

**Evaluation of Multithreading and Caching  
in Large Shared Memory Parallel Computers  
\*\*PhD Dissertation\*\***

*Robert Francis Boothe*

**Report No. UCB/CSD 93/766**

**July 1993**

**Computer Science Division (EECS)  
University of California  
Berkeley, California 94720**



Evaluation of Multithreading and Caching  
in Large Shared Memory Parallel Computers

by

Robert Francis Boothe

B.S. (University of California at San Diego) 1985

M.S. (University of California at Berkeley) 1989

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Abhiram G. Ranade, Chair

Professor David E. Culler

Professor Ronald W. Wolff

1993

## Acknowledgements

There are many people I would like to thank for helping make my stay at Berkeley productive, rewarding, and enjoyable. First is my advisor, Abhiram Ranade, for suggesting this research area and for guiding, encouraging, and supporting my research. Next are my office mates M. T. Raghunath and Jeff Rothman, for the many discussions on research ideas, related work, politics, vegetarianism, and the densest packaging of aluminum cans. I also wish to thank my second and third readers, David Culler and Ronald Wolff, for reading through this lengthy thesis. David Culler's extensive comments helped to significantly improve the presentation of this work.

Besides those who helped influence the content and presentation of my thesis, I would also like to thank the many friends I have developed while at Berkeley. Thanks to Mike Meighan, Seth Teller, and Eric Enderton for friendship, encouragement, and appreciation of T-shirts. And thanks to the members of the Hillegass House (Ramon Caceres, Will Evans, John Hartman, Mike Hohmeyer, Steve Lucco and Ken Shirriff) for many dinners, parties, and Simpson's episodes.

Finally, I wish to thank my parents and particularly my best friend (and wife) Alison Kisch for their love, understanding, and patience during those times when we did not get to spend as much time together as we would have liked because I was busy working on this research.

This work was supported in part by the Air Force Office of Scientific Research (AFOSR/JSEP) under contract F49620-90-C-0029 and NSF Grant Number 1-442427-21936. Computing resources were provided by NSF Infrastructure Grant number CDA-8722788. Their support is gratefully acknowledged.



## Abstract

# Evaluation of Multithreading and Caching in Large Shared Memory Parallel Computers

by

Robert Francis Boothe

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Abhiram G. Ranade, Chair

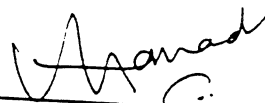
Shared memory multiprocessors are considered among the easiest parallel computers to program. However, building shared memory machines with thousands of processors has proven difficult. Two main problems are the long latencies to shared memory and the large network bandwidth required to support the shared memory programming style.

In this dissertation, we quantify the magnitude of these problems and evaluate multithreading and caching as mechanisms for solving them. Multithreading works by overlapping communication with computation, and caching works by filtering out a large fraction of the remote accesses.

We evaluate several multithreading models using simulations of eight benchmark applications. On systems with multithreading but without caching, we have found that the best results are obtained for the explicit-switch multithreading model. This model provides an explicit context switch instruction that allows the compiler to select the points at which context switches occur. Our results suggest that a 200 cycle memory access latency can be tolerated using multithreading levels of 10 threads or less per processor. On systems with both multithreading and caching, we have found that the switch-on-miss multithreading is best. For this model, our results suggest that a 200 cycle memory access latency can be tolerated using multithreading levels of 3 threads or less per processor.

We show that by using multithreading techniques, systems both with and without

caching are able to tolerate long latencies and achieve execution efficiencies of 80%. The difference between systems with and without caching is that the systems with caching (if properly configured) use an order of magnitude less network bandwidth. For large machines, the network cost will be a large factor in the cost of the machine, and thus the cost and complexity of cache coherency will be offset by the reduction in the network bandwidth requirement.



---

Professor Abhiram G. Ranade  
Dissertation Committee Chair

# Contents

<b>List of Figures</b>	<b>vi</b>
------------------------	-----------

<b>List of Tables</b>	<b>viii</b>
-----------------------	-------------

<b>1 Introduction</b>	<b>1</b>
1.1 The Latency Problem . . . . .	2
1.2 Solutions to the Latency Problem . . . . .	4
1.3 Overview of Previous Multithreading Work . . . . .	6
1.3.1 Fast Pipeline . . . . .	7
1.3.2 Hiding Memory Latency . . . . .	7
1.3.3 Adding Caches . . . . .	8
1.4 Limited Bandwidth . . . . .	9
1.5 Overview of Thesis . . . . .	12
<b>2 Methodology</b>	<b>14</b>
2.1 Machine Model . . . . .	14
2.1.1 Network . . . . .	16
2.1.2 Processor . . . . .	18
2.1.3 Programming Language . . . . .	19
2.2 Benchmark Applications . . . . .	21
2.2.1 Sieve . . . . .	22
2.2.2 Blkmat . . . . .	22
2.2.3 Sor . . . . .	24
2.2.4 Ugray . . . . .	25
2.2.5 Water . . . . .	25
2.2.6 Locus . . . . .	26
2.2.7 Mp3d . . . . .	27
2.2.8 Barnes . . . . .	27
2.2.9 Summary of Application Characteristics . . . . .	28
2.3 Simulation System . . . . .	29
2.3.1 The Simulator . . . . .	29
2.3.2 Simulation Constraints . . . . .	30
2.3.3 Revised Machine Model . . . . .	32

<b>3 Behavior of Multithreading</b>	
3.1 Multithreading Model	34
3.1.1 Analysis Under Constant Run-Lengths	34
3.1.2 More Complex Distributions	35
3.2 Applications' Run-Length Distributions	37
3.3 Testing the Multithreading Model	39
3.4 Conclusions	42
<b>4 Multithreading Without Caching</b>	46
4.1 New Format	47
4.2 Switch-On-Load	47
4.3 Increasing the Run-Lengths: Explicit-Switch	48
4.3.1 Grouping Within Basic Blocks	50
4.3.2 Grouping Beyond Basic Blocks	50
4.3.3 Estimation Experiment	58
4.4 Conclusions	59
<b>5 Multithreading With Caching</b>	62
5.1 Caching	63
5.2 Run-Lengths with Caching	63
5.2.1 Smarter Scheduling	66
5.3 Switch-On-Miss	69
5.4 Conditional-Switch	72
5.5 Conclusions	74
<b>6 Limited Bandwidth</b>	77
6.1 Bandwidth Requirement	78
6.1.1 Bandwidth Requirement Without Caching	79
6.1.2 Bandwidth Requirement With Caching	79
6.2 Squeezing Through a Limited Bandwidth Network	83
6.2.1 Remote Memory Bandwidth	85
6.2.2 Hot Spot Memory Modules	87
6.2.3 Location Hot Spots	91
6.3 Summary and Implications	95
<b>7 Miscellaneous Studies</b>	97
7.1 Synchronization	99
7.1.1 Spin Waiting	99
7.1.2 Non-Spinning Synchronization	99
7.2 Line Size for Minimizing Bandwidth	102
7.3 Cache Degradation due to Multithreading	109
7.4 Longer Latencies	113
	115

<b>8</b>	<b>Hardware Support</b>	<b>120</b>
8.1	Hardware for Explicit-Switch	121
8.1.1	Pipelined Context Switch	121
8.1.2	Result Matching	126
8.1.3	Scheduling	126
8.1.4	Multiple Register Sets	127
8.1.5	A Denser Register File	128
8.2	Hardware for Switch-On-Miss	131
8.2.1	Cache Coherency	131
8.2.2	Result Matching and the Address Table	133
8.2.3	Stalling On Writes	134
8.2.4	Context Switch Delay	135
8.3	Conclusions and Extension to Multiprogramming	136
<b>9</b>	<b>Conclusions and Future Directions</b>	<b>138</b>
9.1	Conclusions	138
9.2	Future Directions	140
	<b>Bibliography</b>	<b>141</b>
<b>A</b>	<b>Distribution Function Histograms</b>	<b>150</b>
A.1	Area Proportional to Value	150
A.2	Aggregation of Adjoining Piles	151
A.3	Mismatch with Old Intuition	152
<b>B</b>	<b>Simulator</b>	<b>153</b>
B.1	Introduction	153
B.1.1	Overview	154
B.2	Previous Simulators and Tradeoffs	154
B.2.1	Cycle-by-Cycle Simulators	155
B.2.2	Execution Driven Simulators	155
B.2.3	Tradeoffs	156
B.3	Simulator	157
B.4	Code Augmentation	160
B.4.1	An Example	161
B.4.2	Virtual Registers	164
B.5	Performance	164
B.5.1	Cost of In-line Context Switching	164
B.5.2	Slowdowns Factors for Basic Simulations	166
B.5.3	Multithreading and Caching	167
B.6	Summary and Future Research	168

# List of Figures

1.1	Round trip network latency as a function of machine size. . . . .	3
1.2	Mechanisms for reducing the impact of memory latency. . . . .	5
1.3	Evolution of multithreading models . . . . .	6
1.4	Bisection bandwidth as a function of machine size. . . . .	10
2.1	Model of large shared memory multiprocessor. . . . .	15
2.2	Some popular network topologies. . . . .	17
2.3	Example of a shared memory program. . . . .	20
2.4	Sieve: a parallel primes finder. . . . .	22
2.5	Blkmat: blocked matrix multiply. . . . .	23
2.6	Sor: successive over relaxation. . . . .	24
2.7	Efficiency on ideal (0 latency) machine. . . . .	31
2.8	Revised model of parallel machine. . . . .	33
3.1	Model of a single thread. . . . .	35
3.2	Multithreading with 3 threads per processor. . . . .	35
3.3	Processor utilization as a function of multithreading. . . . .	36
3.4	Histograms of distribution functions. . . . .	38
3.5	Histograms of the run-lengths distributions for <b>switch-on-load</b> . . . . .	40
3.6	Histograms of the run-lengths distributions for <b>switch-on-load</b> . . . . .	41
3.7	Predicted and observed performance for <b>switch-on-load</b> . . . . .	44
3.8	Predicted and observed performance for <b>switch-on-load</b> . . . . .	45
4.1	New format for presenting multithreading efficiency results. . . . .	48
4.2	Multithreading levels and efficiencies for <b>switch-on-load</b> . . . . .	49
4.3	Inner loop of <b>sor</b> . . . . .	51
4.4	Histograms of the run-lengths distributions for <b>explicit-switch</b> . . . . .	53
4.5	Histograms of the run-lengths distributions for <b>explicit-switch</b> . . . . .	54
4.6	Multithreading levels and efficiencies for <b>explicit-switch</b> . . . . .	56
4.7	Example code fragments with potential for inter-block grouping. . . . .	58
4.8	Multithreading levels and efficiencies with inter-block grouping. . . . .	61
5.1	Histograms of the run-lengths distributions for <b>switch-on-miss</b> . . . . .	67
5.2	Histograms of the run-lengths distributions for <b>switch-on-miss</b> . . . . .	68

5.3	Multithreading levels and efficiencies for <b>switch-on-miss</b> . . . . .	73
5.4	Multithreading levels and efficiencies for <b>conditional-switch</b> . . . . .	76
6.1	Message sizes for remote references to shared memory. . . . .	80
6.2	A 2-D mesh network and its bisection. . . . .	81
6.3	Messages used to support coherent caching. . . . .	83
6.4	Squeeze performance model (informal). . . . .	87
6.5	Squeeze performance model (formal). . . . .	88
6.6	Snippets of remote memory bandwidth profiles. . . . .	90
6.7	Sorted profiles of the applications' remote memory bandwidth usage. . . . .	91
6.8	Sorted profiles of the applications' hot memory module bandwidth usage. . . . .	93
6.9	Sorted profiles of the applications' hot location bandwidth usage. . . . .	96
7.1	Messages for synchronization operations. . . . .	104
7.2	Operation of the waiting queue for a lock. . . . .	107
7.3	Bandwidth as a function of line size. . . . .	110
7.4	Miss rates as a function of line size. . . . .	111
7.5	Processor utilization as a function of line size. . . . .	112
7.6	Bandwidth as a function of line size. . . . .	113
7.7	Cache miss rates as a function of multithreading. . . . .	114
7.8	Assignment of threads to processors. . . . .	115
7.9	Efficiencies with longer latency. . . . .	116
7.10	Efficiencies with longer latency. . . . .	117
7.11	Load imbalance between threads with longer latencies. . . . .	118
8.1	Datapath for RISC processor. . . . .	122
8.2	Datapath with changes for explicit-switch multithreading. . . . .	123
8.3	Pipelined context switch for explicit-switch. . . . .	124
8.4	One bit of register file supporting 12 threads per processor. . . . .	129
8.5	Operation of the denser register file design. . . . .	130
8.6	Datapath with changes for switch-on-miss multithreading. . . . .	132
8.7	Pipelined context switch for switch-on-miss. . . . .	135
A.1	Example histogram showing piles of various sizes. . . . .	150
A.2	Example histogram showing aggregation of adjoining piles. . . . .	151
A.3	Example histogram showing a uniform distribution. . . . .	152
B.1	Diagram of using FAST. . . . .	158
B.2	Example of code augmentation . . . . .	162
B.3	Simulation slowdown . . . . .	166
B.4	Simulation slowdowns under different configurations. . . . .	167

# List of Tables

1.1	Frequency of accesses to shared memory. . . . .	4
2.1	Parallel Applications . . . . .	21
2.2	Summary of Application Characteristics. . . . .	28
2.3	Experiment: efficiency on an ideal machine . . . . .	30
2.4	Limits on the number of threads used. . . . .	32
3.1	Experiment: run-lengths under <b>switch-on-load</b> . . . . .	39
3.2	Experiment: <b>switch-on-load</b> . . . . .	42
4.1	Grouping and mean run-lengths achieved after reorganization. . . . .	52
4.2	Experiment: run-lengths for <b>explicit-switch</b> . . . . .	52
4.3	Experiment: <b>explicit-switch</b> . . . . .	55
4.4	Experiment: <b>explicit-switch</b> with inter-block grouping . . . . .	59
4.5	Grouping estimates if the compiler could do inter-block grouping. . . . .	60
5.1	Experiment: caching . . . . .	64
5.2	Average miss rates and execution efficiencies with caching. . . . .	65
5.3	Experiment: run-lengths under <b>switch-on-miss</b> . . . . .	66
5.4	Execution efficiencies under various scheduling policies. . . . .	71
5.5	Experiment: scheduling under <b>switch-on-miss</b> . . . . .	71
5.6	Experiment: <b>switch-on-miss</b> . . . . .	72
5.7	Experiment: <b>conditional-switch</b> . . . . .	75
6.1	Experiment: remote memory bandwidth needs of <b>explicit-switch</b> . . . . .	79
6.2	Average remote memory bandwidth needs under <b>explicit-switch</b> . . . . .	81
6.3	Bisection bandwidths of real machines. . . . .	81
6.4	Experiment: bandwidth under <b>switch-on-miss</b> . . . . .	84
6.5	Average remote memory bandwidth needs under <b>switch-on-miss</b> . . . . .	84
6.6	Experiment: bursty traffic under <b>switch-on-miss</b> . . . . .	86
6.7	Increased problem sizes of applications. . . . .	86
6.8	Slowdown factors under various bandwidth limits. . . . .	92
6.9	Slowdowns factors based on hot spot memory module bandwidth. . . . .	93
6.10	Over design factors for the network and memory modules. . . . .	94



6.11	Example networks which satisfy bandwidth requirements. . . . .	98
7.1	Frequency of use of various synchronization operations. . . . .	105
7.2	Experiment: Bandwidth versus Line Size . . . . .	110
8.1	Explicit-Switch: Performance loss with 9 cycle context switch. . . . .	125
8.2	Switch-On-Miss: Performance loss with 8 cycle context switch. . . . .	136
B.1	Context Switch Costs . . . . .	165

# Chapter 1

## Introduction

Shared memory multiprocessors are considered among the easiest parallel computers to program [Boo89, HLRW92, Ken92, LM92, LLG<sup>+</sup>92, NL91, RT86, SHG92, TKB92]. Programming is easier because the shared memory programming model allows the programmer to ignore issues such as the explicit location of data and its movement between processors. This model, however, is just an abstraction, and its success depends on the ability of the computer hardware and software to efficiently support it. This is analogous to the abstraction of a large virtual memory.

For small machines, with from 4 to 30 processors, this shared memory abstraction has been relatively easy to provide. It involves snooping caches on a single memory bus connecting all of the processors. This configuration has been named a *multi*[Bel85] and has been widely adopted for building small multiprocessors for which a single bus is able to provide sufficient bandwidth. Examples include the Sequent Symmetry[LT88], Encore Multimax[Enc87], and Silicon Graphics 4D-MP[BJS88].

However, building large shared memory machines has proven to be much more difficult than building other types of large parallel machines. For example, there exist 1,024 processor (message passing) Ncube's, 1,024 processor (message passing) CM-5's, 16,384 processor (SIMD) MasPar's, and 65,536 processor (SIMD) CM-2's[DM93]. Large commercial shared memory multiprocessors such as the KSR1[Ken92] or the Cray-T3D[KS93] have only recently been introduced and have not yet been built in very large configurations. The goal of this dissertation is to understand and address the key difficulties impeding the design and development of large shared memory multiprocessors.

## 1.1 The Latency Problem

By large shared memory machines, we mean hundreds or thousands of processors. For machines of this size, the bandwidth of a single bus is inadequate, and thus more complex networks, such as butterfly or grid networks, are required. The latency problem arises when a processor accesses a shared memory variable that is located in a memory module across the network. To perform this remote memory access, the processor issues a request message into the network. This message then traverses the network to the memory module. The memory module reads the value. And then it sends a result message back to the requesting processor. The interval between the sending of the request message until the return of the result message is called the *remote memory access latency*, or just *the latency*.

The networks of large machines are multi-hop networks, and messages are subject to switching, transmission, and congestion delays at each stage of the network. In a butterfly network, for example, a message traverses  $O(\log p)$  nodes to reach its destination, and in a two dimensional grid network a message traverses  $O(\sqrt{p})$  nodes. The aggregate latency through these networks can be hundreds of cycles. The latency becomes a problem if the processor spends a large fraction of its time sitting idle waiting for remote accesses to complete.

Figure 1.1 shows the extrapolated round trip network latencies (expressed in terms of the processor's cycle time) for several recent or proposed large parallel machines. These machines have a variety of architectures. The CM-5[LAD<sup>+</sup>92] is a message passing machine with a fat-tree[Lei85] network. DASH[LLJ<sup>+</sup>92] is a cache-coherent shared-memory multiprocessor with a two-dimensional toroidal mesh network. The KSR1[Ken92] is also a cache-coherent shared-memory multiprocessor but with a ring (or hierarchy of rings) network. And TERA[ACC<sup>+</sup>90] (expected in 1994) is a shared-memory multiprocessor without caching, and it uses a three-dimensional toroidal mesh network.

The latencies shown in the graph have been extrapolated based on scaling these networks. For the CM-5 and DASH, the latencies do not include congestion affects, and thus the actual latencies in heavily loaded networks will be higher than these curves. The KSR1 has not yet been disclosed well enough to allow extrapolating a complete latency curve.

For machines supporting 1024 processors, these graphs suggest that we can expect latencies of 200 cycles or more, once congestion affects are taken into account. Furthermore,

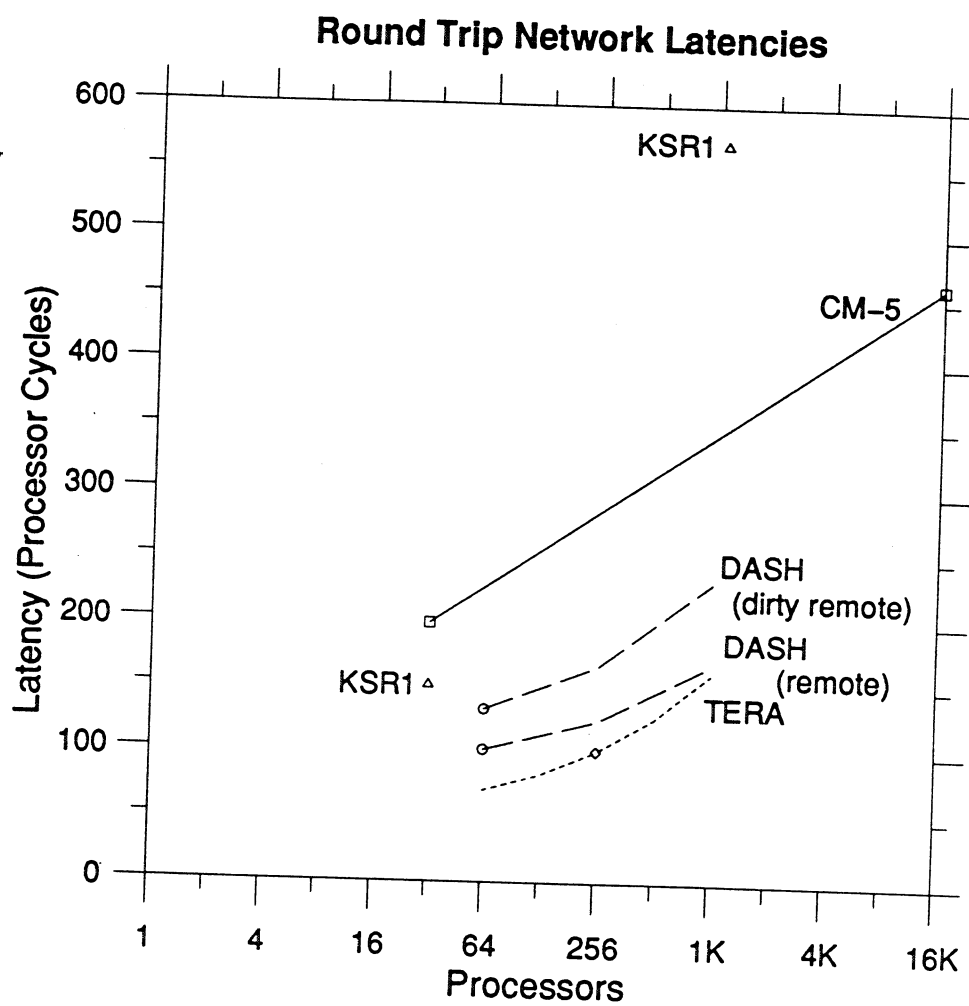


Figure 1.1: Round trip network latency as a function of machine size.

Application	cycles / remote access
sieve	11.0
blkmat	102.5
sor	4.4
ugray	17.5
water	40.2
locus	6.5
mp3d	5.7
barnes	41.3

Table 1.1: Frequency of accesses to shared memory.

we expect that the latencies will continue to increase over time as on-chip speeds are reduced more quickly than off-chip speeds.

Another way to look at these latencies is the opportunity cost in terms of the instructions that might be processed while waiting for a remote reference to return. The processors used in some of these machines execute more than one instruction per cycle<sup>1</sup>, and thus the opportunity cost is the product of the latency by the number of instructions executable per cycle. We expect that future processors will continue to further exploit superscalar and superpipelining techniques, and thus further increase the opportunity cost of remote accesses.

In contrast with these latencies, Table 1.1 shows the reference rates for a number of parallel shared-memory applications. These applications will be used throughout this thesis and will be described in Chapter 2. These remote reference rates were calculated by dividing the total number of execution cycles (excluding stalls) by the total number of shared memory loads and stores. If these applications were run on a machine that had to stall on each remote access for a 200 cycle latency, the machine would be busy as little as 3% of the time. Clearly some means of tolerating the latency is essential for building large shared-memory parallel computers.

## 1.2 Solutions to the Latency Problem

Figure 1.2 lists six mechanisms for addressing the latency problem. Caching[Smi82] works by filtering out redundant references. Multithreading[Kow85] allows switching to a

<sup>1</sup>The KSR1 processor is two-way superscalar, and the TERA processor has a wide instruction word and can issue three operations per cycle.

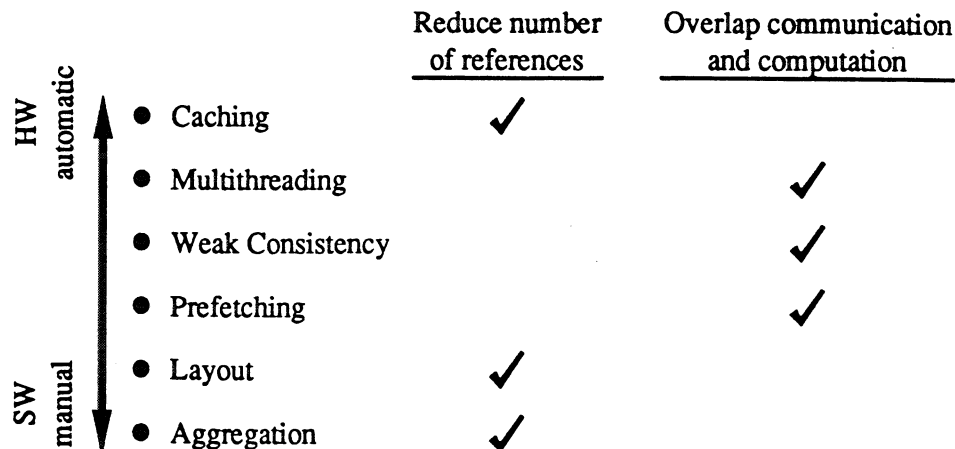


Figure 1.2: Mechanisms for reducing the impact of memory latency.

different thread while waiting for a remote reference to complete. Weak consistency[AH90] allows some overlapping of writes without concern that they might arrive out of order. Prefetching[LYL87] allows requesting data before it will be needed. By layout[Hig93] we mean the idea of arranging data on or near the processor that is going to use it. And aggregation[HLRW92] is the idea of getting large amounts of data at once.

The mechanisms near the top of the diagram are more commonly automatic (or invisible) as far as the programmer is concerned and are generally implemented in hardware. The mechanisms near the bottom of the diagram are often implemented in software either by a smart compiler or manually by the programmer. For example in a message passing program, the programmer explicitly specifies the layout of data and the packaging of messages(i.e., aggregation of data).

