmarks and that it is important to understand. A number of these effects were observed in some of our larger benchmarks such as LUD, and sparse LUD. Improving the performance of complete applications lends weight to the conclusion that the mechanisms are indeed generally useful. A variety of benchmarks have gained popularity over the past few years, in particular the Splash benchmarks from Stanford [87], and it would be useful to port and modify these benchmarks for a multithreaded prioritized system.

10.2.2 Automated Thread Prioritizing

Tools to automatically assign priorities to threads, rather than have the user specify the priority with program annotations and library calls, would be very useful. In many cases this type of prioritization is easy to do. In the case that the program can be represented as a static DAG, a compiler can easily use heuristics to assign static priorities to the threads. Similarly, it is straightforward to automate some of the techniques used to improve cache performance in parallel loop code. Priorities can easily be assigned automatically to achieve good cache performance using favored thread execution. The general problem of prioritizing threads in an arbitrary program is more difficult.

10.2.3 Other Uses of Thread Prioritization

This thesis presented and evaluated thread prioritization as a temporal scheduling mechanism used to decide when a thread should run on a single processor. However, many other potential uses of thread prioritization are possible. The priorities could be used to make certain thread placement decisions. For instance, a load balancer might use the priorities to decide which threads to migrate. The load balancer may decide to migrate lower priority threads so that higher priority threads would not be delayed by the migration overhead. In this context the priority might be an indication of the importance of the thread to the critical path, or may be an indicator of the affinity of the thread for a given processor.

It is also possible that the thread prioritization could be used by the operating system. If space sharing is used, threads from a given application may have to be moved off one node and on to another. The priority can help make decisions about how threads should be redistributed across the processors, and how the schedule should be reorganized on the remaining processors. Further, the multiple contexts and thread prioritization could be used to run operating system tasks in parallel with user code running other tasks, or even to have multiple user tasks running at the same time at different priorities. Further study is needed to determine how prioritization might help operating system scheduling.

10.2.4 Combining Latency Tolerance Strategies

Multithreading is only one method of tolerating latency. Others include prefetching and relaxed memory consistency models. Gupta el. al. showed that allowing multiple outstanding references per thread helps the performance of both prefetching and multithreading [37]. In particular in the context of multithreading it allows longer run lengths between context switches and it allows more memory bandwidth resources to be devoted to a single thread. This reduces the number of contexts required to tolerate latency, and improves single thread performance. Gupta's study also showed that using both prefetching and multithreading at the same time can in fact hurt performance if done naively. It is clear however that both prefetching and multithreading have the potential to be complementary: multithreading can do a good job of tolerating synchronization latencies and irregular memory reference patterns, whereas prefetching can do a good job in tolerating regular reference patterns. Determining the optimal combination of latency tolerance techniques for any given application remains an open problem.

10.3 Epilogue

Although commercial microprocessor developers have not yet embraced the concept of having multiple hardware contexts, there are ever more compelling reasons for them to seriously think about the benefits multiple contexts provide: tolerating long latencies, providing more instructions to keep multiple functional units and long pipelines busy, and supporting fast interrupts and fast multiprocessor network interfaces. Studying the impact of multiple contexts on computer architecture, on application performance, on compilers, and on system software will continue to demonstrate these benefits, and lead to the acceptance of multiple-context processors as a commercially viable approach to enhancing performance.

Appendix A

A Fast Multi-Way Comparator

To choose a context to execute based on priorities, a multiple-context processor must compare the priorities of all the contexts and find out which are the highest priority ready threads. This appendix presents the schematic design of a circuit that compares C, N-bit priorities using carry-select techniques. This is faster than the simpler C-way ripple comparator presented in Section 4.1.

Figures A.1 through A.4 show the design of the circuit. Figure A.1 shows a bit-slice of a ripple-compare circuit that compares two priorities. If the two priorities are equal then both outputs, $Co0$ and $Co1$, are 1. Otherwise the output corresponding to the higher priority thread is 1, and the other is 0. Cascading N single-bit comparators forms an N-bit comparator.