All of these mechanisms have their limitations. Caches must be kept coherent, which becomes complex for large machines[HLRW92, TD91]. Furthermore, the hit rates may be low for accesses to shared data[DRPS87, GHG<sup>+</sup>91]. Multithreading requires complex hardware to allow rapidly switching between the threads on a processor. And since it requires extra threads, it also requires extra parallelism and is thus limited to larger problems. Consistency models prohibit many compiler optimizations. Weak consistency allows more than sequential consistency, but it is a less intuitive programming model[GLL<sup>+</sup>90]. Prefetching is useful for applications with predictable behavior such as many scientific codes, but it is of limited applicability for more chaotic codes that use complex data structures. It

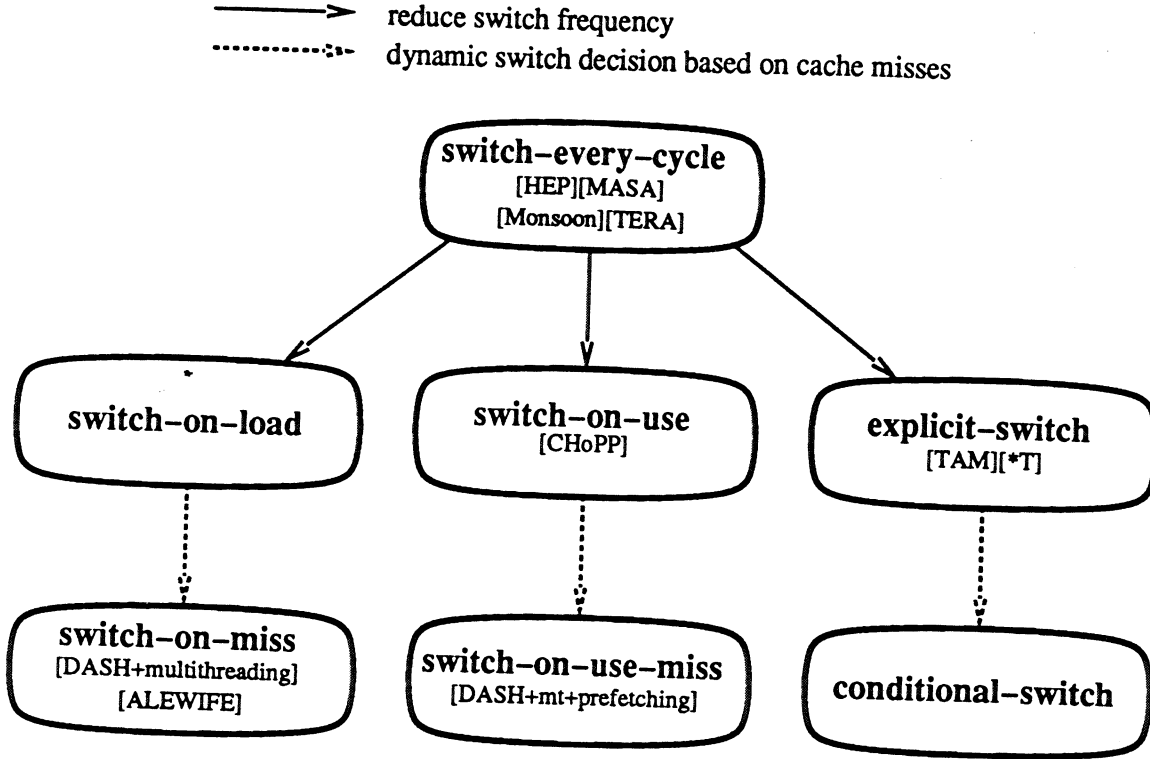


Figure 1.3: Evolution of multithreading models

also may waste bandwidth by prefetching data that is not used. Generation of good layouts is limited to the more regular and predictable applications. Finally, aggregation requires coarse grain parallelism where large data items can be manipulated.

In this dissertation we have chosen to focus on evaluating the mechanisms of multithreading and caching. A weakly consistent memory model is assumed throughout. These are the more automatic mechanisms and are most consistent with the shared memory programming model. The mechanisms of prefetching and layout can be of additional benefit, and we have incorporated them in a limited fashion in a few of our studies, however there remains room for further research in these areas.

### 1.3 Overview of Previous Multithreading Work

Previous multithreading research has been motivated by three concerns: tolerating memory latency, building a fast pipeline, and supporting a dynamic dataflow like execution model. Figure 1.3 shows the evolution of multithreading models and some of the motiva-

tions for moving from one model to another. Some of these models have not been studied previously but can be predicted based on the motivations.

The costs and concerns of multithreading are: the large number of threads needed, the scheduling mechanism used to schedule the many threads on a processor, the cycles lost to context switching overhead, and the large register file. These costs and concerns are influenced by when and how often context switching is performed.

### 1.3.1 Fast Pipeline

The oldest model **switch-every-cycle** was used in the Denelcor HEP [Kow85] and in MASA [HF88]. After each instruction, the processor switches to a different thread. This allows a fast CPU pipeline to be built because it eliminates data dependencies between instructions in the pipeline by interleaving different threads. It also allows memory latencies to be tolerated by not scheduling a thread until its reference has completed. Unfortunately this model requires a large number of threads and a large amount of hardware to support them. Also, by interleaving the instructions from many threads, a single thread is limited to a small fraction of the processing power. TERA[ACC<sup>+</sup>90] is similar to the HEP, but the scheduling policy has been changed so that a thread can issue more than one reference into the network before waiting for the results.

### 1.3.2 Hiding Memory Latency

The rest of the multithreading models that we consider execute a thread for many cycles before context switching. The optimizing compiler is responsible for the ordering of instructions so as to hide the small pipeline delays, and context switches are thus used only to hide the long memory latency of remote accesses.

The **switch-on-load** model switches on load instructions which access shared memory. Loads from local memory and other instructions all complete quickly and can be scheduled by the compiler. Shared memory stores do not wait for their completion and therefore do not cause context switches either. The advantages of this model over **switch-every-cycle** are that a single thread can execute at full speed until it context switches, and fewer total threads will be needed since multithreading is not being used to hide pipeline delays. Simpler hardware may also be possible since context switches are less frequent.

The **switch-on-load** model sometimes context switches sooner than it needs to.



If a compiler can order instructions so that a load is issued several cycles before the value is used, the context switch does not have to occur until the actual use of the value. This allows a thread to hide some of its own memory latency by prefetching data. This can be implemented by adding a *valid* bit to each register. The valid bit is cleared when the load instruction is issued and set when the result returns from the network. The **switch-on-use** multithreading model context switches on the use of an invalid register, rather than on a load instruction. The **switch-on-use** model was used in the design of the CHoPP architecture[MPS87] which had a VLIW multithreaded processor.

A benefit of the **switch-on-use** model is that several load instructions can be grouped together so that all of the loads in the group can be issued into the memory network before any of the results are used. This allows a single context switch (on the first of the uses) to wait for all of the loads in the group. For example, a simple computation may load two values from shared memory and then compute their average. Both loads should be issued into the memory network and then a single context switch performed upon the use of the first value. This allows waiting for both loads together rather than individually.

An alternative method for grouping shared loads together is to add an explicit context switch instruction between the group of loads and their subsequent uses. The **explicit-switch** model allows similar grouping to **switch-on-use**, but is simpler to implement and requires the addition of only a single instruction. We evaluate the **explicit-switch** model in Chapter 4 and find that it can eliminate from 50% to 80% of the context switches needed by the **switch-on-load** model.

The most recent data flow research[CSS<sup>+</sup>91, NPA92] has adopted the **explicit-switch** model. Short threads execute until their completion at which point they cause a context switch to a new thread.

### 1.3.3 Adding Caches

By adding caches to the previous models, the cache can satisfy many of the shared loads without going to shared memory. Only those shared loads that miss in the cache will have long latencies and cause context switches. By filtering out many of the remote references, caching makes the job of hiding the memory latency easier, and fewer threads will be needed to cover the latency. In Chapter 5 our results will show that for most applications just 2 or 3 threads per processor is sufficient.

The **switch-on-miss** model switches at points where load instructions miss in the cache. An early study of this by Weber & Gupta [WG89] suggested substantial performance benefits were available, but a later study as part of the DASH project [GHG<sup>+</sup>91] had less optimistic results. **Switch-on-miss** multithreading was also studied as part of the ALEWIFE project [ALKK90] and achieved good results for a few simple applications. One draw back of context-switching on cache misses is that the context switch is detected after a number of subsequent instructions have started down the CPU pipeline. These instructions must be canceled, and thus there will be a context switch cost of several cycles because of the wasted pipeline slots.

The **switch-on-use-miss** model context switches when a **use** instruction tries to use the value from a shared load that missed in the cache. It was studied (approximately) by the DASH project [GHG<sup>+</sup>91] when they looked at the combination of prefetching and multithreading. Their prefetch instructions act like the initial load instructions, and their subsequent load instructions act like the use of the data. They found little benefit from prefetching when combined with multithreading, however they state that their prefetching method was meant for a single threaded processor and should be done differently for a multithreaded processor.

The **conditional-switch** model adds caching to the **explicit-switch** model. The code appears the same as that for the explicit-switch model: there is a group of load instructions, followed by a context switch instruction, followed by the instructions that use the loaded data. The difference is that the context switch instruction is treated as a *conditional* switch instruction. If any of the loads preceding the switch instruction missed in the cache, a context switch is performed as expected. But if all of the preceding loads hit, the context switch instruction is ignored and the thread continues executing. This model provides the benefits of grouping and caching as in the **switch-on-use-miss** model, but it may be simpler to implement.

## 1.4 Limited Bandwidth

Besides having long latencies on remote accesses, the networks on large parallel machines are also likely to have limited bisection bandwidths<sup>2</sup>. Figure 1.4 shows the bi-

---

<sup>2</sup>Bisection bandwidth is defined as the minimum bandwidth capacity between the two halves of a bisected machines, considering all possible bisections.

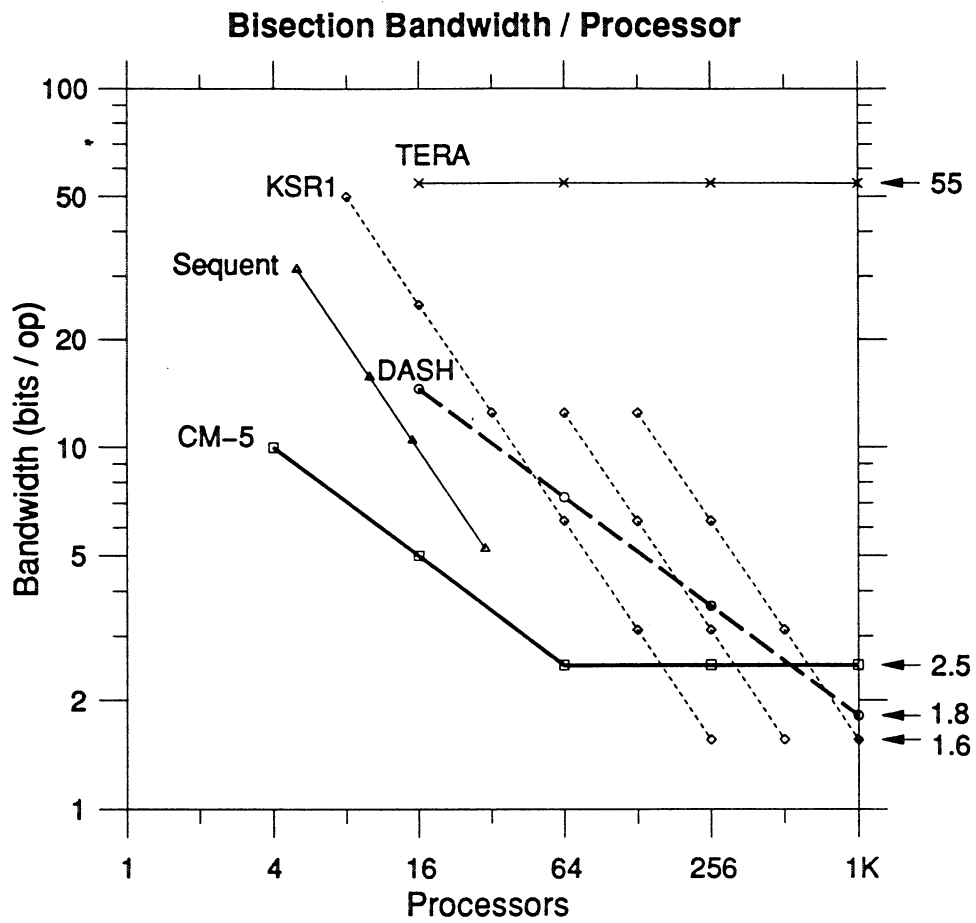


Figure 1.4: Bisection bandwidth of various parallel computers as a function of machine size. The bandwidth is expressed in terms of bits per processor operation at the peak capacity and peak execution rate.

section bandwidths of various parallel computers. These are peak bandwidths and are not expected to be fully achieved under real traffic patterns. Also, the bandwidths have been extrapolated to 1024 processors based on the proposed designs, even if such a large machine has not actually been built. The bandwidth values were calculated based on descriptions of the networks in [ML92], [Ken92], [LLJ<sup>+</sup>92], [LAD<sup>+</sup>92], and [ACC<sup>+</sup>90].

The key point of this graph is that for most networks, the bisection bandwidth drops as the number processors is increased, and that for a large (1024 processor) machine, only 1 or 2 bits of bandwidth per operation will be available. In fact, achievable bandwidth may be only half of that amount because of the congestion caused by irregular traffic patterns [Dal90].

The reason the bandwidth drops off as the number of processors increases is related to the scaling characteristics of the networks. For the Sequent [ML92], there is a single shared bus and thus the bandwidth per processor diminishes in proportion to the number of processors. For both bandwidth and electrical reasons, sharing a single bus limits the number of processors to around 30.

The KSR1 [Ken92] is similar, it uses a single high bandwidth ring for small machines, or a two level ring of rings for larger machines. The bisection bandwidth depends only on the top level ring, and when calculated on a per processor basis, decreases linearly. For large machines they stave off the bandwidth decline by providing multiple rings at the top level. The three lines in Figure 1.4 for the KSR1 represent the three configuration options for these top level rings. The largest option has 4 GB/sec of bandwidth along the ring (8 GB/sec crossing the bisection), but when divided among 1024 processors (two-way superscalar) running at 20 Mhz, this provides only 1.6 bits per operation.

The DASH [LLJ<sup>+</sup>92] architecture scales better because it is based on a 2-D wrap-around mesh (torus) rather than a ring. The bisection bandwidth per processor drop off as the square root of the number of processors. At 1024 processors, which is more than this design was meant for, the bisection bandwidth is 1.8 bits per operation.

The CM-5 network [LAD<sup>+</sup>92] is a fat tree. Fat trees [Lei85] are a family of networks where the connections between nodes at higher levels of the tree are generally "fatter" than the connections between nodes at lower levels of the tree. With the appropriate connection widths, a fat tree can provide constant bisection bandwidth per processor as the machine is scaled. However to save costs, the designers chose to eliminate many of the channels at the higher levels in the tree. For 1024 processors, the bisection bandwidth is 2.5 bits per

operation. This figure is without the optional vector units. If they are included and achieve their potential factor of 4 performance increase, the bandwidth when expressed in bits per operation will be reduced to only 0.6 bits.

Finally, and in contrast to the other networks, the proposed Tera network provides a bisection bandwidth of 55 bits per operation and scales the bandwidth linearly with the number of processors. The network is a sparsely populated 3-D wrap-around mesh[ACC<sup>+</sup>90], and to scale the bandwidth linearly, they increase the number of network nodes faster than the number of processors. For large machines Tera has more than an order of magnitude greater bandwidth than other machines. We suspect however, that providing this large network bandwidth may not prove cost effective.

In this thesis we do not focus on any particular network topology. Instead we measure the bandwidth needs of our benchmark applications, and then use the results to reason about the types of machines that should be built and the bandwidth capacity that they should supply.

## 1.5 Overview of Thesis

In this dissertation we concentrate on the **switch-on-load**, **explicit-switch**, **switch-on-miss**, and **conditional-switch** models. If caches are not used, our results will show that grouping is important and thus **explicit-switch** is preferable to **switch-on-load**. However if caches *are* used, our results will show that grouping has little benefit and thus **switch-on-miss** is preferable to **conditional-switch**.

The remainder of this dissertation is organized as follows: Chapter 2 discusses our simulation methodology and our set of benchmark applications. Chapter 3 presents a performance model for a multithreaded processor. Chapter 4 focuses on hiding latency with multithreading and evaluates the **switch-on-load** and **explicit-switch** multithreading models. Chapter 5 adds coherent caching to the system and evaluates the **switch-on-miss** and **conditional-switch** multithreading models. Chapter 6 considers the problem of limited network bandwidth and presents results on the amount of bandwidth that is needed by the various applications and multithreading models. Chapter 7 presents miscellaneous studies and experiments on synchronization and various caching issues. Chapter 8 discusses the hardware mechanisms needed to support a multithreaded processor. And Chapter 9 presents conclusions and directions for future research.

There are also two appendices: Appendix A explains a new method for plotting distributions that we have introduced in order to visually present both clearly and compactly the types of distributions that we have encountered. And Appendix B explains the techniques used to build the simulator that made this research possible.

## Chapter 2

# Methodology

This chapter discusses our simulation based research methodology. First we present our model of a large shared memory parallel machine. Then we present the benchmark applications that we have used to evaluate this model. And lastly we present our simulation system and its limitations.

We have chosen simulation over analytic modeling because parallel programs are complex interacting systems that are difficult to model accurately. Saavedra-Barrera *et al.*[SBCvE90, SBC91] and Agarwal[Aga92] have done analytic models of simplified multi-threading systems, but for reasons of tractability they are forced to assume that threads are independent and have accesses at exponentially distributed intervals (i.e. according to a Poisson process). The threads of parallel programs, however, are not independent because they have substantial sharing of data, and they use synchronization to coordinate this sharing. We will also see in Section 3.2 that many programs have distinctly non-exponential inter-arrival distributions. Real programs also have many parameters (such as reference rates, amount of sharing, synchronization patterns, imperfect load balancing, parallelization overhead, and time varying behavior) that are difficult to characterize and are not yet well understood.

### 2.1 Machine Model

Figure 2.1 shows our model of a large shared memory multiprocessor. It consists of a number of processors and a like number of shared memory modules that are all interconnected by a switching network. Each processor also has a local memory which holds

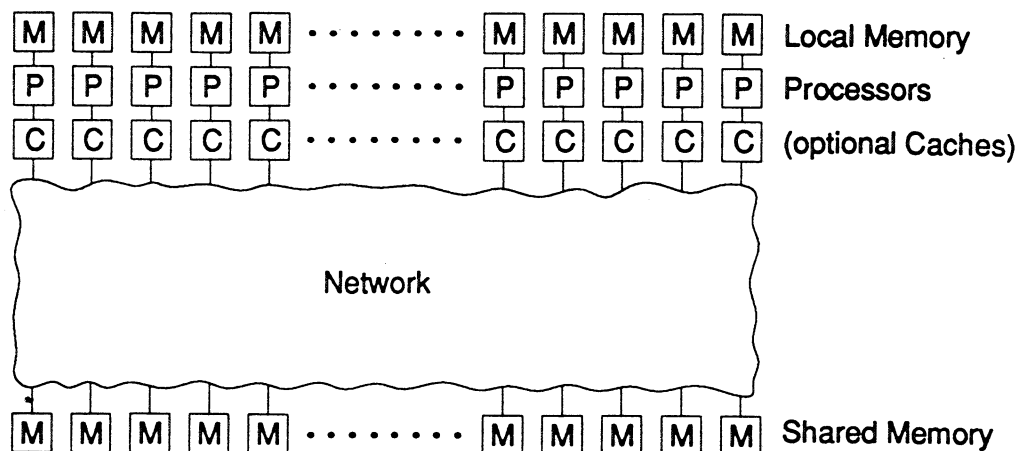


Figure 2.1: Model of large shared memory multiprocessor.

variables local to threads, stacks, and the code. We assume all accesses to local memory are instantaneous. This is reasonable since local data and instructions can be easily cached, and any misses can be serviced locally. Accesses to shared memory are sent across the network and thus have a long latency before they return.

We look at two variations of this model: one with caching of shared data (as shown in the figure), and the other without it. Both types of systems have been and are being built. Examples of systems without coherent caching are the HEP[Kow85], BBN Butterfly[BBN89], and Tera[ACC<sup>+</sup>90]. Systems with coherent caching include all of the shared bus based systems such as the Sequent[Ost89] as well as the more scalable KSR1[Ken92] and research projects such as DASH[LLG<sup>+</sup>90, LLG<sup>+</sup>92, LLJ<sup>+</sup>92] and ALEWIFE[ALKK90].

For a real machine, the model in Figure 2.1 will likely be folded back upon itself so that each processor is directly connected to one of the shared memory modules. This would give each processor direct access to a small portion of shared memory. If the programmer (or compiler) can control the layout of data onto the memory modules, she might be able to arrange the data on or near the processors where it will be used. Such layouts could eliminate a large fraction of the remote references, but they are not possible for many applications[SHG92].

In this research (with our applications, programming languages, and compiler technology) we do not have the capability of customizing the data layout for each application. We therefore assume that data is randomly interleaved across the memory modules. This



interleaving is done at the level of the largest memory access unit. For systems without caches, the largest memory access unit is a double word; for systems with caches, the memory access unit is a cache line.

Without optimization of the data layout, the performance advantage of the folded machine configuration is small. On a 1000 processor machine, for example, only 1/1000th of the accesses will be to the locally accessible memory module. This small factor is insignificant, and thus we study the model as shown in Figure 2.1 (with no locally accessible shared memory.)

Mellor-Crummey and Scott[MCS91] argue against building such “dance hall”<sup>1</sup> machines because their efficient synchronization techniques depend upon having either coherent caching or local access to part of shared memory. In Section 7.1 we propose preferable synchronization techniques that will eliminate this taboo on “dance hall” machines.

### 2.1.1 Network

There are many proposed network topologies<sup>2</sup>. Figure 2.2 shows some that are popular for reasons involving: latency, bandwidth, cost, modularity, and availability of simple routing algorithms. Network design is still an active research area with many competing concerns. In this research we do not select any specific network, but instead we focus on the general characteristics of all of these networks. First, they all are packet switched networks that allow many parallel references to be traveling through the network simultaneously. Second, if the processors can support it, each processor may have several outstanding references in the network at once. And third, references will have long latencies because they are routed through many network nodes and experience congestion and delays along the way.

In this research we are interested in machines which range in size from a hundred processors up to a few thousand processors. We will use the design point of a 1000 processor machine for choosing parameters and further reasoning. Figure 1.1 in Chapter 1 showed the latencies of several existing and proposed large interconnection networks. For a 1000 processor machine, these networks all have round trip latencies in the range of a few hundreds processor cycles. We choose a latency of 200 cycles as representative of these figures

---

<sup>1</sup>The term “dance hall” comes from an analogy between a machine with the processors and memories separated at different ends of the network and a dance hall with separated boys and girls.

<sup>2</sup>See for example: Almasi/Gottlieb[AG89] chapter 8.

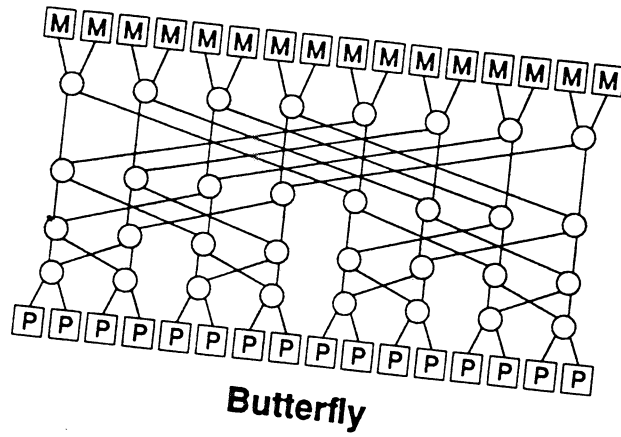
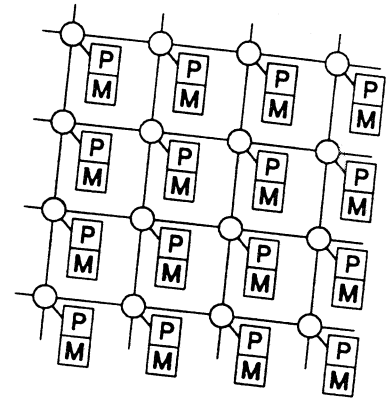
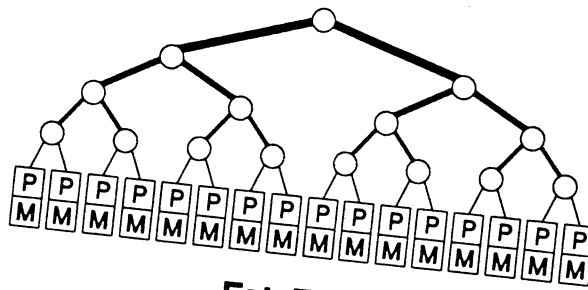
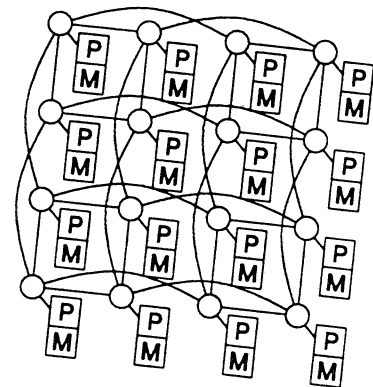
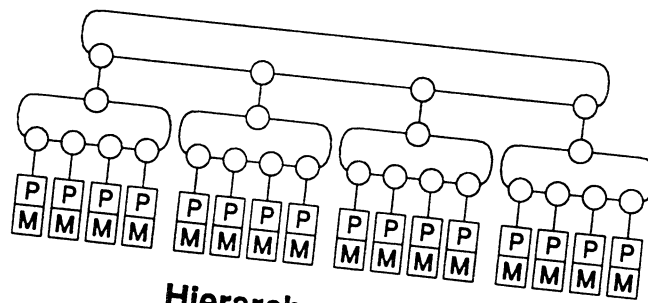
**Butterfly****Mesh****Fat-Tree****Hypercube****Hierarchy of Rings**

Figure 2.2: Some popular network topologies.

and focus this research on tolerating latencies of this magnitude.

We model the network and shared memories simply as a black box that takes 200 cycles to respond to a remote memory reference. In a real network, much of this latency will come from delays due to minor random congestion, and these small delays are an expected component of the 200 cycle latency. A more difficult case is severe congestion caused, for instance, by a program induced hot spot in which every processor tries to simultaneously access the same memory location. In this case delays can become much longer than the delay of 200 cycles that we have assumed. We ignore such congestion initially, but in Section 6.2 we assess the frequency of such hot spots and their impact on our simulation results.

### 2.1.2 Processor

We expect that the processors used in parallel systems will be the same or very similar to the microprocessors used in high performance workstations. This is because the peak performance of a parallel system is the product of the performance of a single processor and the number of processors. Such a large development effort is put into the race for the highest performance microprocessor that these push the technology curve and offer the most cost effective single processor.

To tolerate latency, however, we evaluate multithreading techniques which require that the processor be able to context switch rapidly between threads. In the past, multithreaded processors, such as the HEP[Kow85], have involved very different and complex processor designs that context switch every cycle and use a large number of threads. Instead we look at multithreaded processors that are similar to today's RISC microprocessors with the addition of being able to context switch on long latency remote memory accesses. Chapter 8 discusses the hardware issues in detail. Here we wish simply to specify our assumptions about the multithreaded processors that we simulate, and leave their justification to Chapter 8.

We assume the same instruction set and instruction timings as the MIPS R3000[Kan89], but with a few modifications. Most importantly, we assume that the register file has been replicated on the chip enough times so that each thread running on the chip can have its own set of registers. Because the registers are on chip and do not have to be saved or loaded from memory on a context switch, the processor should be able to switch quickly between threads; in some cases as fast as a single cycle (see Section 8.1.1).

Another modification is that we have added double word loads and stores to the instruction set. Many floating point numbers are stored as double words, and it is crucial (when the network has long latencies) to get the whole thing at once rather than having two separate references as is done on the MIPS R3000. More recent machines, such as the MIPS R4000, all provide double word loads and stores.

Finally, we provide both local and shared versions of all memory access instructions. This is based on the assumption that memory references can be classified by the compiler as either local or shared. For instance references to locations in a shared array would use **shared-load** instructions while references to local variables would use **local-load** instructions. This compiler classification may not be possible in the case of pointers if it is unclear what is pointed to and whether or not it resides in shared or local memory. We call these unclear cases *ambiguous* pointers, and they must be resolved at run-time either with extra code or special hardware, which will likely slow down and/or complicate the machine. Ideally we would like the compiler to classify as many references as possible because this information will be needed for compiler optimizations in Chapter 4.

### 2.1.3 Programming Language

Our applications are written in the augmented C dialect that is used in writing shared memory programs on the Sequent[Ost89]. Figure 2.3 shows an example of a simple program that multiplies two matrices. The arrays **d**, **e**, and **f** are declared as residing in shared memory by the addition of the type modifier “**shared**” before their declaration.

Unfortunately this language does not have shared memory declarations for objects accessed indirectly via pointers. The compiler does not know that the parameters **a**, **b**, and **c** to the **worker** function will be arrays in shared memory. For this simple program the compiler might deduce this information through global analysis, but in the general case this is difficult.

In our simulations we at first used dynamic testing of pointers to determine if addresses were in local or shared memory. Later we observed that for our application programs, true ambiguous cases (where sometimes a pointer points to a local location but at other times points to a shared location) never occurred. We thus collected classification information from an initial run of the application and fed it back into subsequent compilations, as is done in trace analysis. This allowed complete compiler classification of all

```

/*-----
   simple example program that multiplies two matrices
   -----*/
shared double d[10][10], e[10][10], f[10][10];

main()
{
    /* will compute f = d * e */
    ... initialize: d and e ...
    m_set_procs(100);           /* set to 100 threads */
    m_fork(worker, d, e, f);    /* fork the threads */
    ... print result: f ...
}

worker(a, b, c)                /* will compute c = a * b */
{
    double a[10][10], b[10][10], c[10][10];
    int i, j, k, myid;
    double sum;

    myid = m_get_myid();
    i = myid/10;                /* get unique thread id [0,100) */
    j = myid%10;               /* calculate thread's row */
                                /* calculate thread's col */

    sum = 0.0;
    for (k = 0; k < 10; k++)
        sum += a[i][k] * b[k][j];
    c[i][j] = sum;             /* calculate sum of products */
                                /* each thread calculates */
                                /* one element of c */
}

```

Figure 2.3: Example of a shared memory program.

Application	Lines	Cycles	Description & Problem size
sieve	236	106 M	counts primes number of primes < 4,000,000
blkmat	369	87 M	blocked matrix multiply 200 × 200 matrices
sor	333	258 M	successive over relaxation 192 × 192 grid
ugray	10784	1353 M	ray tracing graphics renderer gears (7169 faces), 20 × 512 slice of image
water	1368	1082 M	simulate a system of water molecules 343 molecules, 2 iterations
locus	6347	665 M	route wires in a standard cell circuit Primary2 (1290 cells × 20 channels)
mp3d	1510	192 M	simulate rarefied hypersonic flow 100,000 particles, 10 iterations
barnes	2109	1148 M	gravitational N-body simulation 4096 bodies in two clusters

Table 2.1: Parallel Applications

references.

Although we were able to obtain complete classification information through trace analysis, we would like to advocate that this shared versus local distinction is important and that it should be supported explicitly by future shared memory parallel languages. This could be done by allowing declarations of parameters (such as *a*, *b*, and *c* in the example) and pointers as *pointing to shared memory*.

## 2.2 Benchmark Applications

Table 2.1 shows the eight benchmark applications used in this research. These are all scientific programs that perform some computation or numeric simulation. The first three (*sieve*, *blkmat*, and *sor*) are toy applications written as part of this research. The other five are real applications. *Ugray*[Boo89] was parallelized by myself, and has been used in a few parallelism studies[BR92, LS91, O’K92]. The last four (*water*, *mp3d*, *locus*, and *barnes*) are part of the Stanford SPLASH benchmark set[SWG92] and have been used in many studies, especially those associated with the DASH project[LLJ<sup>+</sup>92].

Each of the applications has some unique behavioral characteristic(s), and the three toy applications were chosen because they each have distinct behaviors that broaden

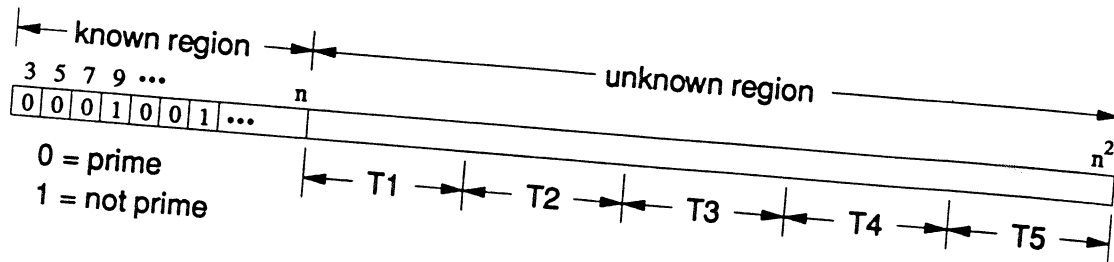


Figure 2.4: Sieve: a parallel primes finder.

the scope of the total benchmark set. The following sections contain brief descriptions of each application and report their main characteristics. Their original descriptions and codes often mix the terminology of *processes* and *processors*. Here we are more careful and will use the term *thread* to mean a *process*, and *processor* to mean a physical processor.

### 2.2.1 Sieve

The **sieve** application finds and counts the number of primes that are less than some given number. Figure 2.4 shows how this algorithm was partitioned for parallel execution. It represents the number space by a bit vector in shared memory with one bit for each odd number (the number 2 is treated separately). Initially all the bits are 0, which means that the numbers might be prime. As the sieve executes, whenever a number is determined to not be prime, its bit is set to 1.

Initially there is a small region of known primes. If this region goes up to  $n$ , then it is adequate for computing all primes up to  $n^2$ . The region from  $n$  to  $n^2$  is called the unknown region, and it is partitioned across the threads. Each thread uses the primes in the known region to perform the sieve in its portion of the unknown region. A barrier synchronization is then done, and the known region is expanded to  $n^2$  and the unknown region becomes  $n^2$  to  $n^4$ .

The main characteristics of **sieve** are:

- Regular intervals between shared memory accesses.
- No sharing of data in the unknown region.
- Very infrequent synchronization.

### 2.2.2 Blkmat

The **blkmat** application multiplies matrices. Figure 2.5(a) shows the partitioning

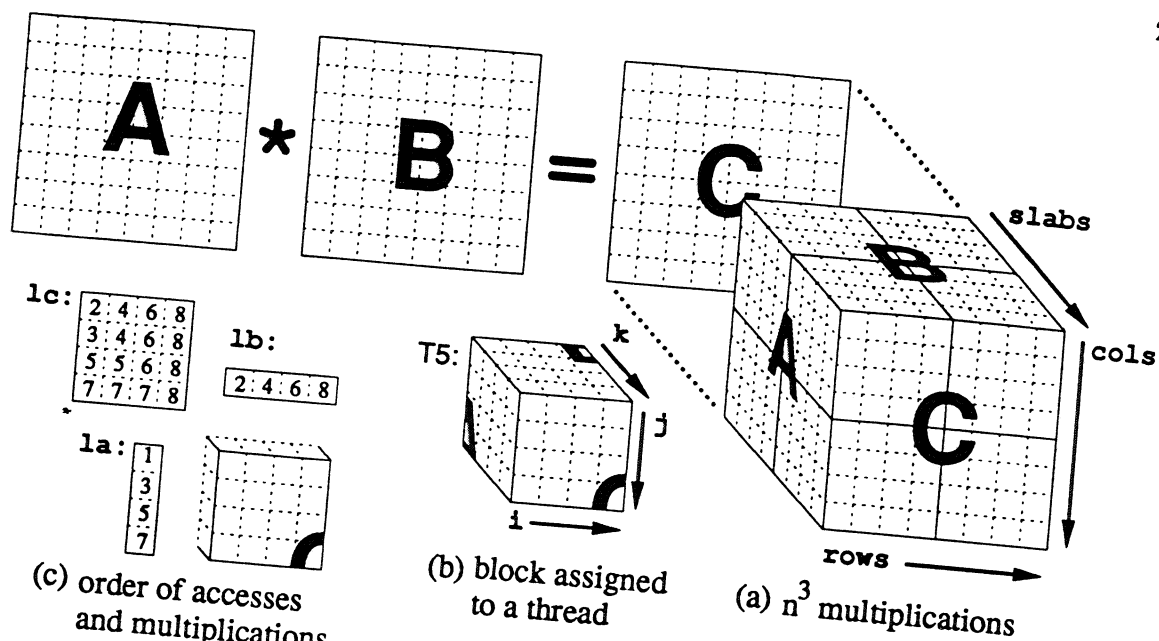


Figure 2.5: Blkmat: blocked matrix multiply.

of the computation for parallel execution. Multiplying two  $n \times n$  matrices uses a total of  $n^3$  multiplications in the standard algorithm, which we have used. This set of multiplications is then partitioned along all three dimensions so as to minimize the surface to volume ratio of the partitions. Surface area represents the amount of inherent communication that must be done, because it is the portion of the arrays  $A$ ,  $B$ , and  $C$  that must be accessed by a thread. The volume represents the amount of computation that is performed. Partitioning along all three dimensions in cuboidal blocks minimizes the ratio of communication to computation.

Figure 2.5(b) shows the  $4 \times 4$  block of the computation assigned to thread 5. To do its computations, thread 5 will read values from a  $4 \times 4$  region of matrix  $A$  and a  $4 \times 4$  region of matrix  $B$ . To achieve the minimum communication level, the program copies the values from  $A$  and  $B$  into local arrays 1a and 1b and reuses these local copies rather than repeatedly retrieving the same values from shared memory. The order of reading values into 1a and 1b and the order of computation of 1c is shown by the numbers in Figure 2.5(c), for one slice of the block. This unusual ordering was used so as to interleave the computation and communication as much as possible.

At the end of the computation, locking is used to coordinate the accumulation of partial sums into the matrix  $C$ . Since this involves little computation, the access rates



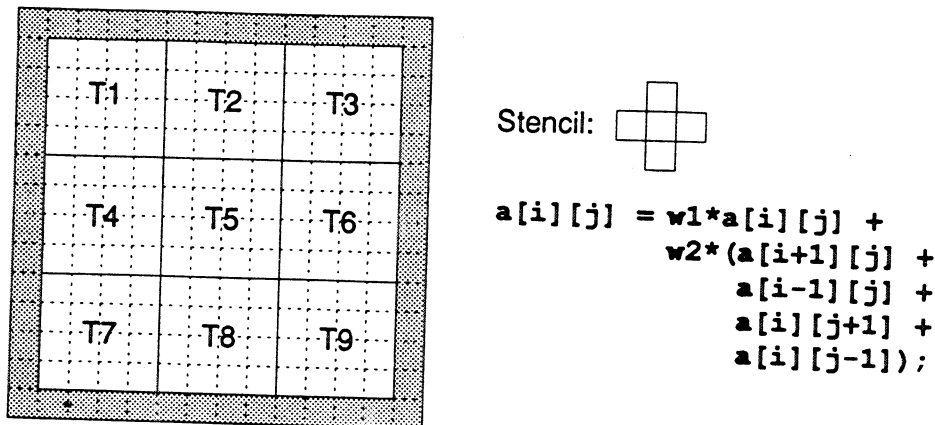


Figure 2.6: Sor: successive over relaxation.

during this phase are much higher than in the main calculation phase of the algorithm.

The main characteristics of `blkmat` are:

- Low access rates to shared memory.
- Varying intervals between accesses.
- A separate termination phase with much higher access rates.

### 2.2.3 Sor

The `sor` application is an iterative solver of Laplace's equation using the method of successive over relaxation. We use it for computing the heat flow in a square metal plate. The plate is represented by a grid of cells and is partitioned into regions as shown in Figure 2.6. Interactions between threads occur along the edges between regions, and thus the cells are partitioned into squarish regions in order to minimize the lengths of their edges. The outside edges of the grid contain the fixed boundary conditions and are not part of any thread's partition.

The computation proceeds by taking a cell in the grid and replacing it with a new value computed as a weighted sum of the old value and the four manhattan neighboring cells. The weights are chosen so as to make the computation converge as quickly as possible. After every few iterations convergence is checked for by comparing the new values to saved copies of previous values.

In order to avoid mixing results from the current and previous iterations, the grid is split like a checkerboard into red and black cells. First all of the red cells are calculated

and updated, and then this is followed by a barrier synchronization. After the barrier, all of the black cells are calculated and updated, and then there is another barrier. Recalculation of the red and black cells alternates in this fashion.

The main characteristics of **sor** are:

- High access rates to shared memory.
- Repeated barrier synchronization.
- Static partitioning and reuse of shared data.

#### 2.2.4 Ugray

The **ugray** application is a ray tracing graphics renderer. This is a computationally intensive rendering algorithm for producing high quality images. The sequential program is discussed in [Mar87], and its parallelization is discussed in [Boo89].

The data structures used to describe a scene are a complex web of cross connected structures. The top level is a large three dimensional array which is a coarse cellular map of space. It is used to quickly locate objects as light rays are traced through and bounce around space. Each cells has a linked list of objects that intersect it, and objects themselves are circular linked lists of vertices with additional links to attributes. The test scene of glass gears has 7169 faces and uses 7 Megabytes of data structures to describe it.

The main characteristics of **ugray** are:

- Complex linked data structures. (can not be prefetched)
- Moderate access rates. (complex calculations)
- Dynamic scheduling of jobs.
- Unpredictable reuse of data.

#### 2.2.5 Water

The **water** application simulates a system of water molecules in the liquid state. A brief description of this application and its parallelization appears in the SPLASH report[SWG92]. It is an N-body simulator that computes the pairwise interactions between molecules, except that it uses a spherical cut-off radius and thus ignores interactions with molecules beyond a certain distance.

The data set we used has 343 molecules (this was the largest data set available). These molecules are statically subdivided among the threads, and the same molecules are

kept from iteration to iteration regardless of whether or not they are close to each other (for load balancing reasons it was actually better if they were not close to each other). Because of the small (and odd) number of molecules, load balancing was usually imperfect. The most heavily loaded thread determines the rate of progress of the computation. Thus for our simulations we tried to choose the number of threads so that any imbalanced threads would have less work, rather than more work, compared to the bulk of the threads. Good values for the number of threads are: 343, 172, 115, 86, 69, 58, 49, etc. With 172 threads, for example, each thread is given 2 molecules except for the last thread which gets only 1. Thus the last thread will finish earlier than the others and have to wait. In contrast, if instead we had used 171 threads, each thread would get 2 molecules except for one of them which would get 3 molecules! This thread would be the slowpoke and all the others would have to wait on it. Nearly the entire machine would sit idle for a third of the time.

This application's writers were conscious of remote reference concerns and ordered the computations so as to quickly reuse data and thereby obtain good cache hit rates. There are also some large calculations that use only local variables and thus there are long periods during which no remote accesses occur.

The main characteristics of *water* are:

- Bursty traffic with long periods having no remote accesses.
- Good reuse of data.
- Imperfect static load balancing that is sensitive to the number of threads used.

### 2.2.6 Locus

The *locus* application is a router for standard cell VLSI circuits. A brief description of this application and its parallelization appears in the SPLASH report[SWG92].

The main task is to route wires through a grid of horizontal and vertical channels, where a good route is one that minimizes the number of wires passing through heavily used channels. The wire counts of the channels are maintained in a two dimensional cost array, and the bulk of the the shared memory accesses come from the evaluation of possible routes through this array. These route evaluations typically involve reading and comparing linear sequences of horizontal or vertical array elements.

Wires to be routed are put on several shared queues, and the threads dynamically schedule and route wires from these queues until the circuit is completely routed. After this

initial routing, wires are ripped up and re-routed to further optimize the result.

We used the largest input circuit available (Primary2.grin) which has 3817 wires and a  $1290 \times 20$  array of routing channels. This input shows good speedups up to around 64 threads, but performance gains diminish past this point. This application has the least parallelism of the applications we have used, but was included for reasons of application diversity.

The main characteristics of `locus` are:

- High access rates.
- Linear sequences of array accesses.
- Dynamic scheduling.
- Limited parallelism.

### 2.2.7 Mp3d

The `mp3d` application simulates rarefied fluid flow, such as that which occurs in the upper atmosphere. It uses Monte Carlo methods and simulates a representative collection of molecules. A brief description of the application and its parallelization appears in the SPLASH report[SWG92].

We simulate a system of 100,000 molecules. These molecules are statically assigned to threads, but no attempt is made to assign nearby molecules to the same thread. Because of this, the interactions of molecules are almost always with molecules assigned to other threads, and since the molecule are all moving, the collection of interactions is changing constantly. The net result is that there is little reuse of data.<sup>3</sup>

The main characteristics of `mp3d` are:

- High access rates.
- Little reuse of data.

### 2.2.8 Barnes

The `barnes` application simulates the gravitational interaction of a system of  $n$  bodies. It uses the  $O(n \log n)$  Barnes-Hut algorithm rather than the  $O(n^2)$  direct pairwise computation. A brief description of this application and its parallelization appears in the SPLASH report[SWG92], and [SHG92] is a more thorough study.

---

<sup>3</sup>`Mp3d` has since been rewritten at NASA-Ames using spatial decomposition techniques and has improved locality of reference[LLJ<sup>+</sup>92]. Unfortunately this improved code has not been publicly released.

Application	Access Rate	Reuse of Data	Comment
sieve	high	very high	regular access intervals
blkmat	low	low	varying intervals between accesses
sor	high	high	many barrier synchronizations
ugray	medium	medium	complex data structures
water	medium	medium	bursty traffic
locus	high	high	linear sequences of accesses
mp3d	high	low	changing data usage
barnes	medium	high	complex data structures

Table 2.2: Summary of Application Characteristics.

In this application, bodies are organized in a three dimensional hierarchical structure called an octree. This allows aggregation of distant particles for computational efficiency, but individual access to nearby particles for computational accuracy. This is a well crafted implementation that assigns neighboring particles to the same thread, and thus there is much reuse of data.

Hierarchical structures are used for both the organization of data and the partitioning of work among the threads. Building tree like structures can not be completely parallelized since there is little concurrency near the root of a tree. With large numbers of threads, a substantial amount of time is spent waiting for synchronization events. This is due both to incomplete parallelization of tree operations and to imperfect load balancing among the threads.

The main characteristics of `barnes` are:

- Moderate access rates.
- High reuse of data.
- Many long synchronization stalls.

### 2.2.9 Summary of Application Characteristics

Table 2.2 summarizes the preceding application discussions in terms of the applications' access rates and reuse of data. These characteristics will affect the results of the experiments presented in this dissertation. High access rates, for example, will require multithreading to use many threads per processor, and low reuse of data will cause caching to perform poorly.

## 2.3 Simulation System

In order to conduct this research we have built a fast and accurate simulator called FAST (for Fast Accurate Simulation Tool). In this section we summarize a few important details and then discuss the usage of the simulator for the studies conducted as part of this thesis. Appendix B contains a detailed discussion of the simulator and the techniques and tradeoffs chosen in its design.

### 2.3.1 The Simulator

The simulator is based on the technique of execution driven simulation. This is a process whereby the application program to be simulated is actually directly executed, but it has been modified so that it counts its own execution time and returns control to the simulator at special events, such as shared memory references. The simulator works by executing the many parallel threads for small periods of time, and then scheduling the resulting events so that they are all simulated in a correct global time order.

The modifications to the application program are best made at the object code level, since at this level accurate timing can be determined based on the individual assembly language instructions. All of our applications were compiled at optimization level “-O2”, and their timing results are based on this.

The simulator accurately models the timing of the MIPS R3000[Kan89] pipeline, and all interactions between threads are accurately ordered. One slight inaccuracy occurs for simulations using caching of shared data: the cache interactions, such as invalidations, are done instantaneously rather than being delayed for the transit time for the invalidation messages to travel from the directory to the cache. This simplification makes the cache simulator much more efficient and easier to write, but means that data gets invalidated slightly sooner than it would on a real machine.

Because of careful use of execution driven simulation techniques, our simulator is approximately 50 times faster than comparable simulators such as Tango[DGH91] or [O’K89]. The main advantage of this speed is that it allows us to run longer and larger simulations (and thus more representative of large systems) than those of previous researchers.

Experiment: efficiency on an ideal machine			
Application	Processors	Multithreading	• Latency = 0 cycles
sieve	1-1024	1	
blkmat	1-1024	1	
sor	1-1024	1	
ugray	1-512	1	
water	1-343	1	
locus	1-128	1	
mp3d	1-1024	1	
barnes	1-512	1	

Table 2.3: Experimental parameters for measuring the execution efficiencies on an ideal machine.

### 2.3.2 Simulation Constraints

There are a number of constraints that have kept us from running simulations as large as we would have liked. First, some of the applications (**water** and **locus**) had only moderate input sizes available. Second, despite our fast simulator, simulation is still time consuming. We thus restricted problem sizes so that individual simulations completed within a few hours. And third, simulations of large parallel machines require a lot of space to hold the state of the many simulated threads and caches. We were limited to 128 mega-bytes that was available on the largest of our simulation host machines.

Table 2.1 listed the input sizes that were used for each of the applications. In order to gauge the amount of parallelism available, we have simulated the applications as if they were executing on an “ideal” machine that had 0 latency and no contention on accesses to shared memory. Such a machine would be impossible to build, but it corresponds to an upper bound on achievable performance. Table 2.3 list the experiment’s parameters and Figure 2.7 shows the results.

Rather than show the standard speedup curves (speedup = execution time on 1 processor / execution time on  $P$  processors), we have plotted the efficiency vs. the number of processors (efficiency = speedup /  $P$ ). Efficiency is much like processor utilization. The difference is that efficiency is directly related to performance, where as utilization is simply a metric of how busy the processors are. For example, processors might be busy spinning or doing redundant work and thus not contributing to overall speedup. The advantage of efficiency over speedup is that it has been normalized by the number of processors and

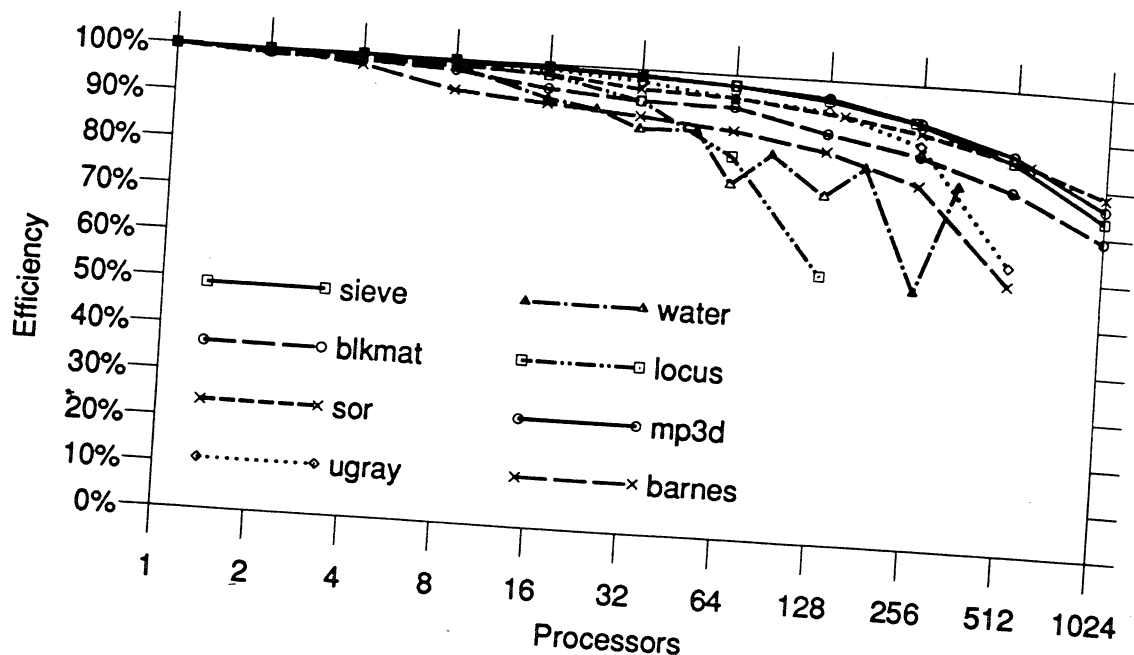


Figure 2.7: Efficiency on ideal (0 latency) machine.

thus can be used to compare machines of different sizes. As can be seen from Figure 2.7, efficiency is a fairly constant metric until the number of processors exceeds some limit, at which point it drops more quickly.

We are simulating fixed size problems, and thus as the number of processors is increased, the work gets partitioned more finely among a larger set of threads. The efficiencies degrade at some point because of various imperfections in the parallelization such as: uneven load balancing, synchronization overhead, redundant calculations, and occasionally restricted parallelism (such as at the root of a tree). **Water** stands out in Figure 2.7 because of its jagged efficiency curve. This is the result of poor load balancing that occurs when the number of processors is incongruent to the number of molecules (343), as was explained in Section 2.2.5.

Figure 2.7 was used to choose a “reasonable” limit on the number of threads (amount of parallelism) that could be used for each application given the fixed problem sizes that we were able to simulate. These thread limits are shown in Table 2.4. For the various simulation experiments in this thesis, these thread limits constrained the number of threads and processors that were used.



Thread Limits	
sieve	256
blkmat	256
sor	256
ugray	256
water	120
locus	64
mp3d	256
barnes	128

Table 2.4: Limits on the number of threads used from the various applications because of increasing performance degradation when trying to extract greater parallelism from fixed size problems.