The delay of a ripple comparator grows linearly with the number of bits and it is desirable to use carry-select techniques [107] to reduce delay when N becomes large. We use the COMPARE/SELECT circuit of Figure A.2 in our carry-select comparator. This circuit takes as input an F-bit wide field of each priority, as well as the results $Ci0$, and $Ci1$, from the comparison of the higher order bits of the two priorities. If the comparison of the higher order bits has already determined the larger priority then this result goes directly to the output, otherwise the result of comparing the next F-bits of the priorities goes to the output. This circuit also outputs the F-bits of the larger priority so that this larger priority can be used in additional comparisons. Figure A.3 shows a 16 bit comparator using three carry-select stages, of length 5, 5, and 6. This increases speed because the fields of the priority are compared in parallel in each of the three stages, and the result selected based on these results. To first order, the carry 16-bit carry-select comparator has about half the delay of a ripple comparator, based on counting the logic levels that the signals have to propagate through.

The circuit of Figure A.4 compares priorities from 4 contexts. It has an output for each context, $Co0$ through $Co3$, and an output is a 1 if the corresponding context has the highest

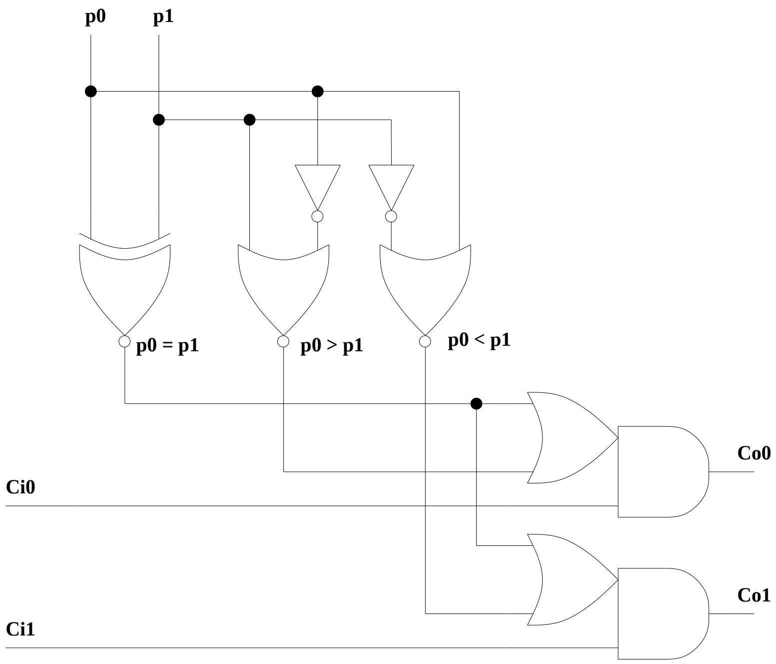


Figure A.1: Bit-slice of a ripple-compare circuit. Cascading N bit-slices forms an N-bit RIPPLE-COMPARE circuit.

priority. The first level of comparators compares $P0$ with $P1$, and $P2$ with $P3$. The second level then compares the highest priorities from the first level. For more than 4 contexts the structure is easily generalizable to a tree of comparators. If C is the number of contexts, we need $log_2 C$ levels of comparators. Note that the second level comparator does not have to wait until the first level of comparators has completed, but can immediately begin comparing the high order bits as they become available.