These thread limits are not strict, and occasionally we exceed them. The main point is that with fixed problems sizes there is also a limit on the amount of parallelism available from the applications. If these applications are to be used on larger parallel machines, larger problem sizes will be needed to supply additional parallelism.

### 2.3.3 Revised Machine Model

In Section 2.1.1 we selected a network latency of 200 cycles based on our expectations of the latency commensurate with a 1000 processor machine. However because of the simulation constraints just discussed, we are unable to run problems large enough for such a machine. We have therefore reduced the number of processors, but kept the same network latency. Our revised machine model is depicted in Figure 2.8. Our typical simulation will be performed with from 16 to 32 processors.

We feel this reduction of the number of processors but maintaining the latency makes our results more directly applicable to a 1000 processor machine. One of the main objectives of this research is evaluating multithreading as a means of tolerating the long memory latencies of large machines. It will turn out that the main factor determining the efficacy of multithreading is the behavior of the individual threads. If threads execute for relatively long intervals between remote references, and if these intervals are fairly constant in length, then multithreading will work well. The "interval" behavior of these threads is primarily determined by the sequences of instructions in the application codes themselves, and not by the number of parallel threads or the size of the problems. Thus with larger

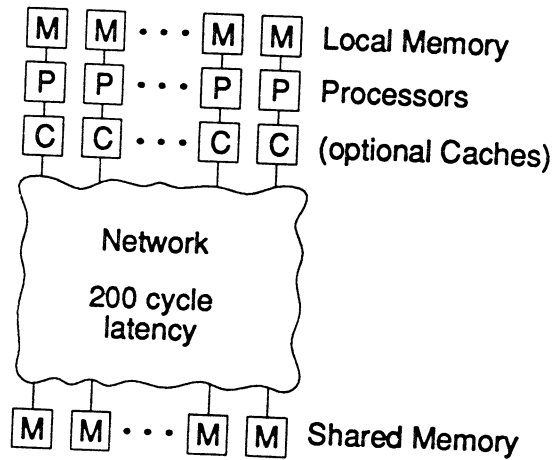


Figure 2.8: Revised model of parallel machine with fewer processors but the same latency as expected on a 1000 processor machine.

problems, we expect to see the same multithreading behavior on large machines as we observe in our studies of the reduced machine model.

A final note on our simulations is that all results are based solely on the parallel phase of computation. All of the applications studied in this dissertation have a sequential initialization phase, a parallel computation phase, and a sequential termination phase. It is common practice in simulation studies to report only the parallel phase. This is done for a number of reasons. First, many of the application are iterative, but only a small number of iterations is simulated, thus artificially decreasing the duration of the parallel phase and thereby increasing the significance of the sequential phases. Second, as problem sizes are increased, the sequential phases become a smaller and smaller fraction of the total computation[Gus88]. Third, many of these applications were written for today's small shared memory machines, and often much of the initialization could have been parallelized, but this was not deemed necessary on a small machine. Finally, a large part of the initialization and termination phases is input and output, which we expect to be done in parallel for larger machines.

## Chapter 3

# Behavior of Multithreading

In this chapter we present an abstract model of multithreaded systems. Its analysis provides intuition into the behavior that we will later observe in our simulation studies.

### 3.1 Multithreading Model

Our model of a multithreaded processor was formulated by Saavedra-Barrera, Culler, and von Eicken in [SBCvE90]. Following is restatement of the model and its basic analysis using the notation of this dissertation.

The model considers just a single processor of a parallel computer. We assume that a thread on this processor repeatedly issues remote references at an interval of  $R$  cycles (called the run-length), and that after issuing a remote reference, the thread must wait for the response to return before resuming execution. The response time depends upon the delay through the interconnection networks to and from memory, and for the model we assume a fixed round-trip latency of  $L$  cycles.

These two parameters,  $R$  and  $L$ , are shown in Figure 3.1. In this figure there is only a single thread executing on the processor, and the processor will have a utilization factor of  $R/(R + L)$ . For large shared memory parallel computers we expect  $L$  to be substantially larger than  $R$ , as shown in the figure, and thus the processor will be poorly utilized.

Figure 3.2 shows the same situation, but with three threads per processor. The number of threads per processor will be called  $M$  for multithreading level. Between threads we have added a context switch cost of  $C$  cycles, which we expect will be just a few cycles or even zero cycles with appropriate hardware support.

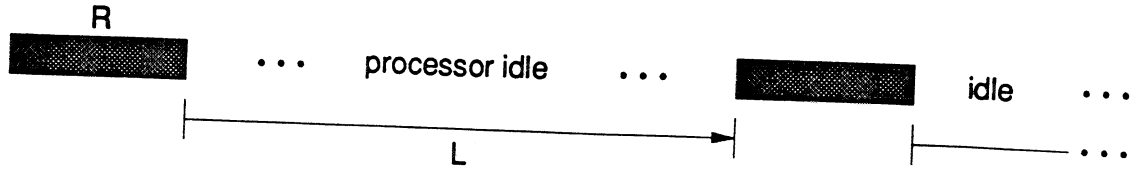


Figure 3.1: Model of a single thread. ( $R$  = Run-length,  $L$  = Latency)

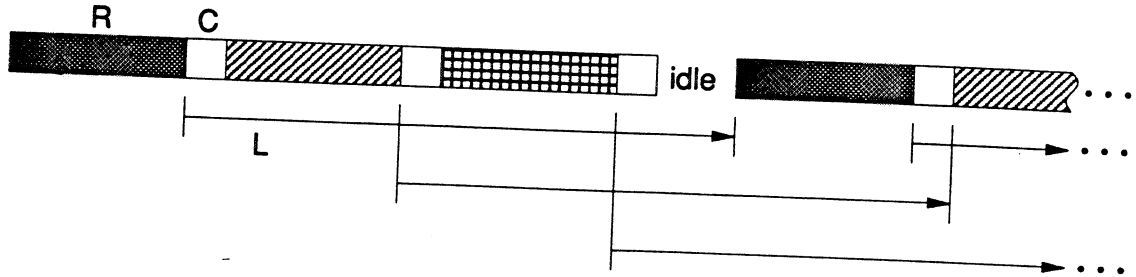


Figure 3.2: Multithreading with 3 threads per processor. ( $C$  = Context switch cost)

### 3.1.1 Analysis Under Constant Run-Lengths

Under simple assumptions about  $R$ ,  $L$ , and  $C$ , we can compute the processor utilization as a function of the multithreading level  $M$ . The simplest assumption is that  $R$ ,  $L$ , and  $C$  are all constants. Under this assumption, the performance analysis can be broken into two separate cases. The first case occurs when there are not enough threads to hide the latency, and thus the processor is sometimes idle. Figures 3.1 & 3.2 both show this case. The system forms a renewal process[Wol89] that starts at the end of each idle period. The length of the renewal period is  $R + L$  and the amount of work done is  $M \cdot R$ . Thus processor utilization is  $MR/(R + L)$ . In this case performance increases linearly with the multithreading level. The second case occurs when there *are* enough threads to hide the latency. In this case the only performance loss comes from the context switch overhead, and thus processor utilization is  $R/(R + C)$ . The boundary between the two case occurs when  $M(R + C) = R + L$ . Solving for  $M$  we get:

$$\text{Processor Utilization} = \begin{cases} \frac{MR}{R+L} & \text{if } M < 1 + \frac{L-C}{R+C} \\ \frac{R}{R+C} & \text{otherwise} \end{cases}$$

If  $C$  is small, we can approximate the number of threads needed to maximize

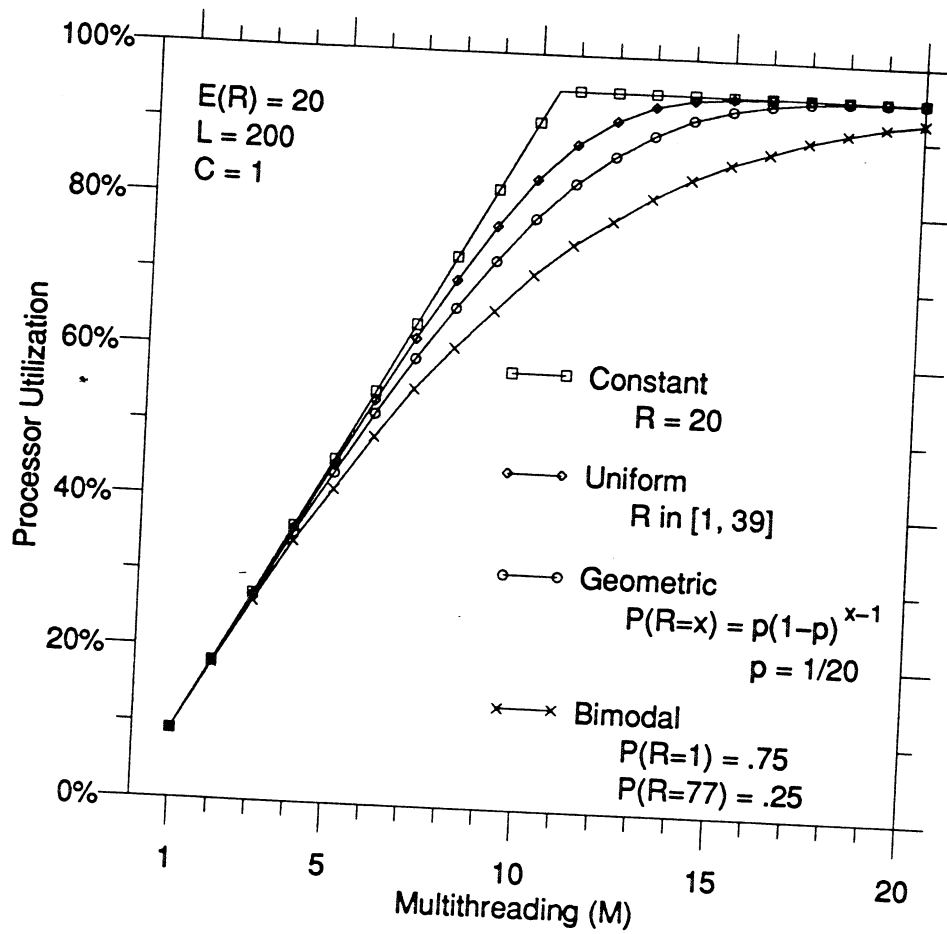


Figure 3.3: Processor utilization as a function of multithreading for various run-length distributions.

processor utilization as  $M = 1 + L/R$ .

**Rule 1** With a constant run-length distribution, approximately  $M = 1 + L/R$  threads are needed to keep the processor busy.

This function is shown graphically in Figure 3.3 as the curve labeled “constant”, using the values of  $R = 20$ ,  $L = 200$ , and  $C = 1$ , which are similar to the values that we expect for real applications and hardware.

### 3.1.2 More Complex Distributions

In real applications the run-lengths will rarely be predictable. A better model is to choose the run-lengths from some random distribution. Figure 3.3 also shows the processor utilization curves for three other distributions: uniform, geometric, and bimodal. Each of these is parametrized to have the same mean run-length ( $R = 20$ ). For the uniform distribution, the run-lengths are chosen with equal probability over the range 1 to 39. For the geometric distribution, the run-length comes from a sequence of biased coin flips where at each step the probability of completion is  $P = 1/20$ . And for the bimodal distribution, the run-length is either:  $R = 1$  with 75% probability, or  $R = 77$  with 25% probability.

For the geometric run-length distribution the model was solved by Saavedra-Barrera and Culler[SBC91] using Markov chain analysis. However for more general distributions, analytic solutions become intractable. We have extended the applicability of the model by using numeric simulation to compute the processor utilization versus multithreading curves for any specified distribution. For the uniform and bimodal distributions, the curves in Figure 3.3 were calculated using this technique.

The histograms of these distribution functions are shown in Figure 3.4. These histograms are drawn with triangular piles whose area represents the percentage of total run-lengths having a particular value. Piles that would overlap are combined to make a larger pile. Refer to appendix A for a complete explanation and rationale of this new and somewhat odd histogram. These histograms will be used as a basis for building intuition into multithreading behavior.

By comparing the distributions and the plots of their performance, we can observe that the distributions with the most short run-lengths need the highest multithreading levels. This occurs even though the mean run-length remains the same (20) for all of these distributions. The short run-lengths cause problems when, by random chance, several threads have short run-lengths in succession, and the remaining threads are unable to hide the latency. In these cases the processor will be forced to stall. In the opposite case, when several successive threads all have long run-lengths, the latency is easily covered, but the excess latency tolerance is wasted.

**Rule 2** *When run-lengths are random, the presence of short run-lengths increases the multithreading level needed to keep the processor busy.*

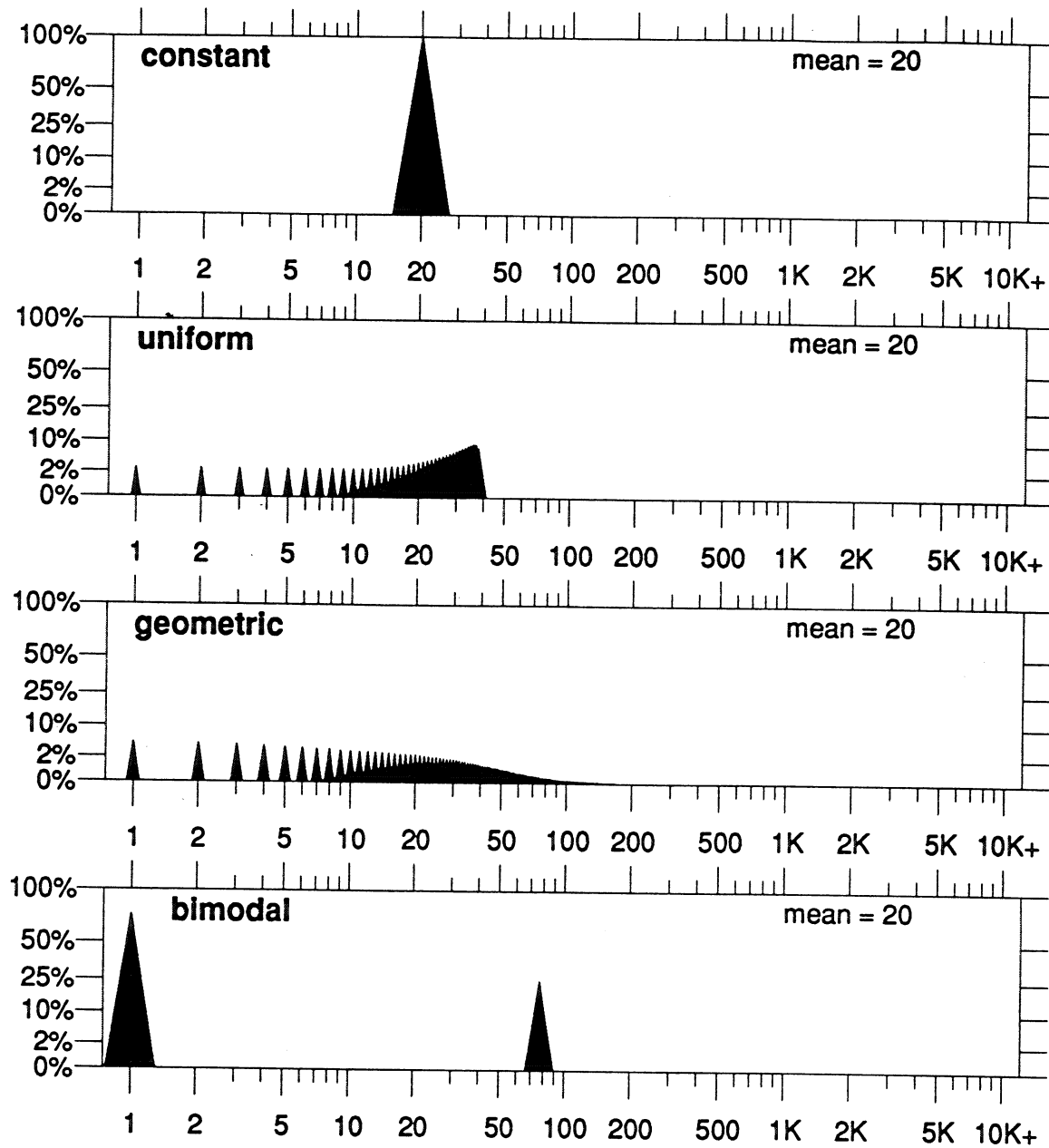


Figure 3.4: Histograms of distribution functions. The horizontal axis shows the run-length. Each data point is represented by a pile who's size corresponds to its percentage of the total, and overlapping piles combine together to make taller piles.

Experiment: run-lengths under switch-on-load			
Application	Processors	Multithreading	
sieve	16	12	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 0 cycles</li> <li>• Scheduling = round robin</li> <li>• No shared memory caches</li> </ul>
blkmat	32	4	
sor	8	45	
ugray	8	16	
water	10	12	
locus	2	32	
mp3d	8	28	
barnes	12	13	

Table 3.1: Experimental parameters for run-lengths under **switch-on-load**.

### 3.2 Applications' Run-Length Distributions

Figures 3.5 & 3.6 show the run-length distributions for the benchmark applications under the **switch-on-load** multithreading model. These run-length distributions were collected from simulations as specified in Table 3.1. The performance results and simulation assumptions will be discussed in the next section. We discuss the run-length distributions first so that we can make predictions based on what we have learned from the multithreading model.

The applications have widely different run-length distribution functions and means. The first four applications are in fact similar to the four mathematical distributions given in the previous chapter: **sieve** is similar to the constant distribution, **blkmat** (excluding the pile near 20) is similar to the uniform distribution, **sor** is similar to the bimodal distribution, and **ugray** is similar to the geometric distribution. Also note that **water** has some very long run-lengths (around 5000 cycles). These occur because this application was written so that it copies shared data values into local memory and then performs large calculations using only the local copies. The **blkmat** application also makes local copies of shared data, and because of this, it achieves the highest average run-length (164 cycles). **Water** (43) and **barnes** (42) also exhibit long mean run-lengths. **Ugray** (22) and **sieve** (19) have more moderate run-lengths. And **mp3d** (9.5), **locus** (7), and **sor** (6) have short mean run-lengths.

Based on the mean run-lengths and the distributions functions, we can make predictions about how these applications will perform on a multithreaded machine. Because of



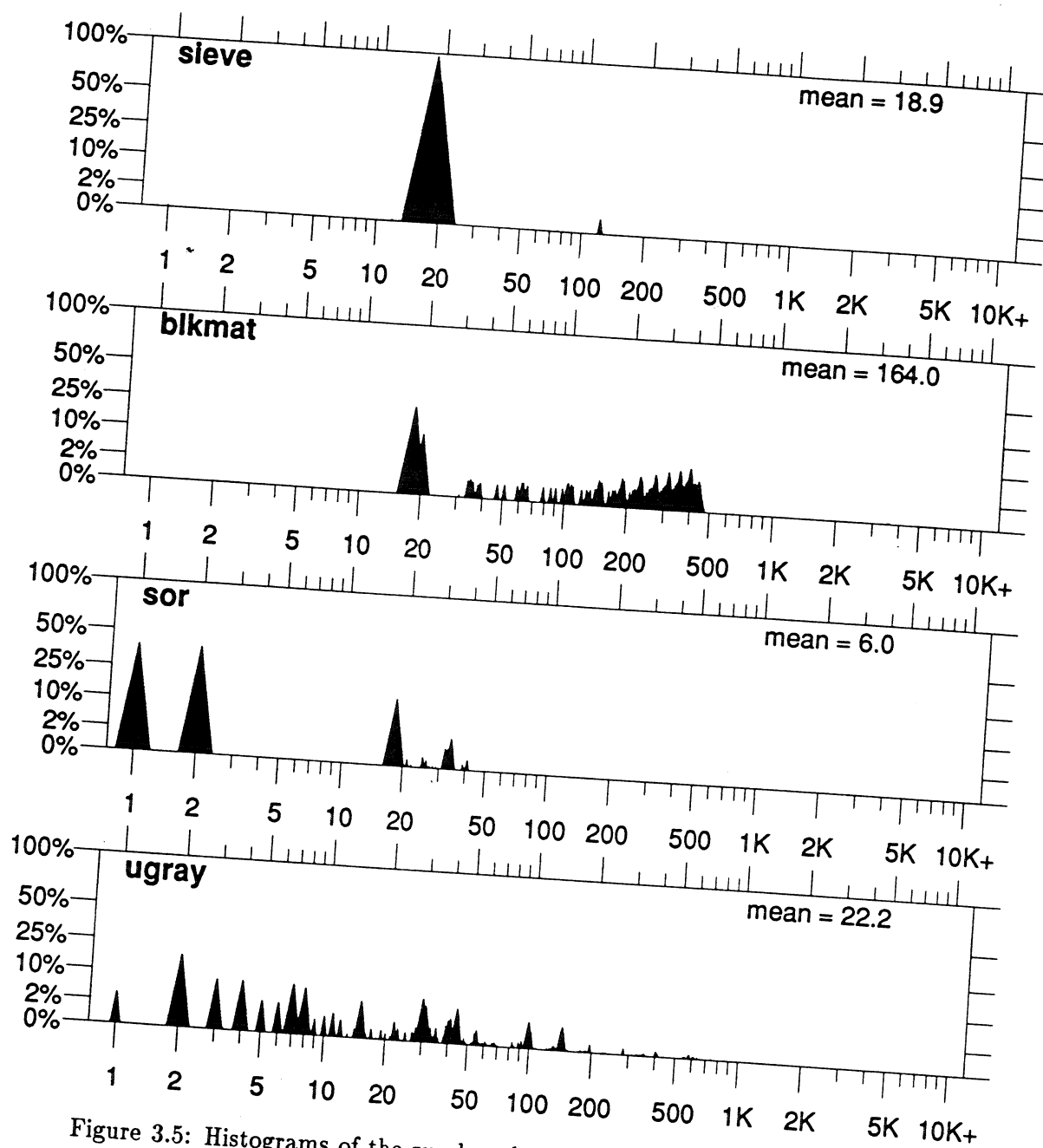


Figure 3.5: Histograms of the run-lengths distributions of the applications running under **switch-on-load**.

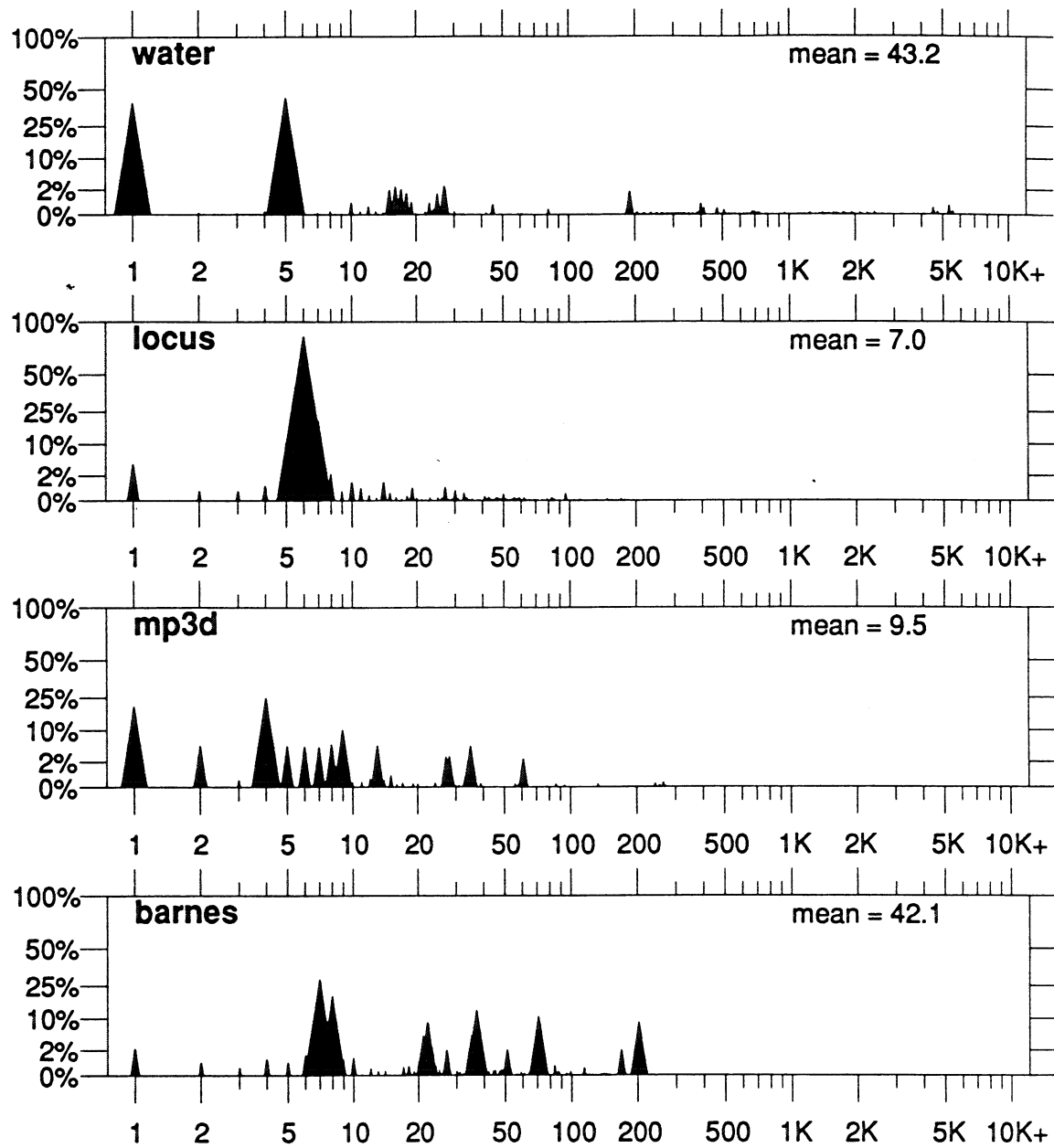


Figure 3.6: Histograms of the run-lengths distributions of the applications running under **switch-on-load**.

Experiment: switch-on-load			
Application	Processors	Multithreading	
sieve	16	1-20	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 0 cycles</li> <li>• Scheduling = round robin</li> <li>• No shared memory caches</li> </ul>
blkmat	32	1-20	
sor	8	1-50	
ugray	8	1-20	
water	10	1-34	
locus	2	1-40	
mp3d	8	1-40	
barnes	12	1-20	

Table 3.2: Experimental parameters for **switch-on-load**.

short mean run-lengths, we can expect **sor**, **locus**, and **mp3d** to require high multithreading levels to keep the processor busy. Also, **ugray** and **water** will require extra multithreading because of the large number of short run-lengths in their distributions.

### 3.3 Testing the Multithreading Model

In this section we compare the performance predicted by the multithreading model to the actual performance observed in simulations. The parameters of the simulation experiments are shown in table 3.2. We have assumed a 200 cycle remote access latency and a context switch cost of 0 cycles<sup>1</sup>.

Many of the applications will require a large multithreading level in order to reach high execution efficiencies, but the total number of threads available is limited by the fixed problem sizes that we are able to simulate (as discussed in Chapter 2). Therefore, for each application, we have taken the multithreading level ( $M$ ) needed in order to achieve high efficiency, and selected the number of processors so that  $P \cdot M$  is approximately equal to the thread limit. The results are presented here as if a just a single set of experiments were performed, but, in fact, preliminary experiments were also performed in order to determine the multithreading levels needed by the applications.

For some applications, such as **locus**, the resultant number of processors used was quite small. In later experiments, with better multithreading models that require fewer threads per processor, we will increase the number of processors used in our simulations.

<sup>1</sup>The zero cycle context switch is justified in Chapter 8.

Figures 3.7 & 3.8 show the predicted and observed performance of the applications under the **switch-on-load** multithreading model. The predicted performance is based on the multithreading model and the run-lengths distributions presented in the previous section. The observed performance is based on the simulations.

The multithreading model predicts *processor utilization* rather than our preferred metric of *execution efficiency*, which we use for most of the results presented in this dissertation. The model predicts processor utilization because the run-length distributions fed into it reflect the entire parallel execution. Some of this execution may include extra operations performed by the parallel program that are not performed by the sequential program. These extra operations keep the processor busy, but they do not contribute to application speedup. Figures 3.7 & 3.8 show both the processor utilizations and the execution efficiencies observed in the simulations. For some applications (**water**, **mp3d**, and **locus**) the processor utilizations and execution efficiencies are indiscernible from each other. For the others, the gap between utilization and efficiency arises because of the extra operations done by the parallel programs. **Locus** and **ugray**, for example, both do dynamic job scheduling and use spinning to wait for jobs to become available. This spinning keeps the processors busy but does not perform useful computation. **Sieve**, **blkmat**, and **sor** also exhibit a gap between processor utilization and efficiency. These applications do static partitioning of the work among the threads. Each thread does the partitioning calculation, and thus with more threads, more time is spent doing these partitioning calculations that are not needed by the sequential programs. All of the applications actually have parallel overheads. They are just much more visible for **sieve**, **blkmat**, and **sor** because these applications have shorter execution times than the other applications.

For most of the applications, there is also a large gap between the processor utilization predicted by the multithreading model and the processor utilization observed in the simulations. This gap arises because the processors sometimes sit idle or underutilized while threads wait on synchronization or because of imperfect load balancing. The jaggedness in the processor utilization curves for **sor** and **water** is an indicator of this load imbalance problem. Another inaccuracy of the multithreading model is that it assumes that the run-lengths drawn from the distribution are mutually independent. In actuality, the applications proceed through different phases of their computations; some phases have short run-lengths, and some phases have long run-lengths. All of these reasons contribute to the optimistic predictions of the multithreading model.

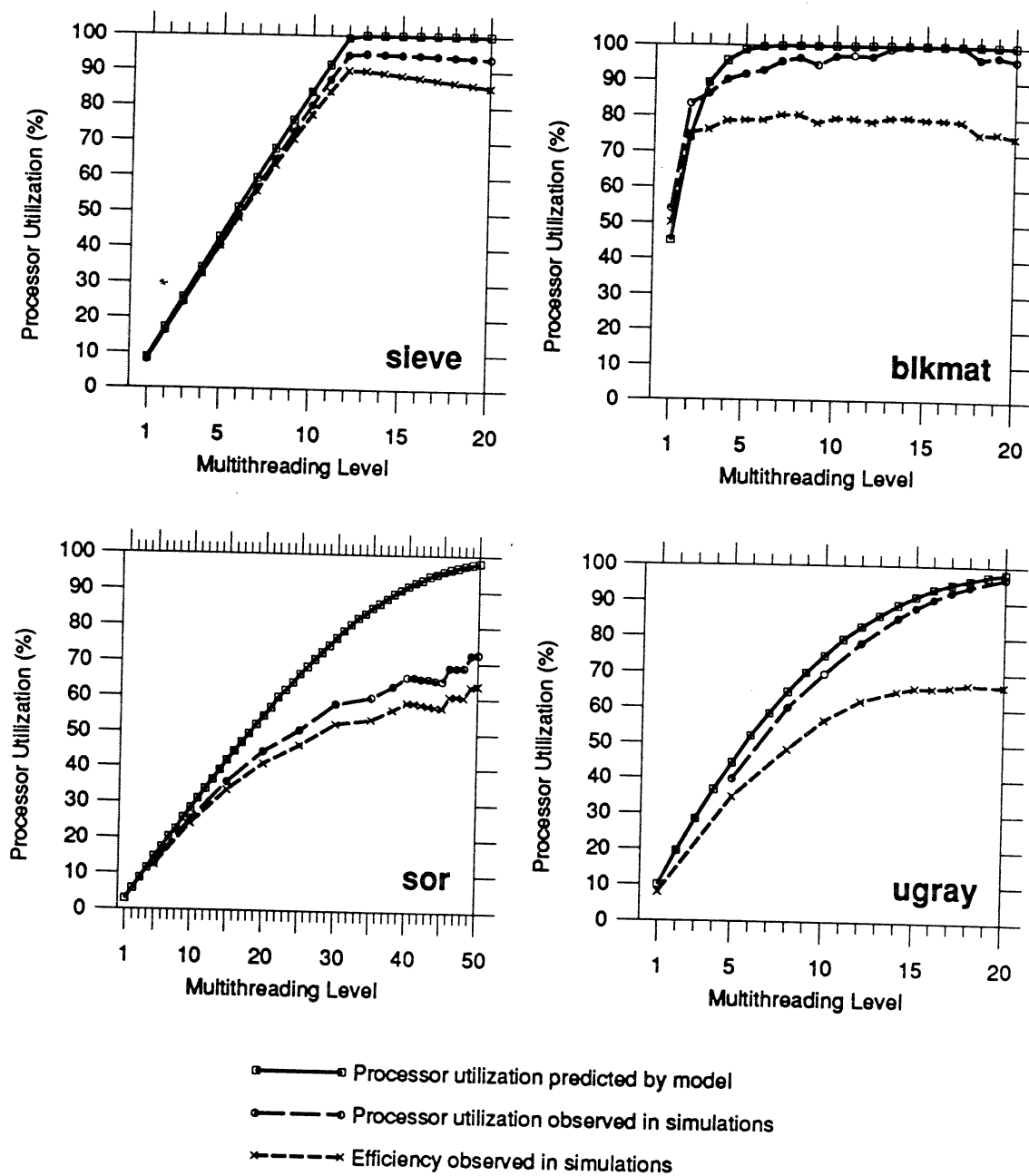


Figure 3.7: Predicted and observed performance for **switch-on-load**.

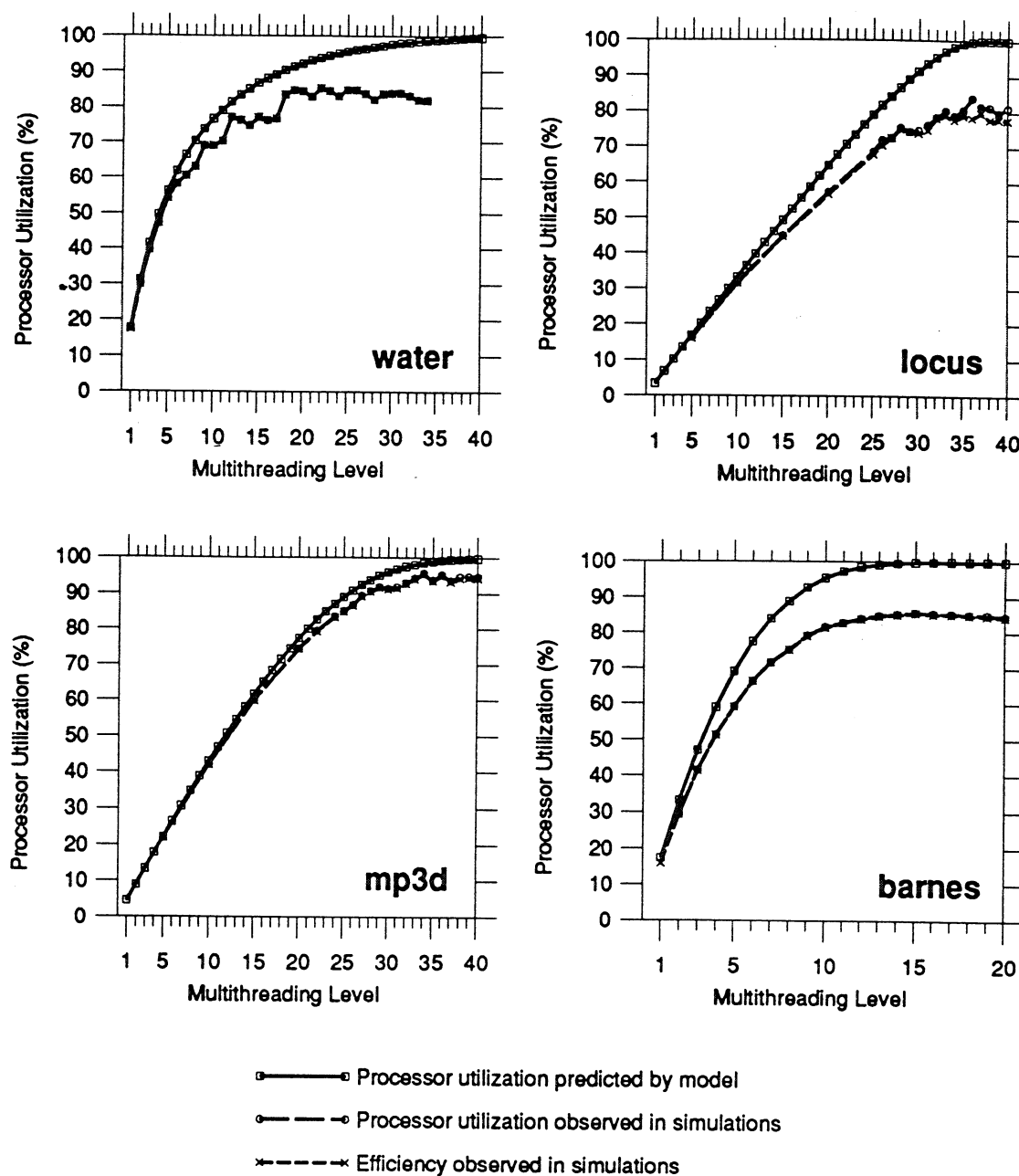


Figure 3.8: Predicted and observed performance for **switch-on-load**.

**Sor** has a very strong correlation between threads. The threads all have a repeating pattern of four short run-lengths followed by a long run-length. Since the threads all start in synchrony, when one thread has a short run-length, so do all the others. Likewise the threads all have their long run-lengths during the same scheduling cycle. If the threads were not so well synchronized, long and short run-lengths would be intermixed and the overall latency hiding would improve. The exact synchrony can be jumbled by random scheduling, and for **sor** it improves the efficiency at  $M = 40$  from 59% to 73%. Other applications and other multithreading models do not benefit from this random scheduling, and thus we do not pursue it further.

Usually the performance model predicts processor utilizations higher than what is achieved. An exception occurs for **blkmat** at multithreading levels of 1 or 2. Unlike the rest of the applications, the run-length distribution for **blkmat** varies considerably with the problem size and number of threads. **Blkmat** partitions the computation into blocks (as described in Section 2.2.2), with one block per thread. It makes a local copy of the shared data used within a block, and thus with larger blocks (fewer threads), the average run-length between shared memory references increases. The run-length distribution used by the performance model was obtained from a simulation of 32 processors at a multithreading level of 4, and thus the simulations at smaller multithreading levels had longer run-lengths.

### 3.4 Conclusions

The performance model presented in this chapter provides intuition into the behavior of multithreaded systems. If run-lengths are constant, a multithreading level of  $M = 1 + L/R$  is needed to hide the latency. For other distributions, more multithreading is required, particularly for those distributions with short run-lengths.

For real applications, the situation is more complex. Unlike our mathematical model of program behavior, real applications have varying behavior over time, and their threads are not independent. In **sor**, for example, there are alternating phases of computation and convergence checking. The convergence checking phases have little computation and thus shorter average run-lengths than the computation phases. Also, threads synchronize with each other and wait (because of load imbalance) for other threads to complete their calculations. While modeling does give some insights into the behavior to expect from multithreaded systems, in the rest of this dissertation we present only the simulation results.

## Chapter 4

# Multithreading Without Caching

In this chapter we continue the evaluation of **switch-on-miss** and also evaluate the **explicit-switch** multithreading model. These are machine models that do not provide caching of shared data. Their advantage over systems with caching is that they avoid the cost and complexity of cache coherency. Their disadvantages are that they will context switch more frequently, require more threads per processor, and use more network bandwidth.

The **switch-on-load** model will turn out to perform poorly compared to **explicit-switch** because it context switches too frequently. The **explicit-switch** model solves this problem by providing a mechanism for the compiler to group together several shared accesses. The processor can then issue the entire group of messages into the network before it context switches.

### 4.1 New Format

The previous chapter gave performance results for simulations of **switch-on-load**. These results were presented as plots of efficiency versus multithreading level, as shown in Figure 4.1(a). We are mainly interested in the level of performance that can be obtained and the multithreading level needed to obtain it. For future results, we therefore condense this graph into a single bar as shown in Figure 4.1(b). The height of the bar shows the efficiency obtainable, and the number at the top indicates the multithreading level required. The other lines indicate the efficiencies obtainable at lesser multithreading levels.

In this example, the highest bar we show is for  $M = 19$ . At this level the efficiency



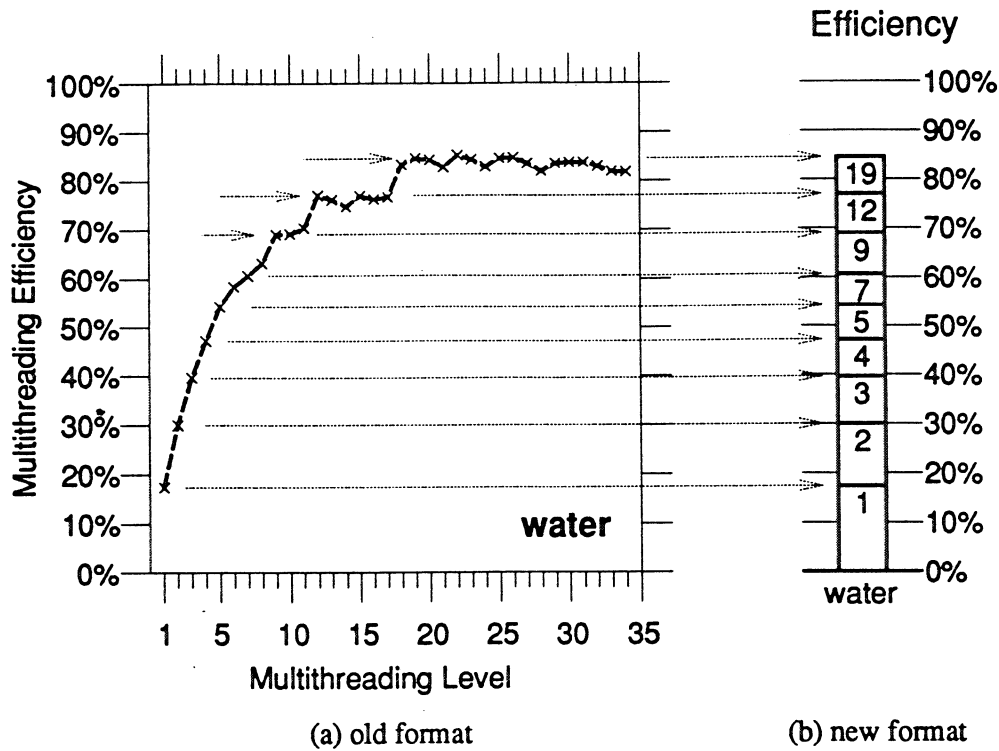


Figure 4.1: New format for presenting multithreading efficiency results.

is 84.6%. A slightly higher efficiency of 85.2% is achieved at  $M = 22$ , but we do not show it because such a minor increase in performance would probably not be worth increasing the multithreading level. Although we may hope that applications will have abundant threads, for many applications threads will be a limited resource. In this and other results presented in this bar graph format, we report the highest efficiency up to the point where the efficiency increases by less than 1% per additional thread.

## 4.2 Switch-On-Load

Figure 4.2 shows the switch-on-load multithreading efficiencies in the new format. Many of the applications need large multithreading levels. Particularly high are `sor` ( $M = 40$ ), `locus` ( $M = 32$ ), and `mp3d` ( $M = 29$ ). Furthermore, even at high multithreading levels, some applications are achieving only moderate efficiencies: `sor` (59%), `ugray` (66%).

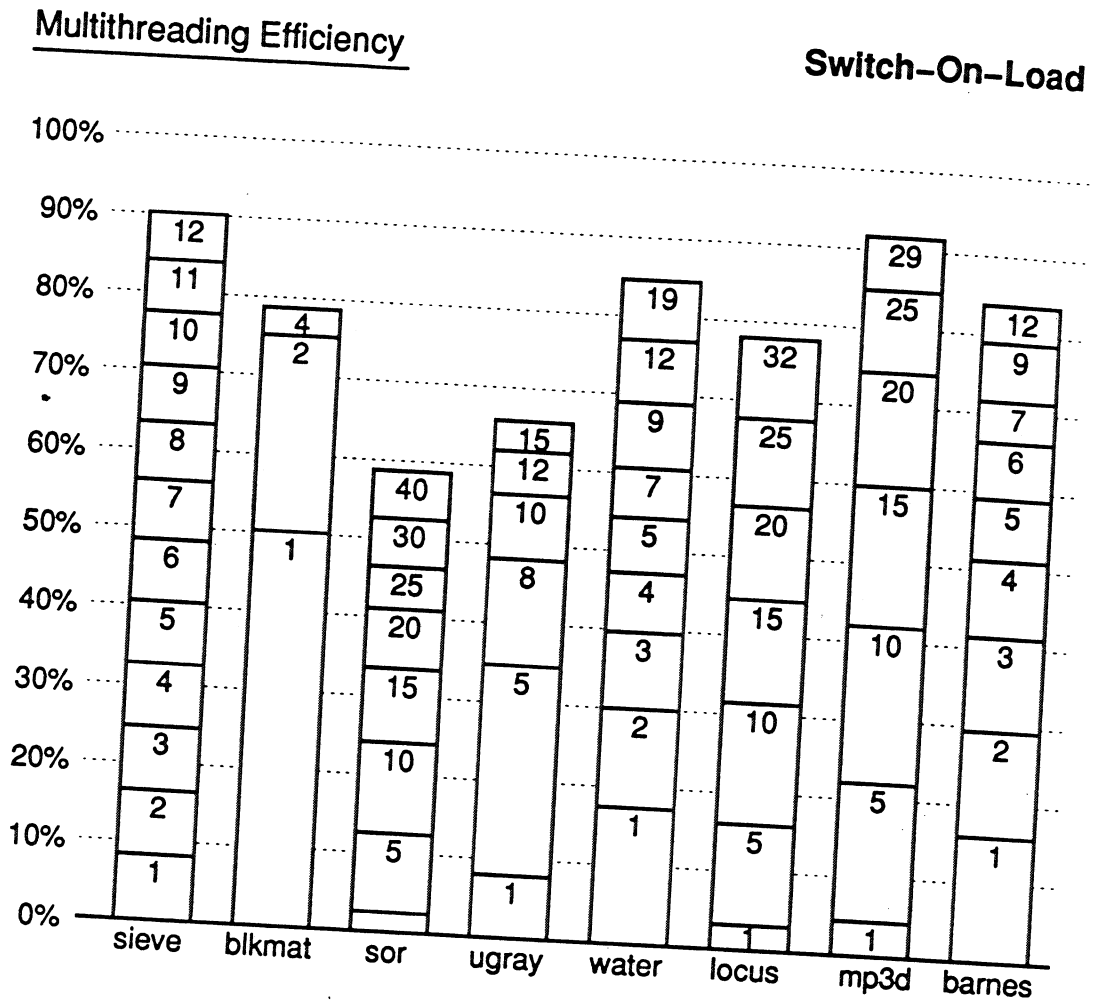


Figure 4.2: Multithreading levels and the efficiencies they achieve under switch-on-load.

These high multithreading levels were predicted by the multithreading model because of the short average run-lengths and the many short run-lengths in the distributions. With such high multithreading levels, large problem sizes will be required in order to provide sufficient parallelism. Also the hardware cost to support these multithreading levels will be large because of the large number of register sets required. Machines, such as the HEP[Kow85], have been built that provide more than enough register sets, however decreasing the required multithreading level has many benefits: smaller problem instances can utilize the full set of processors, less hardware is needed to hold the register sets, and less application overhead is incurred when fewer threads are used.

### 4.3 Increasing the Run-Lengths: Explicit-Switch

The key to decreasing the multithreading level and increasing performance is to increase the run-lengths. This involves both raising the average run-lengths and eliminating short run-lengths. To do this, a thread must be allowed to issue more than one reference into the network before being context switched. There are two multithreading models that address this: **switch-on-use** and **explicit-switch**.

Under **switch-on-use**, the hardware issues remote loads into the network and continues executing the same thread. It context switches only when the thread tries to use a value that has not yet returned. If the compiler can arrange instructions so that several remote loads are grouped together, the loads will all be issued into the network before the thread tries to use any of the results and is forced to context switch. This will eliminate excess context switching and thereby increase run-lengths.

Another way to allow issuing multiple loads before context switching is to provide an explicit context switch instruction that the compiler can insert between the group of loads and the later uses of the requested data. The effect is the same as under **switch-on-use**. The difference is that the hardware may be a little simpler under **explicit-switch** than under **switch-on-load** because it does not have to check the status of registers as they are used. In this section we will explore performance of the **explicit-switch** model. We expect that the results for **switch-on-use** would be virtually identical. Relevant hardware issues are discussed in Chapter 8.

#### 4.3.1 Grouping Within Basic Blocks

The inner loop of the **sor** application is shown in Figure 4.3(a) as an example. Without grouping, the 5 loads are issued one at a time, with a context switch after each one. In Figure 4.3(b) the code has been reorganized so that all 5 loads are grouped together and are then followed by a single context switch instruction. Rather than having four short run-lengths followed by one long run-length, there is now just a single long run-length.

A compiler designed for a multithreaded architecture will group shared loads whenever possible. Since the compilers we have today do not do this grouping, we wrote a post-processor which finds the basic blocks in an object file<sup>1</sup>, does dependency analysis within the

---

<sup>1</sup>A basic block is a sequence of instructions that are executed without any branches into or out of it except at the ends.

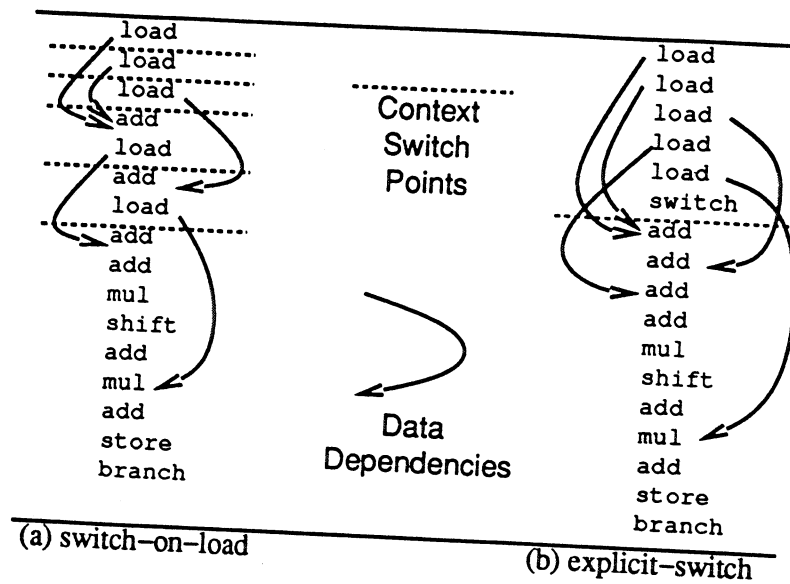


Figure 4.3: Inner loop of `sor` under **switch-on-load** and reorganized for **explicit-switch**.

basic blocks, and then reorganizes the instructions so as to group the shared loads together. The reorganization is performed by percolating shared load instructions upward through the basic block as far as possible while still obeying all data dependencies. Some register renaming is used to eliminate artificial dependencies caused by the original compiler's register assignment, but because we do this analysis at the assembly language level, we must make pessimistic assumptions<sup>2</sup> which restrict our ability to reorganize the code. After the shared load instructions have been percolated upward, the code reorganizer inserts context switch instructions as needed to separate the groups of independent loads from the use of their results. Despite the limited knowledge available when optimizing assembly language, our code reorganization appears to work very well for basic blocks.

Table 4.1 shows the grouping obtained by reorganizing the basic blocks of our benchmark applications. The grouping factor is the average number of shared loads per context switch. For `sor` and `water` this grouping was very successful, grouping on average almost 5 loads between context switches. Grouping in the other applications was less successful, with `locus`, `sieve`, and `blkmat` having only marginal or no grouping at all.

<sup>2</sup>We assume that every shared store might have a conflict with every shared load because of address aliasing.

Application	Grouping Factor	Mean Run-Length
sieve	1.00	19.3
blkmat	1.00	164.9
sor	4.65	29.2
ugray	1.29	26.4
water	4.76	206.6
locus	1.05	8.3
mp3d	2.28	22.6
barnes	1.68	71.5

Table 4.1: Grouping and mean run-lengths achieved for the applications after reorganization of their basic blocks.

Experiment: run-lengths for explicit-switch			
Application	Processors	Multithreading	
sieve	16	12	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 1 cycle</li> <li>• Scheduling = round robin</li> <li>• No shared memory caches</li> </ul>
blkmat	32	4	
sor	16	8	
ugray	8	12	
water	20	6	
locus	2	28	
mp3d	16	14	
barnes	16	8	

Table 4.2: Experimental parameters for measuring run-lengths for **explicit-switch**.

The new run-length distributions are shown in Figures 4.4 & 4.5; they were obtained from experiments as specified in Table 4.2. These run-length distributions for **explicit-switch** should be compared to the run-length distributions for **switch-on-load** that were shown on pages 40 & 41.

For **sor** and **water** virtually all of the short run-lengths have been eliminated. **Mp3d** and **barnes** still have some short run-lengths, but they show higher mean run-lengths and fewer short run-lengths.