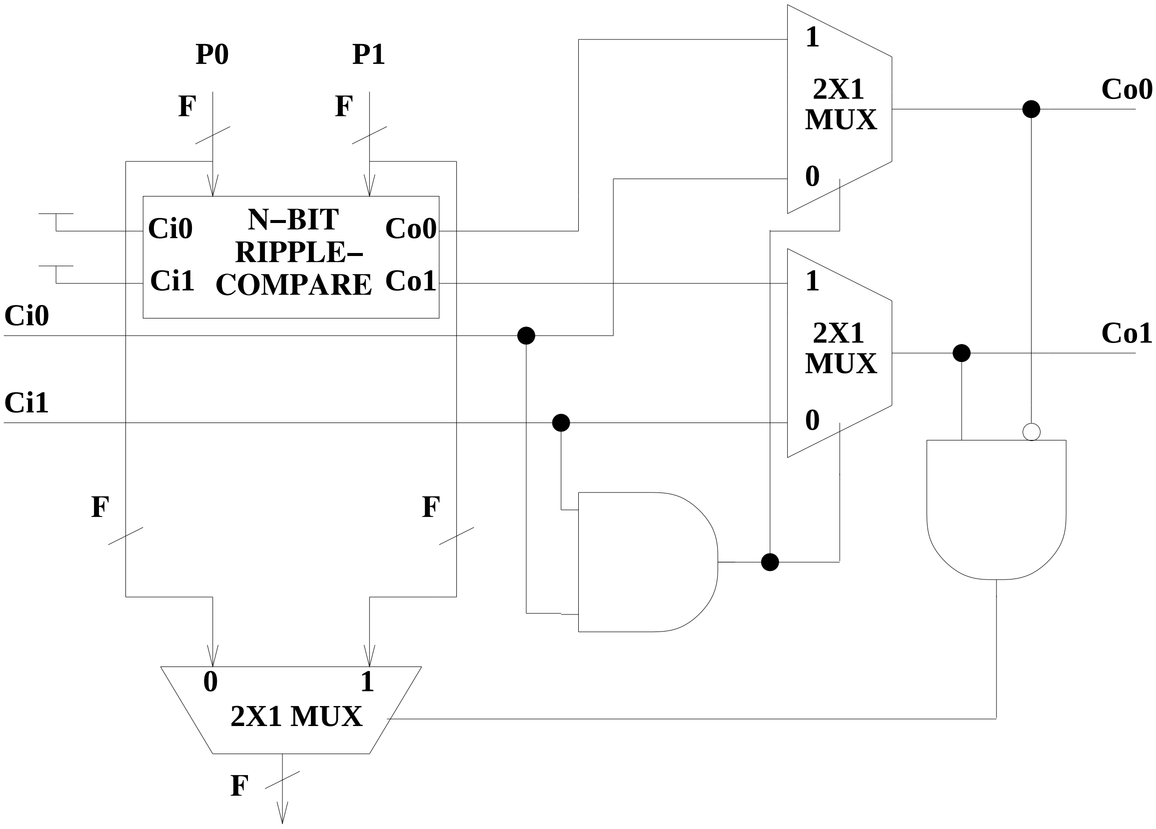


Figure A.2: F-bit COMPARE/SELECT circuit used in the carry-select comparator.

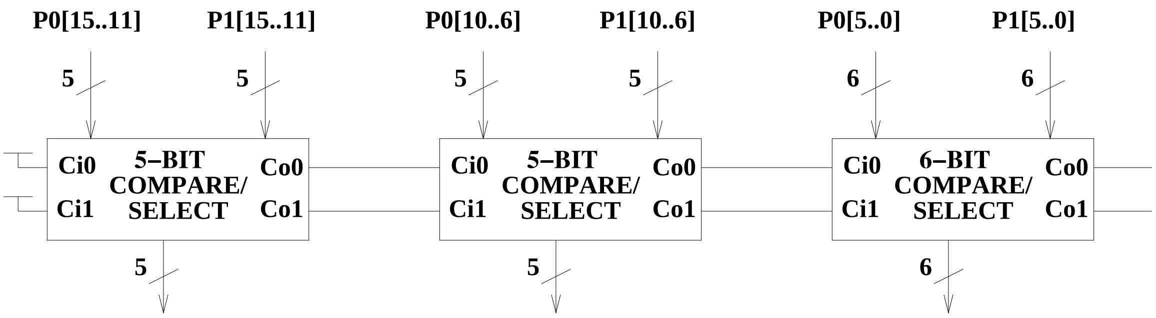


Figure A.3: 16-bit carry-select COMPARATOR circuit using 3 COMPARE/SELECT comparators of length 5, 5, and 6.

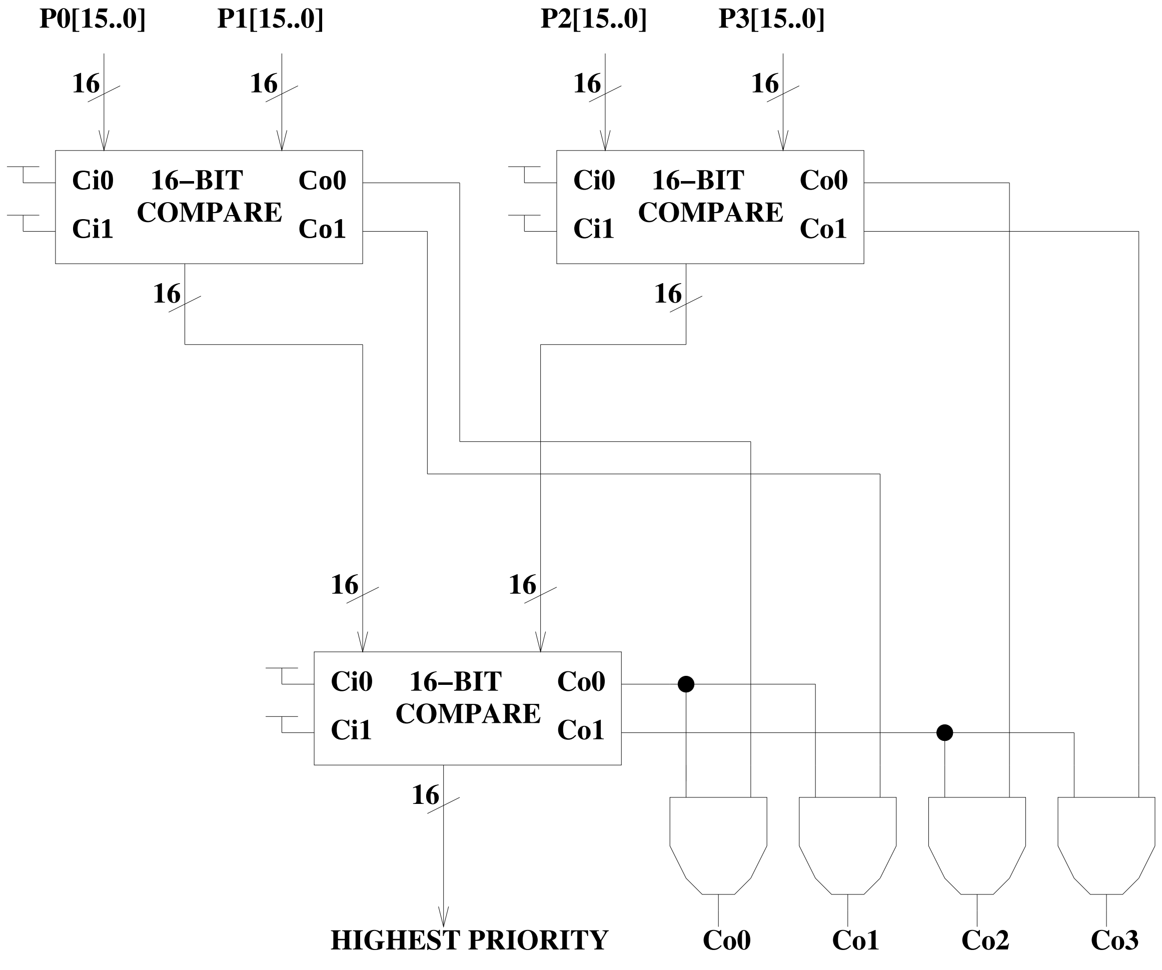


Figure A.4: 4-priority comparison circuit.

Bibliography

[1] Anant Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, October 1992.

[3] Anant Agarwal and Mathews Cherian. Adaptive Backoff Synchronization Techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 396–406. ACM, June 1989.

[4] Anant Agarwal et al. The MIT Alewife machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[5] 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. ACM, 1990.

[6] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, pages 1–6, June 1990.

[7] Robert Alverson, David Callahan, Brian Koblenz, and Burton Smith. Exploiting Heterogeneous Parallelism on a Multi-Threaded Multi-Processor. In *Proceedings of the International Conference on Supercomputing*, June 1992.

[8] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[9] Arvind and David E. Culler. Managing Resources in a Parallel Machine. In *Proceedings of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture*. North-Holland Publishing Company, July 1985.

[10] Jean-Loup Baer and Tien-Fu Chen. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proceedings of Supercomputing '91*, pages 176–186, November 1991.

[11] Robert D. Blumofe. Managing Storage for Multithreaded Computations. Technical Report MIT/LCS/TR-552, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1992.

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[13] 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, Gold Coast, Australia, May 1992. ACM.

[14] Richard P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[15] Eric A. Brewer, Chrysanthos N. Dellarocas, et al. Proteus: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1991.

[16] 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. ACM, April 1991.

[17] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[18] David L. Chaiken. Cache Coherence Protocols for Large-Scale Multiprocessors. Technical Report MIT/LCS/TR-489, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1990.

[19] Frederic T. Chong, Shamik D. Sharma, Eric A. Brewer, and Joel Saltz. Multiprocessor Runtime Support for Irregular DAGs. In *Toward Teraflop Computing and New Grand Challenge Applications*. Nova Science Publishers, Inc., 1995. To appear.

[20] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, Cambridge, MA, 1990.

[21] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175. ACM, April 1991.

[22] William J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6), June 1990.

[23] William J. Dally, J.A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[24] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–53, May 1987.

[25] Chrysanthos N. Dellarocas. A High-Performance Retargetable Simulator for Parallel Architecctures. Technical Report MIT/LCS/TR-505, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1991.