The other four applications show little change. **Locus** had a small amount of grouping that eliminated the shortest (1 or 2 cycle) run-lengths, but this is not very significant because these short run-lengths comprised only 4.5% of the total. The change in

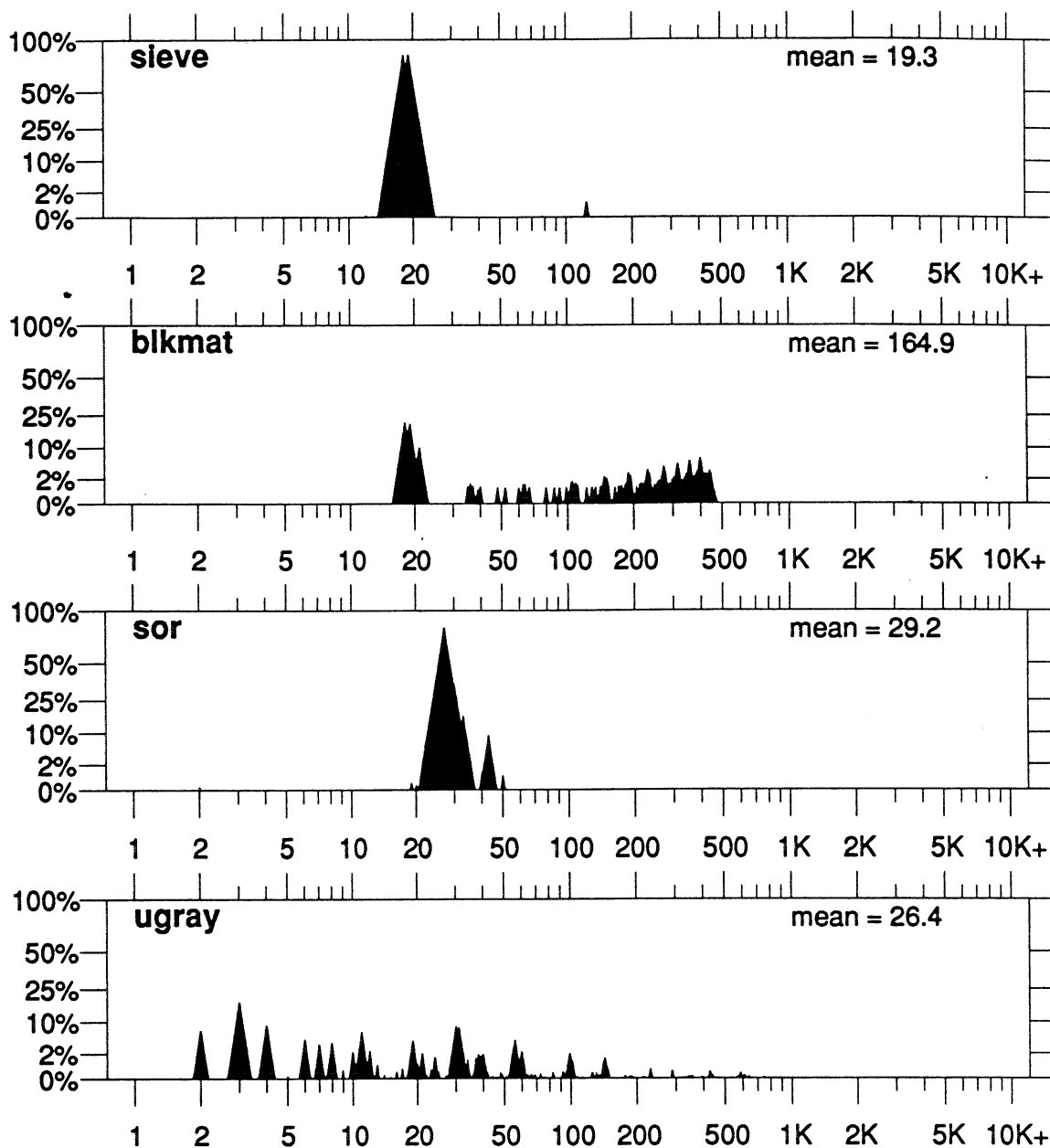


Figure 4.4: Histograms of the run-lengths distributions of the applications running under **explicit-switch**.

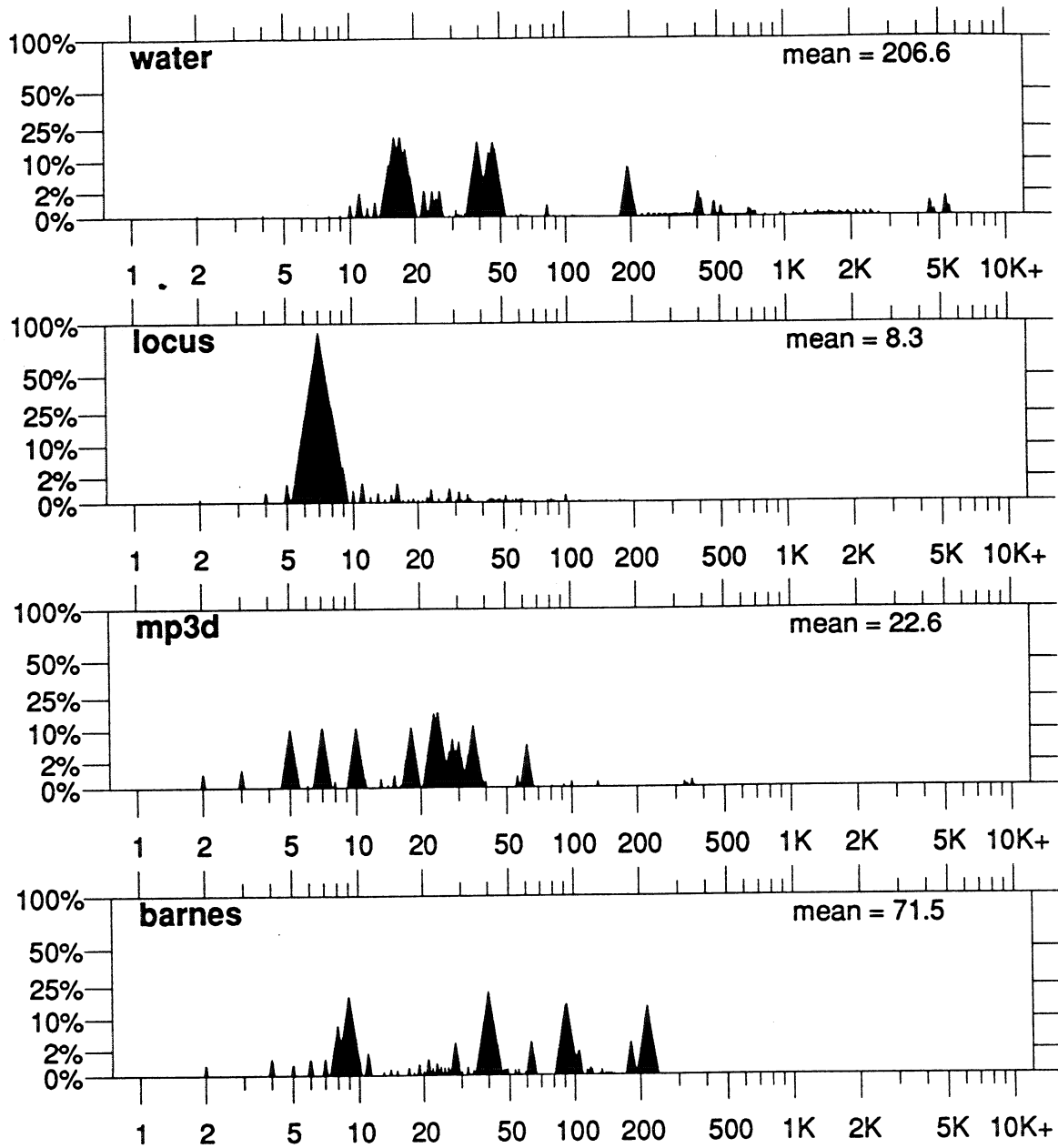


Figure 4.5: Histograms of the run-lengths distributions of the applications running under **explicit-switch**.

Experiment: explicit-switch			
Application	Processors	Multithreading	
sieve	16	1-12	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 1 cycle</li> <li>• Scheduling = round robin</li> <li>• No shared memory caches</li> </ul>
blkmat	32	1-4	
sor	16	1-8	
ugray	8	1-12	
water	20	1-6	
locus	2	1-28	
mp3d	16	1-14	
barnes	16	1-8	

Table 4.3: Experimental parameters for **explicit-switch**.

mean run-length from 7.0 under **switch-on-load** to 8.3 under **explicit-switch** can mainly be attributed to the extra cycle in each run-length from the added switch instruction. This extra cycle is overhead and diminishes performance. The next most troubling application is **ugray**. The grouping factor was only 1.29 and there are still many short run-lengths of just 2, 3, or 4 cycles. These short run-lengths will hamper the efforts of multithreading. The lack of grouping for **sieve** and **blkmat** is unimportant since these applications already had well behaved run-length distributions and moderate or long mean run-lengths.

The experiments used to measure **explicit-switch** execution efficiencies are listed in Table 4.3. There is now a context switch cost of 1 cycle because of the added context switch instructions. We have also increased the number of processors used for **sor**, **water**, **mp3d**, and **barnes**. Under **explicit-switch** they use lower multithreading levels than they did under **switch-on-load**, and thus the surplus threads were used to increase the number of processors. It might seem odd to compare results from **switch-on-load** and **explicit-switch** that use different numbers of processors, nevertheless it is reasonable because there is very little difference in the results when using either the old or new processor numbers. This might seem more obvious if we recall that the multithreading behavior depends on the run-length distributions. For the **switch-on-load** and **explicit-switch** multithreading models the run-lengths usually do not depend on the number of threads used. As long as the number of threads is kept within the limits set by the available parallelism, the number of processors used does not have much impact on the efficiency results obtained. We have chosen to increase the number of processors because it makes the simulations more similar to the way that applications will be run on real machines (with many processors).



Multithreading Efficiency**Explicit-Switch**

(grouping within basic blocks)

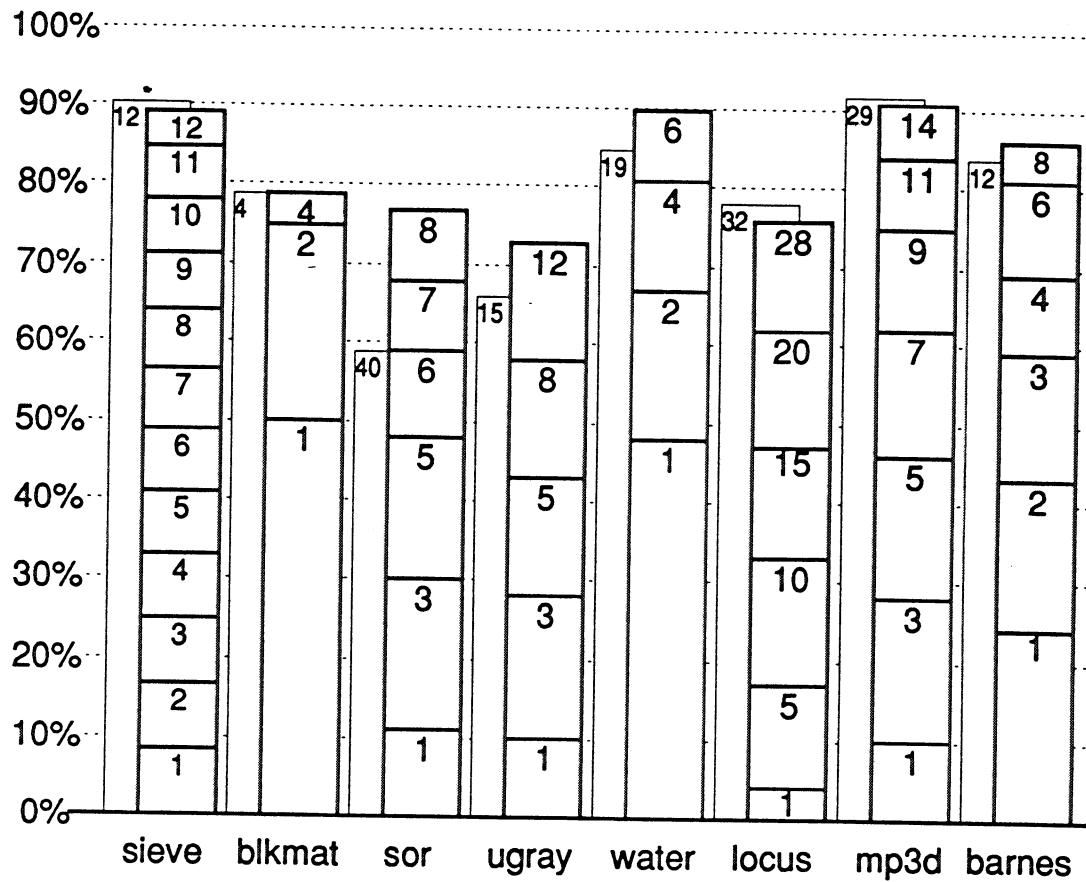


Figure 4.6: Multithreading levels and the efficiencies they achieve under **explicit-switch**. The bars in the foreground show the results for **explicit-switch**, while the bars in the background show the results for **switch-on-load** for comparison.

Figure 4.6 shows the performance results for **explicit-switch**. For each application, two bars are shown. The bar in front shows the efficiency achieved under **explicit-switch** at various multithreading levels. The bar in back shows the previous results from the **switch-on-load** model for comparison. The applications with large grouping factors (*sor*, *water*, and *mp3d*) achieve equal or better performance while using less than half of the multithreading that they needed under **switch-on-load**. *Barnes* and *ugray* had smaller grouping factors, but they were still enough to provide significant decreases in the multithreading levels required. The one disappointment was with *locus*, which had negligible grouping. Its efficiency actually declined because of the overhead of the added switch instructions.

Besides the overhead of 1 cycle for each added switch instruction, there are also software and hardware overheads caused by the grouping of loads. The software overhead arises because the compiler has many optimizations that it must perform. On the MIPS processor, on which these results are based, most floating point operations take more than 1 cycle to execute. These extra cycles are called *delay slots*, and they can be hidden by other instructions, such as loads, if they are available. However by adding the extra constraint of grouping shared memory loads, the compiler has less flexibility in filling these delay slots.

The hardware overhead arises from the increased burstiness of the traffic. Grouped loads are issued into the network in quick succession. The network may not be able to accept a reference every cycle, and thus later references in the group may be delayed. For our simulations, we assume there is sufficient buffering that the processor never has to stall upon issuing a reference. While overheads do exist, their costs are small compared to the benefit of grouping.

In the *water* application, there are some run-lengths of around 5,000 cycles. It might make sense to try some sort of preemptive scheduling mechanism rather than round-robin scheduling in order to better utilize those long run-lengths for hiding the latency of other threads. This will be explored in Chapter 5.

Overall, the results for **explicit-switch** look very promising. Most of the applications can achieve around 80% efficiency with a multithreading level of 10 or less threads per processor. For a large parallel machine, 80% efficiency is commendable, and a multithreading level of 10 is reasonable for on-chip implementation, as will be shown in Chapter 8. *Locus* and *ugray* are the two applications with inadequate performance. In the next section we will argue that better compiler optimization techniques could improve the grouping

<pre> t1 = shared_x; if (t1 &gt; xmax)     xmax = t1; t2 = shared_y; if (t2 &gt; ymax)     ymax = t2; </pre>	<pre> if (shared_flag)     sum += shared_x; </pre>	<pre> while (i &gt; 0) {     sum += shared_x[i];     i--; } </pre>
(a) Code Motion	(b) Speculative Loading	(c) Loop Unrolling

Figure 4.7: Example code fragments with potential for inter-block grouping.

for these applications and raise their performance to be comparable with the rest of the applications.

### 4.3.2 Grouping Beyond Basic Blocks

In the previous section, our code reorganization and grouping of shared loads was done only within basic blocks. Compiler based optimization could do better by looking beyond the scope of a single basic block.

Figure 4.7 shows three simplified examples of situations taken from the *ugray* and *locus* applications that would be amenable to inter-block grouping by a good optimizing compiler. In these examples, shared variable are prefixed with “*shared\_*”, and all other variables are local. In example (a), the loading of *shared\_y* can be moved upward past the conditional test and grouped with the loading of *shared\_x*. In example (b), the loading of *shared\_x* could be moved ahead of the *if* statement and grouped with the loading of *shared\_flag*. This is called a speculative load since it is done on the speculation that the conditional test will be true and that the load will in fact be needed. In example (c), several iterations of the loop could be unrolled and the exposed multiple loads from the *shared\_x* array could then be grouped.

Code motion and loop unrolling are standard optimizations for a good optimizing compiler. Speculative loading, however, is trickier. It might be the case that the conditional test checks the boundary conditions of an array. If the load is moved before the boundary check, it might access off the end of the array and cause an unwarranted memory trap. Rogers and Li[RL92] have proposed a simple mechanism of dealing with this problem by adding a poison bit to each register and taking a trap only upon the use of a poisoned register. A further problem arises if speculative loads are used indiscriminately. If many of

Experiment: explicit-switch with inter-block grouping			
Application	Processors	Multithreading	
sieve	32	1-5	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 1 cycle</li> <li>• Scheduling = round robin</li> <li>• Each thread has a one line cache with a line size of 32 words.</li> </ul>
blkmat	64	1-3	
sor	16	1-9	
ugray	12	1-10	
water	20	1-5	
locus	8	1-7	
mp3d	32	1-11	
barnes	16	1-5	

Table 4.4: Experimental parameters for **explicit-switch** with inter-block grouping.

the loaded values are not actually used, the amount of memory traffic will increase. Because of this, speculative loading will require careful evaluation.

#### 4.3.3 Estimation Experiment

Since we do not have a compiler that does inter-block grouping, we have designed a simple experiment to estimate the amount of potential grouping available. The rationale for this experiment comes from our inspections of **locus** and **ugray**, the applications from which we are most interested in obtaining additional grouping. We noticed that in these two applications many of the inter-block grouping opportunities came either from small stride accesses to an array or from accesses to multiple fields in a structure. Both of these access patterns will have memory addresses that are close together, and therefore if we detect these access patterns, we can predict which references might be groupable.

A cache acts as a detector of spatially and temporally nearby references, and we have used a very small cache as our detection mechanism. We simulated a one line cache associated with each thread, and any hits in the cache were presumed to have been groupable with the preceding reference. The cache line was 32 words, which is long enough to hold many of the structures in **ugray**. We used a one line cache so as to be as conservative as possible in our estimations. We also added instrumentation so that we could verify the results. For **locus** we were in fact conservative, but for **ugray** it is unclear since there were both erroneous groupings and missed opportunities.

The experimental parameters are summarized in Table 4.4. For **ugray** 42% of the

Application	Grouping Factor	Mean Run-Length
sieve	12.31	236.8
blkmat	1.40	208.0
sor	4.99	31.5
ugray	1.94	40.4
water	7.17	311.2
locus	7.03	56.1
mp3d	3.29	33.3
barnes	3.02	129.2

Table 4.5: Grouping and mean run-length estimates if the compiler could do inter-block grouping.

loads hit in this cache, and the grouping factor increased from 1.29 to 1.94. For *locus* the hit rate was 84%, and the grouping factor increased from 1.05 to 7.03. This dramatically shows the potential for compiler based grouping. Table 4.5 shows the grouping estimates for all of the applications, and Figure 4.8 shows the expected multithreading levels based on these higher grouping factors.

This experiment is only a means of estimating the available grouping opportunities. To verify the results, we examined the code of *ugray* and *locus* to see where the cache hits were coming from. In *ugray* we found that 47% of the identified grouping opportunities were valid. These were cases where one field in a structure was examined, and if it met some condition, a second field was loaded and used in a computation. A smart compiler could group these two loads together by speculatively loading the second value in the expectation that the test would succeed and thus the second load would be needed. The other 53% of the grouping opportunities identified by the cache experiment turned out to be cases of coincidental memory allocation such as when two small structures were allocated in the same cache line. While examining the code, however, we also found many interblock grouping opportunities that were missed by the cache. Overall, it remains unclear whether this experiment overestimated or underestimated the interblock grouping potential for *ugray*.

For the *locus* application, a single instruction was found to be responsible for 95% of the cache hits. This turned out to be in a loop that was stepping horizontally through a large two dimensional array. A compiler could easily group these loads by unrolling the loop. In addition, we found similar loops that stepped vertically through the array. The

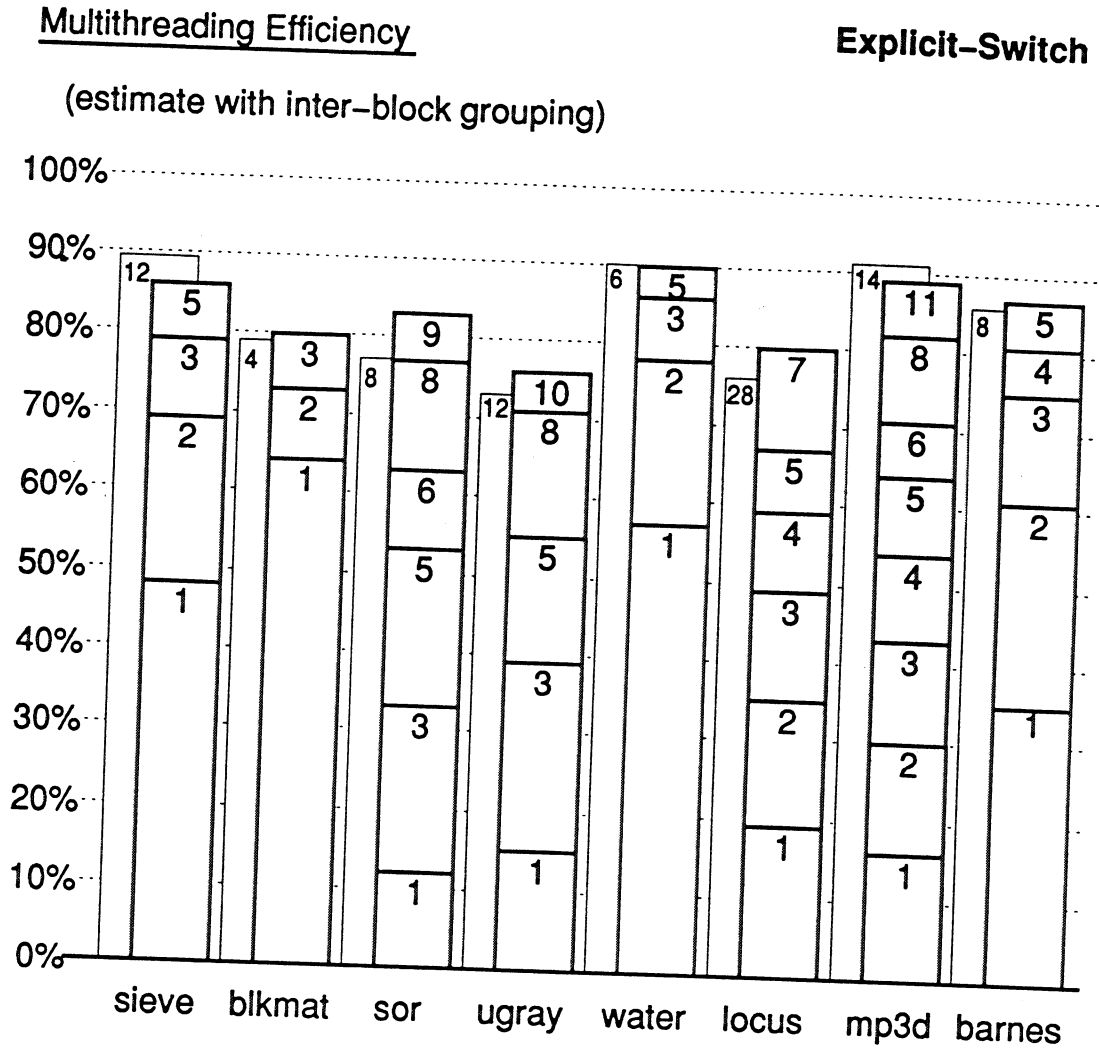


Figure 4.8: Multithreading levels and the efficiencies they achieve under **explicit-switch** with estimated inter-block grouping. The bars in the foreground show the results for **explicit-switch** with inter-block grouping, while the bars in the background show the earlier results for **explicit-switch** without it.

same compiler unrolling technique could group these loads as well, but they were missed by the cache. Thus for `locus` our experiment underestimated the potential for interblock grouping.

For the toy applications (`sieve`, `blkmat`, and `sor`), we have also verified that inter-block grouping is possible. In `sieve` this would involve inter-procedural analysis. In `blkmat` it involves a complex code motion. And in `sor` it involves a simple loop unrolling.

Figure 4.8 shows that with the addition of inter-block grouping, all of the applications can now obtain efficiencies near or above 80% using 10 threads or less per processor. In particular, notice the dramatic improvement of `locus` because of the grouping made possible by loop unrolling.

## 4.4 Conclusions

In this chapter we have shown that multithreading is effective at hiding long latencies to shared memory. The `switch-on-load` model performs poorly for applications that access memory frequently, but the `explicit-switch` model solves this problem by allowing the grouping of independent loads and thereby eliminates many extraneous context switches. For most of our applications grouping within basic blocks is adequate, and for the others there do exist inter-block grouping opportunities. Further research in compiler optimization is needed to fully explore the grouping of accesses.

Simulation results indicate that a multithreading level of 10 threads per processor is adequate for hiding a 200 cycle remote reference latency, and that we can expect efficiencies of 80% or better from a multithreaded parallel machine. This machine provides no hardware caching of shared data, and thus it does not have the complexity of providing cache coherency. The one drawback, which is the subject Chapter 6, is that *all* accesses to shared data are sent across the interconnection network, and thus the network bandwidth requirements will be high.

## Chapter 5

# Multithreading With Caching

Caches are advantageous because they can reduce the number of remote memory references. However, they complicate a parallel machine because they require mechanisms for maintaining coherency among all the caches in the system[ASHH88, CF78, HLRW92, ON90]. In this chapter we first evaluate a system that provides caching but not multithreading, and then evaluate the additional performance improvement possible with **switch-on-miss** and **conditional-switch** multithreading.

### 5.1 Caching

Simulation studies in the literature have reported varying effectiveness for caching on large scale parallel computers. Early DASH results[GHG<sup>+</sup>91] reported miss rates of 20%, 23%, and 34% for three applications (**mp3d**, **pthor**, and **lu**). While O'Krafka[ON90] reported miss rates of 1.3%, 4.8%, and 2.7% on a different set of applications (**ssim**, **verf**, and **genie**). The higher miss rates in the DASH studies were a result of their choosing three benchmarks all with poor reference locality. Later DASH results[LLJ<sup>+</sup>92] with a larger set of benchmarks exhibit much lower miss rates (although they are not explicitly reported).