[26] Murphy Devarakonda and Arup Mukherjee. Issues in Implementation of Cache Affinity Scheduling. In *Proceedings Winter 1993 USENIX Conference*, pages 345–357, January 1992.

[27] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 432–442, June 1986.

[28] Ian S. Duff, Roger G. Grimes, and John G. Lewis. User's Guide for the Harwell-Boeing Sparse Matrix Collection. Technical Report TR/PA/92/86, CERFACS, Toulouse, France, october 1992.

[29] Geoffrey Fox et al. *Solving Problems on Concurrent Computers.* Prentice Hall, 1988.

[30] M. Garey and D. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[31] Apostolos Gerasoulis and Tao Yang. A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[32] 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. ACM, April 1991.

[33] 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 International Symposium on Computer Architecture*, pages 15–26, May 1990.

[34] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75. ACM, April 1989.

[35] Edward H. Gornish, Elana D. Granston, and Veidenbaum Alexander V. Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies. In *International Conference on Supercomputing*, pages 354–368, June 1990.

[36] Gary Graunke and Thakkar Shreekant. Synchronization Algorithms for Shared-Memory Multiprocessors. *Computer*, 23(6):60–69, June 1990.

[37] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263. ACM, May 1991.

[38] Linley Gwennap. 620 Fills Out PowerPC Product Line. *Microprocessor Report*, 8(14), October 1994.

[39] Linley Gwennap. UltraSparc Unleashes SPARC Performance. *Microprocessor Report*, 8(13), October 1994.

[40] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2), February 1995.

[41] Robert H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[42] Robert H. Halstead and Tetsuya Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *15th Annual Symposium on Computer Architecture*, pages 443–451. IEEE Computer Society, May 1988.

[43] John L. Hennessey and Norman P. Jouppi. Computer Technology and Architecture: An Evolving Interaction. *Computer*, 24(9):18–29, September 1991.

[44] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 111–122. ACM, October 1992.

[45] Waldemar Horwat. Concurrent Smalltalk on the Message-Driven Processor. Technical Report 1321, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, September 1991.

[46] Kirk L. Johnson. The Impact of Communication Locality on Large-Scale Multiprocessor Performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 393–402, Queensland, Australia, may 1992. ACM.

[47] 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. ACM, May 1990.

[48] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 41–55, 1991.

[49] A. H. Karp. Programming for Parallelism. *Computer*, 20(5):43–57, May 1987.

[50] Stephen W. Keckler and William J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 202–213, Queensland, Australia, May 1992. ACM.

[51] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53. ACM, May 1991.

[52] Kathleen Knobe, Joan D. Lukas, and William J. Dally. Dynamic Alignment on Distributed Memory Systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, number TR 92-8, pages 394–404. ACPC, July 1992.

[53] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[54] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 54–63, San Diego, May 1993.

[55] David Kranz, Beng-Hong Lim, and Anant Agarwal. Low-Cost Support for Fine-Grain Synchronization in Multiprocessors. Technical Report MIT/LCS/TR-470, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA, June 1992.

[56] David Kroft. Lockup-Free Instruction Fetch/Prefetch Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–97, 1981.

[57] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*. ACM, October 1992.

[58] John D. Kubiatowicz. Closing the Window of Vulnerability in Multiphase Memory Transactions: The Alewife Transaction Store. Technical Report MIT/LCS/TR-594, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge MA, February 1993.

[59] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance in Large-Scale Multiprocessors. In *International Symposium on Shared Memory Multiprocessing*, pages 91–101, 1991.

[60] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, et al. The Stanford FLASH Multiprocessor. In *Proc. 21st International Symposium on Computer Architecture*, pages 302–313. IEEE, April 1994.

[61] Monica Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimization of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 318–328, 1988.

[62] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, c-28(9):690–691, September 1979.

[63] James Laudon, Anoop Gupta, and Mark Horowitz. Architectural and Implementation Tradeoffs in the Design of Multiple Context Processors. Technical Report CSL-TR-92-523, Computer Systems Laboratory, Stanford University, CA 94305, May 1992.

[64] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318. ACM, October 1994.

[65] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Data Prefetching in Shared Memory Multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 28–31, August 1987.

[66] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of 4.3BSD UNIX Operating System.* Addison Wesley, 1990.

[67] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

[68] Steven Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 200–211, June 1992.

[69] Evangelos P. Markatos and Thomas J. LeBlanc. Load Balancing vs. Locality Management in Shared-Memory Multiprocessors. Technical Report 399, The University of Rochester Computer Science Department, October 1991.