The experimental setup shown in Table 5.1 was used to measure the cache miss rates for our applications. For these simulations, the cache size is 64K bytes, it is 4 way set associative, and it has a 16 byte line size<sup>1</sup>. Coherency is maintained with the Censier and Feautrier[CF78] invalidation based cache coherency protocol. This protocol maintains a full directory of owners for each data line. It serves as a convenient starting point for

<sup>1</sup>The 16 byte line size is shown in Section 7.2 to be the best choice for minimizing bandwidth needs.



Experiment: caching			
Application	Processors	Multithreading	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.</li> </ul>
sieve	128	1	
blkmat	64	1	
sor	16	1	
ugray	32	1	
water	29	1	
locus	10	1	
mp3d	32	1	
barnes	32	1	

Table 5.1: Experimental parameters for caching without multithreading.

comparison with other research, however this may not be the most cost effective choice because of its large hardware cost[ON90].

For simplicity, we continue to assume a latency of 200 cycles for all network references. In reality, references causing coherency traffic would take longer than other references because of the additional message(s) sent to maintain coherency. For example, a straightforward implementation of invalidations would take two round-trip message times (four messages): the request message from the processor to the memory, the invalidation message from the memory to the invalidation site, the acknowledgment message back to the memory, and finally the response message back to the processor. However, a smarter implementation, such as the DASH protocol[LLG<sup>+</sup>90], can reduce this from four message times to three. Furthermore, in their prototype implementation[LLJ<sup>+</sup>92] they found the extra latency of a reference requiring coherency to be only 30% over that for a normal reference. Our constant latency assumption is thus slightly optimistic.

Table 5.2 shows the simulation results. For most of the applications the miss rates are just a few percent and caching performs well. The two exceptions are mp3d and blkmat.

Mp3d has *low* reuse of data<sup>2</sup>, and its high miss rate is a result of this. It also has a *high* access rate, and thus despite the presence of caches, it still sends a large number of accesses into the network. Without multithreading, it achieves an execution efficiency of only 15%. Gupta *et al.*[GHG<sup>+</sup>91] obtained a processor utilization of 26% for this application on their simulations of the DASH multiprocessor<sup>3</sup>. They assumed a latency of less than half

<sup>2</sup>See Section 2.2.7.

<sup>3</sup>This value was calculated based on their results under release consistency, which is similar to our assumption of weak consistency.

Application	Miss Rate	Efficiency
sieve	0.3%	89%
blkmat	42.5%	59%
sor	0.9%	67%
ugray	3.9%	63%
water	2.9%	73%
locus	1.8%	65%
mp3d	16.0%	15%
barnes	2.3%	78%

• Table 5.2: Average miss rates and execution efficiencies on a machine with 64K byte caches, 200 cycle latency, but no multithreading.

of what we did, and thus our lower efficiency is to be expected.

**Blkmat** also has a high miss rate (42.5%), but because it has a *low* access rate, the resultant access rate is low enough to allow it to achieve 59% efficiency. **Blkmat** has a low access rate because it was programmed to make local copies of shared data. These local copies can be thought of as software caching, and thus the hardware cache is superfluous.

For the other applications, the efficiencies are in the 60% to 70% range. These efficiencies are acceptable for large parallel machines. For instance, executing at 70% efficiency on a 1000 processor machines would give a speedup of 700. We thus conclude that multithreading is not essential when caching is provided.

Gupta *et. al.*[GHG<sup>+</sup>91] obtained quite different results in their studies of cache coherent multiprocessors. They looked at just three applications: **mp3d**, **pthor**, and **lu**. These all have high miss rates and low execution efficiencies, as **mp3d** does in our studies. In our larger application suite, **mp3d** is the exceptional case.

Most of our applications achieve acceptable execution efficiencies, but there is still significant performance loss due to latency. Thus there is an opportunity for multithreading to help push execution efficiencies higher. In the subsequent sections we look at the performance improvements that can be obtained by using multithreading to hide the network latency.

Experiment: run-lengths under switch-on-miss			
Application	Processors	Multithreading	
sieve	128	1	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy</li> <li>• Scheduling = lock-priority + spin-switch<sup>4</sup></li> <li>• Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.</li> </ul>
blkmat	64	3	
sor	16	4	
ugray	32	3	
water	29	3	
locus	10	2	
mp3d	32	11	
barnes	32	2	

Table 5.3: Experimental parameters for measuring run-lengths under switch-on-miss.

## 5.2 Run-Lengths with Caching

Multithreading behaves very different when there is a cache in the system compared to when there is not. Without a cache, under **explicit-switch**, context switches occur at rates ranging from once every 30 cycles, to once every 300 cycle. However with caches, we can now expect most of the previous context switches to be avoided and the mean run-lengths between context switches to rise considerably. Rather than multithreading many threads in order to hide each other's latency, we will need perhaps only two threads per processor so that one can execute while the other is waiting on memory.

The experiments described in Table 5.3 were used to measure the run-length distributions of the applications. The multiple threads on a processor all share the cache, and thus they may interfere with each others' cached data. The miss rates will thus be higher under multithreaded execution than the miss rates listed in Table 5.2 for execution without multithreading<sup>5</sup>. The differing miss rates imply differing run-lengths, and thus the run-length distributions with caching will vary based on the level of multithreading and the size of the caches. We gathered the run-lengths at the multithreading levels that were found to be appropriate for each application.

Figures 5.1 & 5.2 show the run-length distributions under **switch-on-miss**. The

<sup>4</sup>The scheduling policy will be discussed in Section 5.2.1. This policy is non-optimal, but it was chosen because it causes minimum interference with the run-lengths.

<sup>5</sup>Section 7.3 studies the increase in miss rates from multithreading.

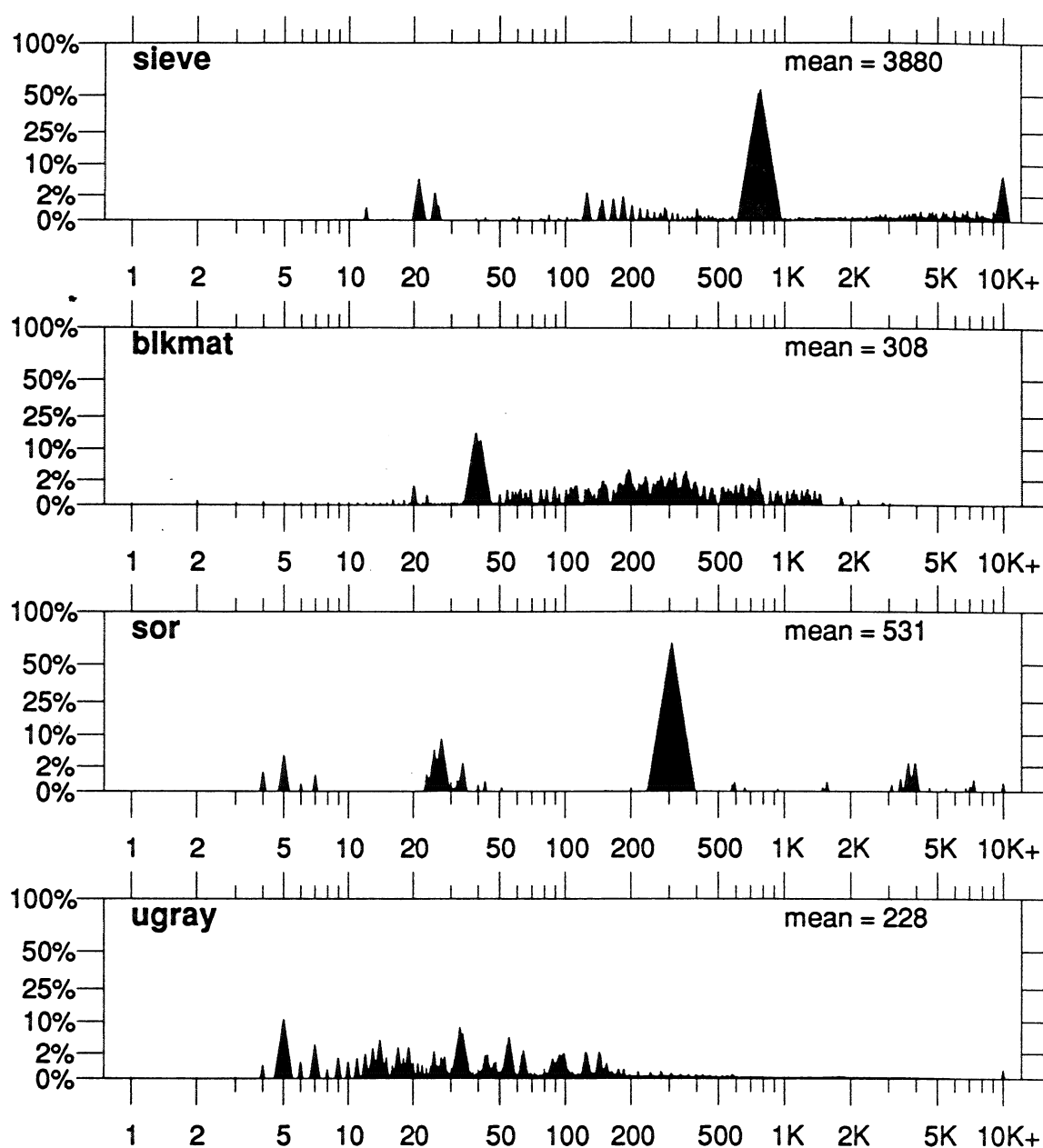


Figure 5.1: Histograms of the run-lengths distributions of the applications running under switch-on-miss.

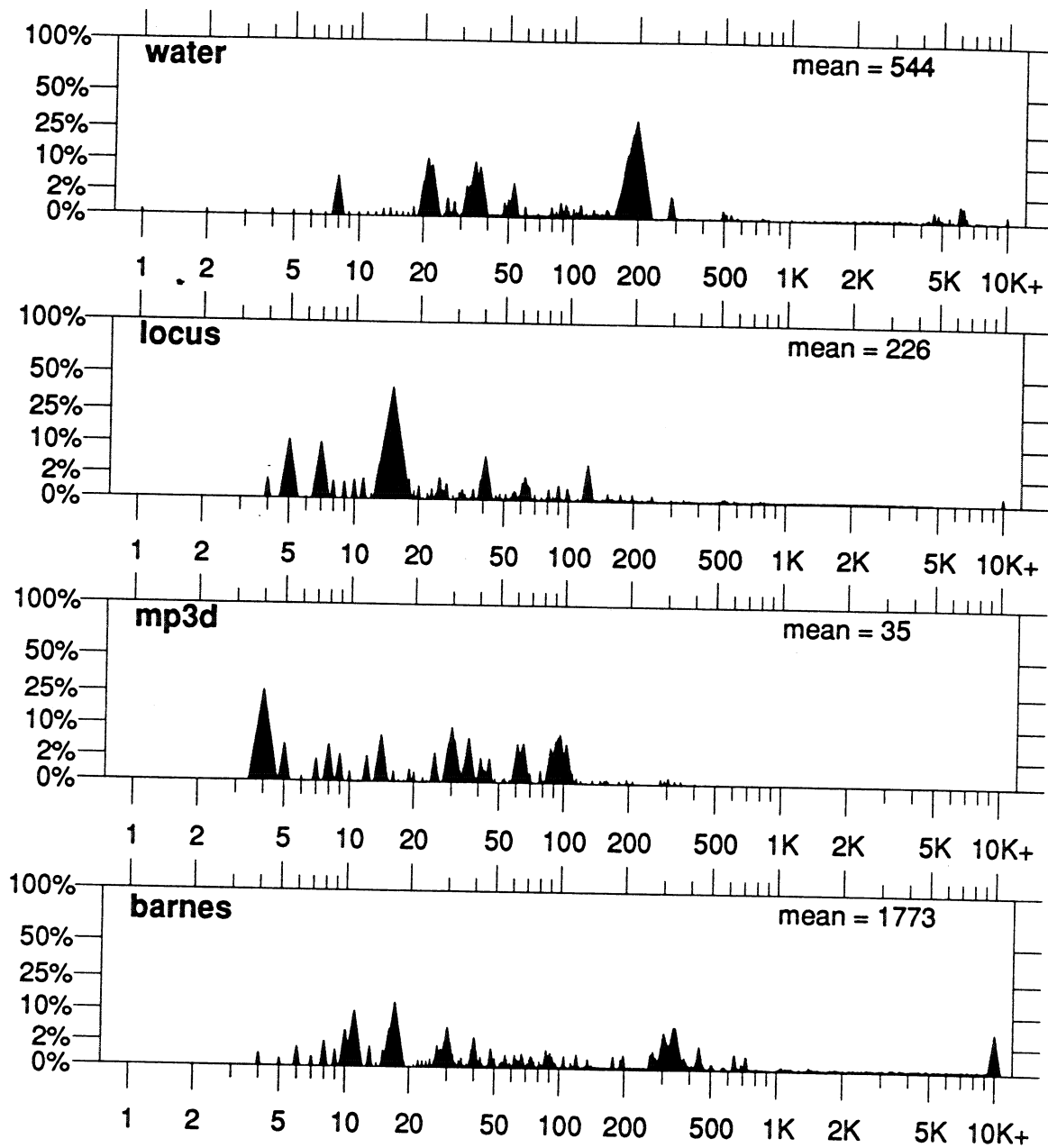


Figure 5.2: Histograms of the run-lengths distributions of the applications running under switch-on-miss.

mean run-lengths are now above 200 cycles (except for mp3d). If the run-length distributions were constant,  $M = 2$  would be sufficient to completely hide the 200 cycle latency, but unfortunately this is not the case. There are still many short run-lengths where misses occur on successive references. The net effect of the cache is that it raises the average run-lengths and spreads out the run-length distributions. When long sequences of accesses hit in the cache, there are run-lengths that last for thousands or even tens of thousands of cycles.

### 5.2.1 Smarter Scheduling

The disparity in run-lengths suggests that a simple round-robin scheduling policy may no longer be the best choice. Long run-lengths can cause problems because they block out other threads from the processor. Consider the following scenarios with two threads on a processor:

**unbalanced scenario:** Thread *A* is executing with long run-lengths taking thousands of cycles, while thread *B* is executing short run-lengths of just 20 cycles. A good scheduling policy should switch out thread *A* whenever thread *B* is ready to run. This allows hiding the latency from as many of *B*'s references as possible.

**locking scenario:** Thread *A* is executing with long run-lengths, while thread *B* is attempting to obtain a lock, do a few critical operations, and release the lock. In order to minimize contention for the critical region, it is important for *B* to hold the lock for as short a time as possible. Ideally thread *A* should be switched out when thread *B* is ready to run, giving *B* priority when it is holding a lock.

**spinning scenario:** If thread *B* is spinning while waiting for some event to happen, it should be given lower priority so that thread *A*, which is doing useful work, can make progress. In fact, it is essential that *A* be given access to the processor since *B* might be waiting on an event that will be caused by *A*.<sup>6</sup>

These scenarios all suggest that context switching must be done more often than just on cache misses. In fact, long run-lengths can be broken into several smaller and more

---

<sup>6</sup>Spinning is a bad idea on a multithreaded processor since the processor will usually have work that can be done by another thread. In Section 7.1.2 we will discuss the implementation of synchronization primitives that do not involve spinning.

uniform run-lengths to help improve their latency hiding capacity. Below are a number of basic scheduling policies that we studied alone and in combination with each other. These policies all switch on cache misses, but also context switch for the additional reasons specified by the policies.

#### Basic Scheduling Policies:

**spin-switch:** Spinning threads are switched out after every shared memory load instruction. This minimizes the number of execution cycles wasted by spinning threads.

**timeout(N):** Threads are forced to switch after they have held the processor for N cycles. (Tried with N ranging from 10 to 200.)

**lock-priority:** Threads holding a lock are given preemptive priority. This allows a thread to execute and exit a critical region as quickly as possible.

**new-priority:** Newly ready threads (those having just received a result from a remote reference) are given preemptive priority. The object is to give priority to those threads that are executing with short run-lengths.

**always-switch:** Threads are context switched after every shared memory load instruction regardless of whether it missed in the cache. This is a simple policy that gives all threads frequent access to the processor.

Table 5.4 shows the execution efficiencies under some of the scheduling policies that we studied. These simulations are for the **switch-on-miss** model, which will be discussed in the next section. We present these scheduling results first because the best scheduling policy found here will be used in the next section for the **switch-on-miss** simulations. Experimental parameters are specified in Table 5.5.

Overall, the best policy that we studied was one that combined **timeout(100)**, **lock-priority**, and **spin-switch**. This was selected as *best* based on averaging the execution efficiencies of all of the applications except **sieve** and **mp3d**. **Sieve** was excluded because it runs well without multithreading, and thus the scheduling policy is irrelevant when there is only one thread on a processor. **Mp3d** was excluded because we will see in Chapter 6 that its performance will likely be constrained by bandwidth rather than latency.

Application	Multithreading	spin-switch	lock-priority + spin-switch	new-priority + spin-switch	always-switch	timeout(100)	timeout(100) + lock-priority + spin-switch
blkmat	M=3	75.3	75.6	76.6	77.3	76.2	76.2
sor	M=4	79.7	79.7	84.4	89.6	88.8	88.8
ugray	M=3	73.6	81.9	89.3	88.3	89.4	89.8
water	M=3	89.1	92.8	92.8	92.1	93.4	93.5
locus	M=3	74.5	75.3	84.8	85.6	85.6	89.5
mp3d	M=11	84.5	84.5	83.7	92.4	84.6	84.6
barnes	M=2	80.0	80.9	82.0	82.5	82.3	82.4
Average (excluding mp3d)		78.7	81.0	85.0	85.9	86.0	86.7

Table 5.4: Execution efficiencies under various scheduling policies.

Experiment: scheduling under switch-on-miss			
Application	Processors	Multithreading	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy</li> <li>• Scheduling = <i>experimental parameter</i></li> <li>• Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.</li> </ul>
blkmat	64	3	
sor	16	4	
ugray	32	3	
water	29	3	
locus	10	3	
mp3d	32	11	
barnes	32	2	

Table 5.5: Experimental parameters for evaluating scheduling policies under switch-on-miss.



Experiment: switch-on-miss			
Application	Processors	Multithreading	
sieve	128	1	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy</li> <li>• Scheduling = timeout(100) + lock-priority + spin-switch</li> <li>• Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.</li> </ul>
blkmat	64	1-4	
sor	16	1-4	
ugray	32	1-4	
water	29	1-3	
locus	10	1-3	
mp3d	32	1-11	
barnes	32	1-3	

Table 5.6: Experimental parameters for **switch-on-miss**.

Many other scheduling policies performed nearly as well as the chosen policy. In fact, simple policies such as **timeout(100)** or **always-switch** performed within 1% of the chosen policy on average. These policies address the three scenarios given above because they limit the interval in which a thread can dominate the processor. We thus conclude that choosing a particular scheduling policy is not critically important and can be based on what the hardware designer finds most convenient.

### 5.3 Switch-On-Miss

The experimental parameters used in our simulations of **switch-on-miss** are shown in Table 5.6. The context switch cost was 3 cycles if caused by a cache miss, but 0 cycles if forced by the scheduler because of some scheduling policy related decision such as a preemption or timeout. The differing context switch times depend upon whether the context switch decision is made early (scheduler) or late (cache miss) in the pipeline. This is explained in Chapter 8.

Figure 5.3 shows the execution efficiencies at various multithreading levels. The bars with  $M = 1$  are the results that were presented in Section 5.1 for caching without multithreading. A few bars, such as  $M = 3$  and  $M = 4$  for **blkmat**, are unlabeled because there was not sufficient room to insert the labels. In all cases, these unlabeled bars correspond to the next sequential multithreading level.

At  $M = 1$ , most of the applications perform in the 60% to 70% efficiency range, and

# Multithreading Efficiency

# Switch-On-Miss

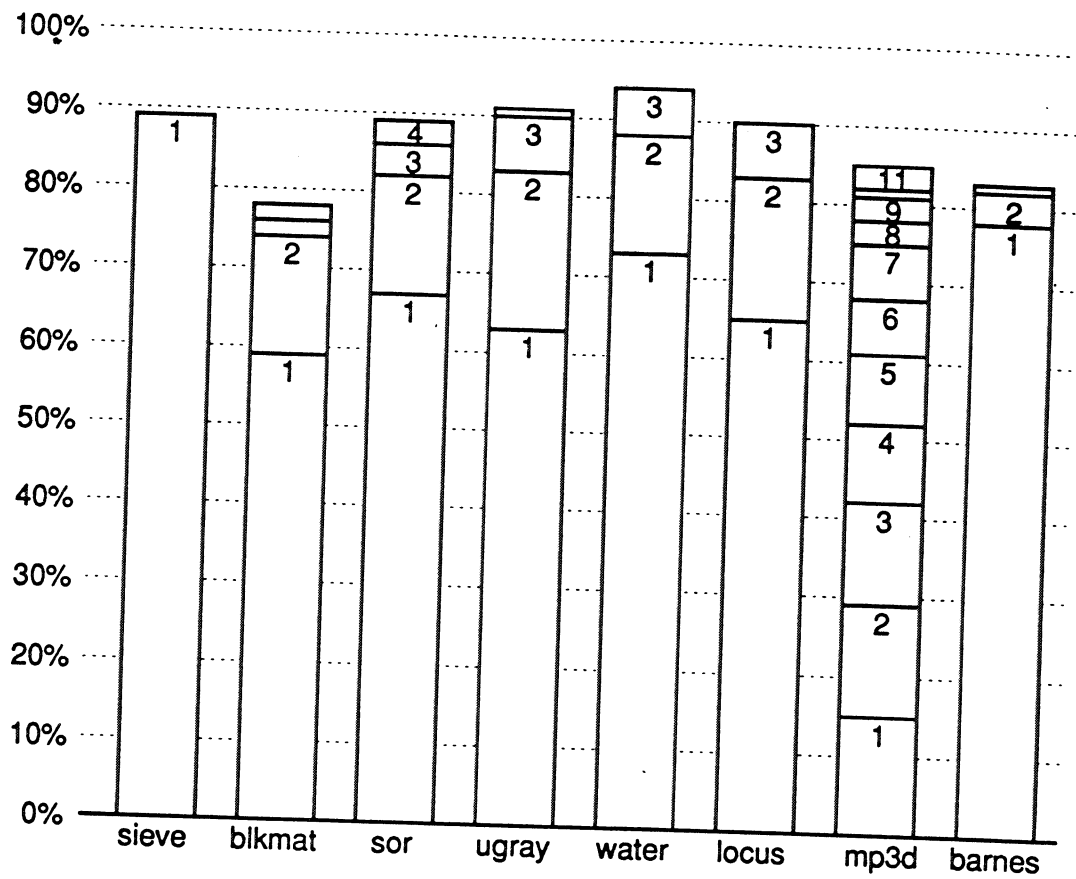


Figure 5.3: Multithreading levels and the efficiencies achieved under **switch-on-miss**.

the addition of multithreading raises the performance to the 80% to 90% range. Expressed in terms of relative performance (multithreaded performance/single threaded performance), multithreading provides a 30% to 40% performance increase for most applications. There are three exceptions. **Sieve** caches extremely well and thus has no use for multithreading. **Barnes** has a large performance loss due to synchronization, which is not helped by multithreading, and **mp3d** caches poorly and thus has room for and achieves much larger performance gains from multithreading.

The number of threads used for **sor** is small because of the sensitivity of its performance to the degree of parallelization. It partitions the 192 by 192 grid into as many square (or rectangular) regions as there are threads. Cache interactions occur just along the edges of these regions because the algorithm accesses only neighboring values in the grid. The cache hit rate is thus strongly affected by the size of the regions. To allow a fair comparison between **switch-on-miss** and **explicit-switch**, we kept the number of processors the same. This lets **switch-on-miss** receive the benefit of requiring fewer threads and thus having larger regions for a given problem size. In the configuration used here ( $P = 16$ ,  $M = 4$ ), **/sor/** runs at 89% efficiency. With more processors and threads ( $P = 64$ ,  $M = 4$ ), and thus finer partitioning, efficiency drops to 75%.

Compared to the results for multithreading without caching (from Chapter 4), the execution efficiencies vary from a few percent worse to 15% better, depending on the application. The big change is that since run-lengths are much longer with caching, not as many threads are needed, and the improvement due to multithreading is much less. For most of the applications, multithreading of 3 threads per processor is adequate to hide the 200 cycle latency.

## 5.4 Conditional-Switch

Grouping was very effective at improving the performance and decreasing the multithreading levels needed under **explicit-switch** compared to **switch-on-load**. We can apply the same idea to a caching system by treating the switch instructions conditionally. Under the **conditional-switch** model, if all of the references proceeding a switch instruction hit in the cache, the switch instruction is ignored, but if any of them miss, then the switch is taken in order to wait for the result(s). The potential benefit is that we can issue more than one reference per thread into the network before waiting for the results to return.

Experiment: conditional-switch			
Application	Processors	Multithreading	
sieve	128	1	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 1 to 3 cycles</li> <li>• Scheduling = timeout(100) + lock-priority + spin-switch</li> <li>• Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.</li> </ul>
blkmats	64	1-4	
sor	16	1-4	
ugray	32	1-4	
water	29	1-3	
locus	10	1-4	
mp3d	32	1-9	
barnes	32	1-3	

Table 5.7: Experimental parameters for **conditional-switch**.

The experimental parameters for **conditional-switch** are shown in Table 5.7. They are the same as those for **switch-on-miss** except for the different multithreading model and different context switch timing assumptions. The context switch cost varies from 1 to 3 cycles depending on when the cache miss occurs relative to the context switch instruction. If the cache miss has already occurred when the switch instruction enters the pipeline, the context switch can be done immediately as it was for **explicit-switch**. Otherwise, the context switch will occur deeper in the pipeline as it did for **switch-on-miss**.

Figure 5.4 shows the performance of the applications under **conditional-switch** multithreading. These simulations were run using grouping only within basic blocks, since we do not have a compiler that can do inter-block grouping. At this level of grouping, all of the applications have equivalent or lower performance than under **switch-on-miss**, except for mp3d. Mp3d is an exception because it does not cache well and thus retains some of the behavior of an uncached system.

The lower performance indicates that grouping is not useful in conjunction with caching. This occurs because grouping is beneficial only when more than one reference is sent into the network before a context switch. If we look at a group of references when the cache is working well, usually all or most of the references will hit in the cache, and only in the rare case when two (or more) references both miss, will grouping be beneficial. In fact, even this overstates the benefit of grouping since sometimes the two accesses may be to elements of the same cache line, and thus the first access would have brought both values in anyways.