[70] Evangelos P. Markatos and Thomas J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. In *Proceedings of Supercomputing '92*, pages 104–113, November 1992.

[71] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. Technical Report 342, University of Rochester, Computer Science, Rochester NY, April 1990.

[72] Eric Mohr, David Kranz, and Robert H. Jr. Halstead. Performance Tradeoffs in Multithreaded Processors. Technical Report MIT/LCS/TR-449, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA, June 1991.

[73] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software Controlled Prefetching in Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

[74] Shashank S. Nemawarkar, R. Govindarajan, Guang R. Gao, and Vinod K. Agarwal. Performance Analysis of Multithreaded Multiprocessors using an Integrated System Model. Technical Report ACAPS Technical Memo 84-1, McGill University, July 1994.

[75] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. Computation Structures Group Memo 325-1, Laboratory for Computer Science, Massachusetts Institute of Technology, November 1991.

[76] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224–235, San Diego, California, May 1993. IEEE.

[77] Peter R. Nuth and William J. Dally. A Mechanism for Efficient Context Switching. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, pages 301–304. IEEE, October 1991.

[78] Gregory M. Papadopoulos and David E. Culler. Monsoon: an Explicit Token-Store Architecture. In *The 17th Annual International Symposium on Computer Architecture*, pages 82–91. IEEE, 1990.

[79] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[80] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, San Diego, May 1993. ACM.

[81] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *ACM Symposium on Parallel Algorithms and Architecture*, pages 169–178. ACM, July 1990.

[82] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[83] Vivek Sarkar. Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors. Technical Report CSL-TR-87-328, Stanford University, April 1987.

[84] C. Scheurich and M Dubois. Lockup-Free Caches in High-Performance Multiprocessors. *Journal of Parallel and Distributed Computing*, (11):25–36, 1991.

[85] Charles W. Selvidge. Compilation-based Prefetching for Memory Latency Tolerance. Technical Report MIT/LCS/TR-547, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1992.

[86] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[87] Jasinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[88] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Vol. 298 Real-Time Signal Processing IV*, pages 241–248. Denelcor, Inc., Aurora, CO, 1981.

[89] James E. Smith. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Computer Architecture Symposium*, pages 112–119, 1982.

[90] Mark S. Squillante and Edward D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. Technical Report 89-06-01, Department of Computer Science and Engineering, University of Washington, May 1990.

[91] Mark S. Squillante and Randolph D. Nelson. Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling. In *ACM Signmetrics Conference on Measurement and Modelling of Computer Systems*, pages 143–155, 1991.

[92] Per Stenstrom, Fredrik Dahlgren, and Lars Lubdberg. A Lockup-free Multiprocessor Cache Design. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 161–165, 1991.

[93] Przybylski Steven. DRAMs For New Memory Systems (Part 3). *Microprocessor Report*, 7(4):22–26, March 1993.

[94] Ricardo Telichevesky. A Numerical Engine for Distributed Sparse Matrices. PhD Thesis, Massachusetts Institute of Technology, Cambridge MA, January 1994.

[95] Radhika Thekkah and Susan J Eggers. The Effectiveness of Multiple Hardware Contexts. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 328–337. ACM, October 1994.

[96] Radhika Thekkath and Susan J. Eggers. Impact of Sharing-Based Thread Placement on Multithreaded Architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 176–186, Chicago, Illinois, April 1994. ACM.

[97] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[98] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, 1989.

[99] Ten H. Tzen and Lionel M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, pages 87–98, January 1993.

[100] Thorsten Von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pages 256–266. IEEE, 1992.

[101] Carl A. Waldspurger and William E. Weihl. Register Relocation: Flexible Contexts for Multithreading. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 120–130, San Diego, May 1993. ACM.

[102] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.

[103] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995. To appear.

[104] Yung-Terng Wang and Robert J. T. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[105] 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, Jerusalem, Israel, May 1989. ACM.

[106] Michael Weiss, C. Robert Morgan, and Zhixi Fang. Dynamic Scheduling and Memory Management for Parallel Programs. In *Proceedings of the 1988 International Conference on Parallel Processing, Volume II Software*, pages 161–165, 1988.

[107] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, 1985.

[108] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[109] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[110] Donald Yeung and Anant Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. In *Principles and Practices of Parallel Programming, 1993*, pages 187–197, San Diego, CA, May 1993. IEEE. Also as MIT/LCS-TM 479, October 1992.

[111] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[112] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.