Any benefit of grouping is counterbalanced against the extra cost of the added

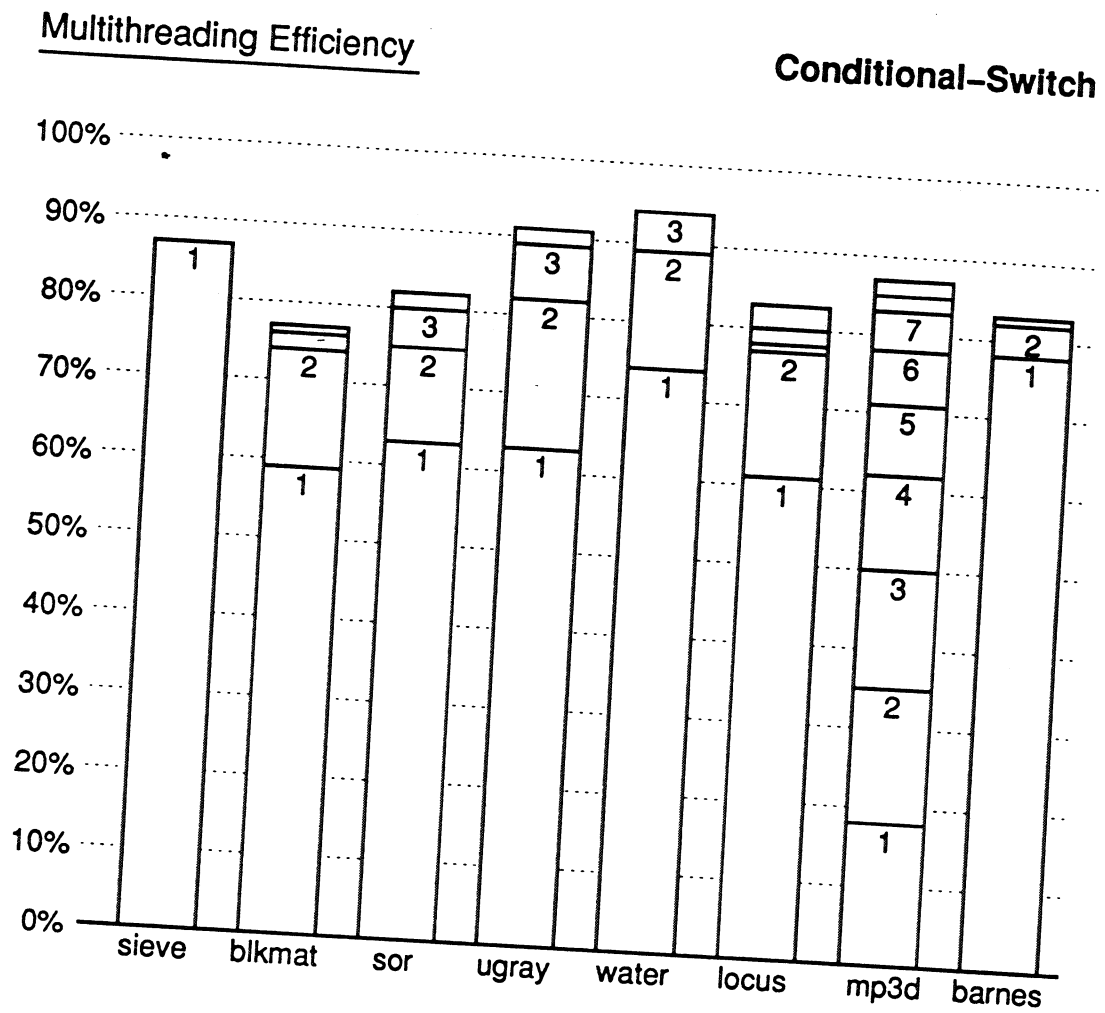


Figure 5.4: Multithreading levels and the efficiencies achieved under conditional-switch.

context switch instructions. These extra instructions take a cycle in the execution stream regardless of whether they are useful or not.

## 5.5 Conclusions

Caching is effective for most of our applications. We observed miss rates ranging from 1% to 4%. These low miss rates mean that threads execute for longer intervals before context switching and thus fewer threads will be needed to hide the latency.

However, sometimes these long execution intervals can cause performance problems by letting one thread hold the processor and thereby block other threads from executing. This can be dealt with by adding a timeout or other mechanism to the scheduling policy.

Our simulations show that a machine without multithreading can obtain efficiencies of 60% to 70% with a latency of 200 cycles, and that a machine with **switch-on-miss** multithreading using 3 threads per processor can boost these efficiencies to 80% to 90%.

Finally, our simulations of the **conditional-switch** model show that grouping is not beneficial in conjunction with caching.

## Chapter 6

# Limited Bandwidth

In the previous chapters we have shown that the long latencies of the communication network can be tolerated by using multithreading techniques. In this chapter we look at the other main characteristic of a communication network: bandwidth.

Where as long latencies are inevitable because of the large number of processors and memories that must be connected together, the bandwidth capacity of a network can be increased by spending more money and adding more wires and/or switches. Unfortunately, as machines grow, the network becomes a larger and larger fraction of the total system hardware. For example on indirect networks such as butterflies and fat-trees[Lei85],  $O(p \log p)$  routing nodes are used to connect  $p$  processors. For direct networks, the number of routing nodes is the same as the number of processors, but if you count pins and wires, the amount of hardware increases for direct networks as well. On a hypercube, the degree of the routing nodes increases as  $O(\log p)$ . On a 2-D mesh, the width of the channels must grow as  $O(\sqrt{p})$  if a fixed bisection bandwidth/processor is to be maintained. The bottom line is that for a large machine, the network will be expensive, and therefore we need to understand and minimize the bandwidth demands put upon it.

In this chapter we present the bandwidth needs of our benchmark application suite under the **explicit-switch** and **switch-on-miss** multithreading models. Our results will show that caching substantially reduces the the network bandwidth needed. We then look more closely at the traffic patterns of **switch-on-miss** systems. The traffic on these systems will be bursty and thus some execution periods will need more network bandwidth than others. We measure this burstiness and use the results along with a performance model to suggest the level of bandwidth that should be supplied by the network.

Experiment: remote memory bandwidth needs of explicit-switch			
Application	Processors	Multithreading	
sieve	32	5	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 1 cycle</li> <li>• Scheduling = round robin</li> <li>• Inter-block grouping estimates as in Section Section-IBG<sup>1</sup></li> </ul>
blkmat	64	3	
sor	16	9	
ugray	12	10	
water	20	5	
locus	8	7	
mp3d	32	11	
barnes	16	5	

Table 6.1: Experimental parameters for measuring the remote memory bandwidth needs of the applications under **explicit-switch**.

## 6.1 Bandwidth Requirement

The bandwidth which an application uses depends upon a number of factors. First, the application may be either computationally or communication intensive. Second, if the machine provides caching, much of the potential traffic may get filtered out by the cache. And third, if the processor is multithreaded, the higher processor utilization and thus higher computational rate will increase the bandwidth requirement.

### 6.1.1 Bandwidth Requirement Without Caching

We measured the bandwidth requirements of the applications by summing the sizes of all messages sent through the network. This gave us the total traffic used by an application. We then normalized this to bits/cycle/processor by dividing the total traffic by the execution time and by the number of processors. We call this the *remote memory bandwidth*.

Table 6.1 lists the simulation parameters. We measured the bandwidths of **explicit-switch** with inter-block grouping, which was the best performing multithreading model (without caching). The bandwidths were computed based on the message sizes shown in Figure 6.1. These messages are used for sending loads and stores and for returning their results. The first field in a message is its destination memory module or processor.

<sup>1</sup>For the inter-block grouping estimates we used a one line cache for each thread. This affected the grouping but not the bandwidth results. The bandwidth was calculated as if all messages were sent into the network and no caching was present.



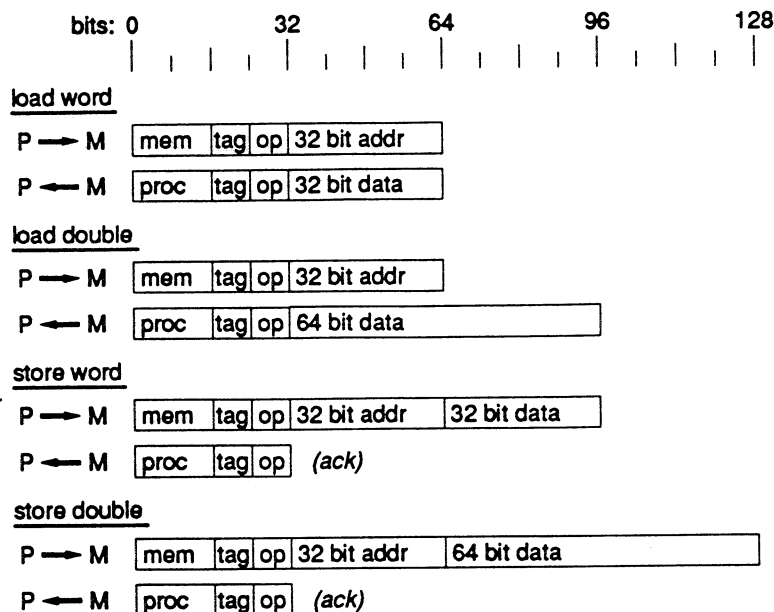


Figure 6.1: Message sizes for remote references to shared memory.

Next is an 8 bit tag field that is used to identify results as they are returned<sup>2</sup>. Then is an 8 bit opcode that specifies the operation type and message size. The last field(s) is either the address being referenced, the data returned, or the address and data for a write.

These messages sizes are at the small end of the spectrum of possible implementations. For instance, we have assumed that the only routing information needed is the number of the destination memory bank or processor, and that the return address can be generated as the message is routed[GGK<sup>+</sup>82]. Also we have used 32 bit addresses, whereas a large parallel machine will likely support a larger address space. To apply our simulation results to a machine using larger messages, our bandwidth results should be scaled up proportionally to the increase in message sizes.

Table 6.2 shows the remote memory bandwidth results. The bandwidths vary considerably by application, and range as high as 30 bits/cycle/proc to as low as 1.44 bits/cycle/proc. For comparison, Table 6.3 shows the bisection bandwidths of proposed and existing machines. These bandwidths are for machines scaled to 1024 processor and are taken from Figure 1.4 in Chapter 1.

At first glance, our measurements of remote memory bandwidth in Table 6.2 may

<sup>2</sup>See Section 8.1.2.

Application	Remote Memory Bandwidth (bits/cycle/proc)
sieve	9.80
blkmat	1.44
sor	30.20
ugray	6.16
water	3.59
locus	15.07
mp3d	19.91
barnes	3.08

Table 6.2: Average remote memory bandwidth needs of applications under **explicit-switch** multithreading.

Machine (P = 1024)	Bisection Bandwidth (bits/op/proc)
TERA	55.0
CM-5	2.5
DASH	1.8
KSR1	1.6

Table 6.3: Bisection bandwidths of proposed and existing machines if scaled to 1024 processors.

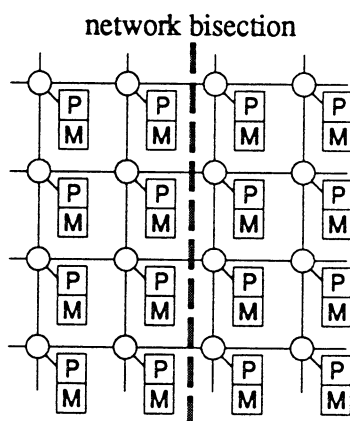


Figure 6.2: A 2-D mesh network and its bisection.

not seem directly comparable to the bisection bandwidths in Table 6.3. The remote memory bandwidth denotes the total network bandwidth used by the applications; whereas bisection bandwidth denotes the bandwidth between two halves of a machines. The amount of bisection bandwidth used depends upon the network and traffic patterns. For example, Figure 6.2 shows a 2-D mesh network with a dashed line drawn across its bisection. If data is laid out so that most traffic is to nearby nodes, then little traffic will cross the bisection. For some applications with regular communication patterns, such as *sor*, good layouts are possible if the network topology matches the communication pattern. However for many applications, the communication patterns and data usage are not predictable, and for some such as *barnes*, there is inherent long distance communication in the algorithm[SHG92].

While good layouts could solve the bandwidth problem, bad layouts could devastate performance by placing several heavily used variables on the same memory module. In this dissertation we do not try to solve the layout problem. Instead, we assume that data is randomly spread across the memory modules in order to avoid bad layouts. With random data layout, half of traffic will cross the network bisection, and thus a bisection bandwidth of  $X$  would be sufficient to support a remote memory bandwidth of  $2X$ . Furthermore, the comparison between bisection bandwidth and remote memory bandwidth is actually closer than this factor of two because networks do not achieve their peak bandwidth capacity.

Thus comparing the remote memory bandwidth needs of our applications (as high as 30 bits/cycle/proc, from Table 6.2) to the bisection bandwidths of most networks (just 1 or 2 bits/operation/proc, from Table 6.3)<sup>3</sup>, we must conclude that these networks will be inadequate for an **explicit-switch** multithreaded system. The only network which can handle these bandwidths is the proposed Tera network, which is in fact a multithreaded system without shared memory caching. This network has perhaps more bandwidth than is actually needed, but was purposefully designed so as not to be bandwidth limited[Smi92]. For the other networks, there must be some mechanism for reducing the bandwidth requirement.

---

<sup>3</sup>We have expressed our bandwidth results in terms of *bits/cycle/proc* because it is an easily understood unit. However, we have expressed the network bandwidth in the more architecturally independent units of *bits/operation/proc*. These units are equivalent for our simulations because we model a pipelined RISC processor that executes at one operation per cycle.

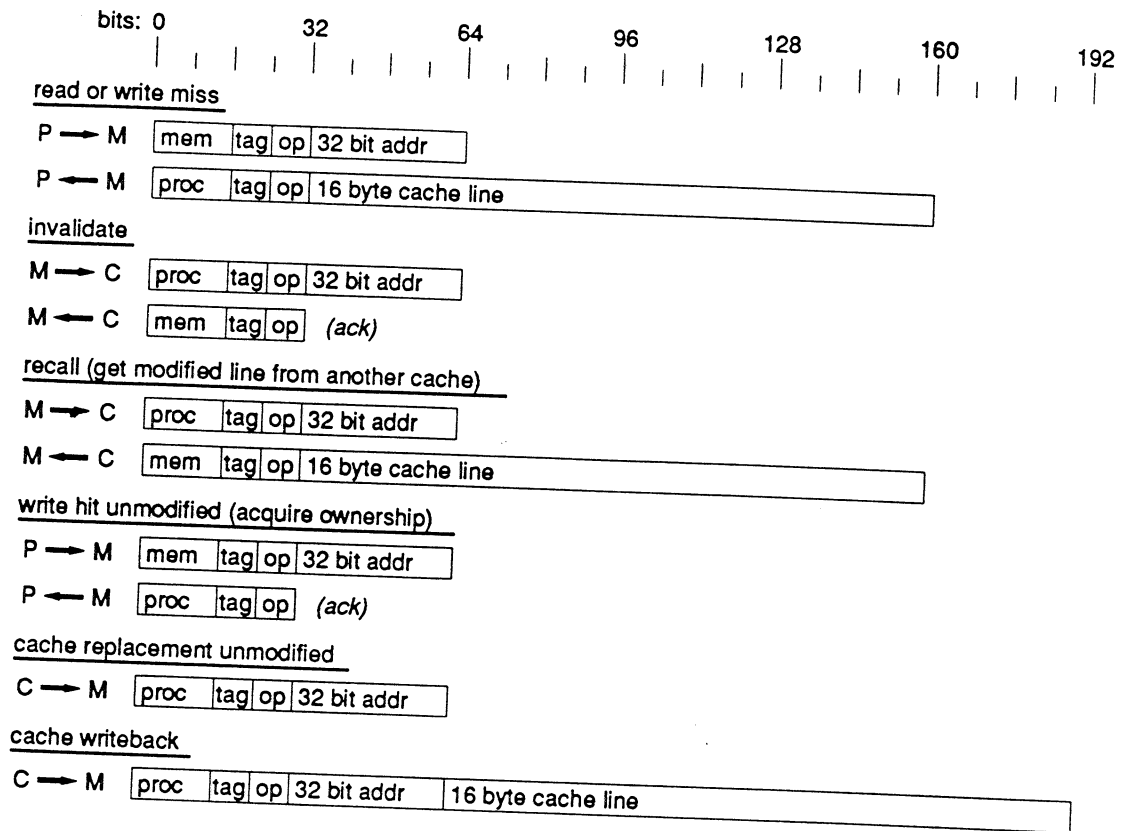


Figure 6.3: Messages used to support coherent caching.

### 6.1.2 Bandwidth Requirement With Caching

In Chapter 5 most applications showed high cache hit rates and thus caching should be effective at reducing the network bandwidth requirements. The bandwidth reductions, however, will not be as high as the hit rates. Caches typically have large line sizes, and these large lines will require more bandwidth to transmit than the single word memory accesses used on systems without caching. Also, extra traffic will be needed to maintain cache coherency.

Figure 6.3 shows the messages and sizes that we have assumed for our simulations of cache coherent systems. These are similar to the messages for a non-caching system (Figure 6.1), but now the memory returns an entire cache line of data rather than just a single or double word. There are also additional messages, such as invalidation and recall messages, that are used to maintain cache coherency.

Table 6.4 shows the simulation parameters, and Table 6.5 shows the bandwidth

Experiment: bandwidth under switch-on-miss			
Application	Processors	Multithreading	
sieve	128	1	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy</li> <li>• Scheduling = timeout(100) + lock-priority + spin-switch</li> <li>• Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.</li> </ul>
blkmat	64	4	
sor	16	4	
ugray	32	4	
water	29	3	
locus	10	3	
mp3d	32	11	
barnes	32	3	

Table 6.4: Experimental parameters for measuring bandwidth under switch-on-miss.

Application	Bandwidth (bits/cycle/proc)
sieve	0.10
blkmat	0.91
sor	1.06
ugray	1.09
water	0.50
locus	1.97
mp3d	14.61
barnes	0.18

Table 6.5: Average remote memory bandwidth needs of applications under switch-on-miss.

results measured for **switch-on-miss**. The bandwidths are under 2.0 bits/cycle/proc for all of the applications except **mp3d**. This compares to bandwidths without caching (see Table 6.2) of as high as 30 bits/cycle/proc. This large reduction in bandwidth is a clear advantage of systems using caching.

**Mp3d** is able to achieve high execution efficiencies when given enough multithreading, but because it has high cache miss rates, its network bandwidth requirement is an order of magnitude higher than that of the other applications. Such a high bandwidth will probably not be provided unless many applications need it, and thus we must conclude that **mp3d** is incompatible with cache coherent multiprocessors.

## 6.2 Squeezing Through a Limited Bandwidth Network

So far we have reported just the average remote memory bandwidth needs of the applications. Simple averages, however, give an optimistic view of the demands on the network because actual traffic will be bursty rather than uniformly spread out over time. Applications pass through different computational phases, some of which do more or less computation or communication than others. Furthermore the communication is not spread evenly across the collection of memory modules. Some memory modules are likely to be subject to higher usage than others (a hot spot) because of many parallel accesses to a single shared variable, or simply due to random coincidence.

The network chosen for an actual machine will necessarily be a compromise. It should have enough bandwidth to satisfy the demands of most application most of the time, but will not be able satisfy all the applications all the time. During periods of inadequate bandwidth, the network will limit the performance of the machine to the rate at which messages can squeeze through the network. A properly chosen network should provide sufficient bandwidth that the overall performance loss to periods of inadequate bandwidth is limited.

In the following sections we take a more detailed look at application behavior in order to determine how bursty the traffic is, how much network bandwidth is really needed, and how useful is special hardware for message combining. We develop a performance model for bursty traffic in a bandwidth constrained network, and apply it to the traffic patterns obtained from simulations of our applications.

Certain aspects of traffic patterns, particularly hot spot references to shared vari-

Experiment: bursty traffic under switch-on-miss			
Application	Processors	Multithreading	
blkmat	256	2	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy</li> <li>• Scheduling = timeout(100) + lock-priority + spin-switch</li> <li>• Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.</li> </ul>
sor	256	4	
ugray	256	3	
barnes	256	2	

Table 6.6: Experimental parameters for bursty traffic under **switch-on-miss**.

Application	Problem Size	Simulation Length (cycles)	Processor Utilization	Cache Miss Rate
blkmat	320 × 320 matrices	1.8 M	93%	35%
sor	768 × 768 grid	first 20 M	94%	1%
ugray	gears — 160 × 512 slice of image	first 20 M	88%	12%
barnes	16,384 bodies	first 20 M	72%	15%

Table 6.7: Increased problem sizes of applications.

ables, will be more pronounced in larger systems with more processors. For this reason we have simulated systems as large as possible. The simulations were for 256 processors and are specified in Table 6.6. The problem sizes were increased so as to provide enough work to allow adequate parallelism.

We selected four of the eight benchmarks for the studies in the remainder of this chapter: **blkmat**, **sor**, **ugray**, and **barnes**. We could not use **locus** or **water** because we did not have large enough inputs for these applications. We rejected **mp3d** because it caches poorly and thus is incompatible with a **switch-on-miss** parallel machine, and we rejected **sieve** because its bandwidth usage is so low that it is not interesting for this study.

Table 6.7 lists the increased problem sizes. Unfortunately, these larger problems (except **blkmat**) took far too long to allow executing them to completion. Thus **sor**, **ugray**, and **barnes** were executed only for the first 20 million cycles. We list the processor uti-

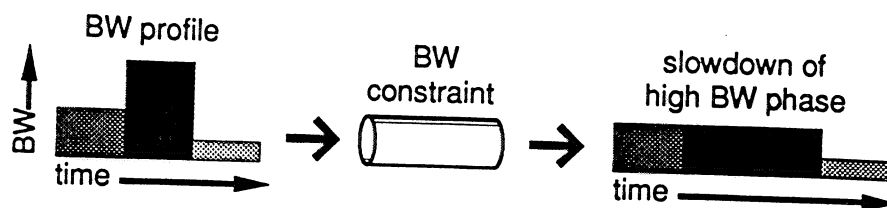


Figure 6.4: High bandwidth phases will slow down as they squeeze through the bandwidth constrained network.

lizations over this period since the execution efficiencies can not be computed unless the applications are run to completion. For **ugray** and **barnes** the cache miss rates have increased because of the larger problem sizes.

### 6.2.1 Remote Memory Bandwidth

In this section we look at how the total remote memory bandwidth needs of the applications vary over time. We will predict the performance that will be achieved on machines with limited bandwidth networks by developing a performance model and then applying it to the bandwidth needs of the applications.

#### Squeeze Performance Model

Figure 6.4 shows the basic idea of our performance model. We start with a bandwidth profile of an application. This is obtained from simulations, and it shows the varying bandwidth needs of the application as a function of time. For most applications the bandwidth will not be uniform. Instead, the applications will have different phases with different bandwidth needs as shown in the figure. The network of a real machine will have a maximum bandwidth capacity, which is represented in the figure by the pipe labeled *BW constraint*. For our performance model, we assume that during phases when an application needs less bandwidth than is available, it will execute at full speed. But during phases when then bandwidth needs exceed the network bandwidth capacity, we assume that execution slows down and makes progress at the rate as which messages squeeze through the network.

Figure 6.5 formally specifies our performance model. This model is much more accurate than simply looking at the average bandwidth over the entire run of the execution, but it is still optimistic. Under some adverse traffic patterns there may be some links of the network or memory modules that are more heavily used than others. These will be



Squeeze Performance Model

$$\text{slowdown} = \frac{\sum_{i=1}^n t_i \max\left(1, \frac{bw_i}{bw_{\text{net}}}\right)}{\sum_{i=1}^n t_i}$$

$n$  = number of phases  
 $t_i$  = duration of phase  $i$   
 $bw_i$  = bandwidth needed in phase  $i$   
 $bw_{\text{net}}$  = bandwidth available

Figure 6.5: Performance model of an application having phases with varying bandwidth needs being executed on a machine with limited network bandwidth.

bottlenecks and could further slow down the execution of the machine. Hopefully, such bottlenecks will be rare when data is spread randomly across the machine as we have assumed.

## Simulation Results

In practice, applications do not exhibit long uniform phases as suggested by the squeeze performance model. The processors are all semi-independent systems which issue occasional messages into the network. Together they form a very bursty system. At some particular point in time, there might be a large burst of new messages resulting from random coincidence. However, this burst will not slow down the machine if on subsequent cycles there is a compensating lull in new traffic. On a small time scale the network and its buffering serve to smooth out the traffic.

To take into account this natural smoothing of the traffic, we have gathered our simulation data over intervals of 100 cycles. Much shorter sample intervals would be pessimistic since they would report bursts of traffic that could be smoothed out by a real network, and likewise much longer sample intervals would be optimistic since they would

smooth over long bursts of traffic that on a real network *would* have a performance impact. We chose the value of 100 cycles because it is half of the expected 200 cycle network latency. This latency will consist of both physical delays and congestion delays with perhaps half of the latency due to each. The congestion delay is caused by the jostling of messages as they move through the network, and this is the period in which small bursts of traffic are smoothed out.

Figure 6.6 shows snippets from the remote memory bandwidth profiles of the applications. These snippets are for 100,000 cycles, whereas the applications were simulated for 20,000,000 cycles. The vertical bars in these graphs each represent a sample interval of 100 cycles. During each interval, the sizes of all messages sent into the network were added together to give a single sample value: the remote memory bandwidth total for that interval. These sample values were then normalized to bits/cycle/proc.

These bandwidth profiles graphically show both the short term and long term burstiness of traffic, but the entire 20,000,000 cycle profiles are too large to be included in their entirety. We therefore present the complete bandwidth profiles by sorting the sample intervals. The sorted profiles are shown in Figure 6.7. Sorting the samples allows drawing a compact graph that more clearly shows the fraction of time the applications operate at various bandwidth levels. For example, when running *sort* the network sits idle (or nearly idle) for 54% of the time. For 12% of the time (the interval of sorted samples from 54% to 66%) the bandwidth is between 0 and 1 bits/cycle/proc, and the rest of the time it is higher. About 10% of the time it is higher than 4 bits/cycle/proc. The other applications exhibit less variance in their bandwidth profiles.

These sorted profiles can be used to easily visualize and compute the performance loss that will result when the applications are run on a machine with a bandwidth constrained network. For *blkmat*, for example, if the network has a bandwidth capacity of 1 bit/cycle/proc, 85% of the program execution has bandwidth needs less than this, and will be unimpeded, but the remaining portion will be slowed down, much of it by about a factor of four.

Table 6.8 shows the precise slowdowns of the applications under various bandwidth limits. These were calculated by applying the squeeze performance model to the bandwidth profiles. *Blkmat*, for instance, under a bandwidth of 1 bit/cycle/proc will slow down by a factor of 1.37.

Using this table, one can choose an appropriate bandwidth level such that the

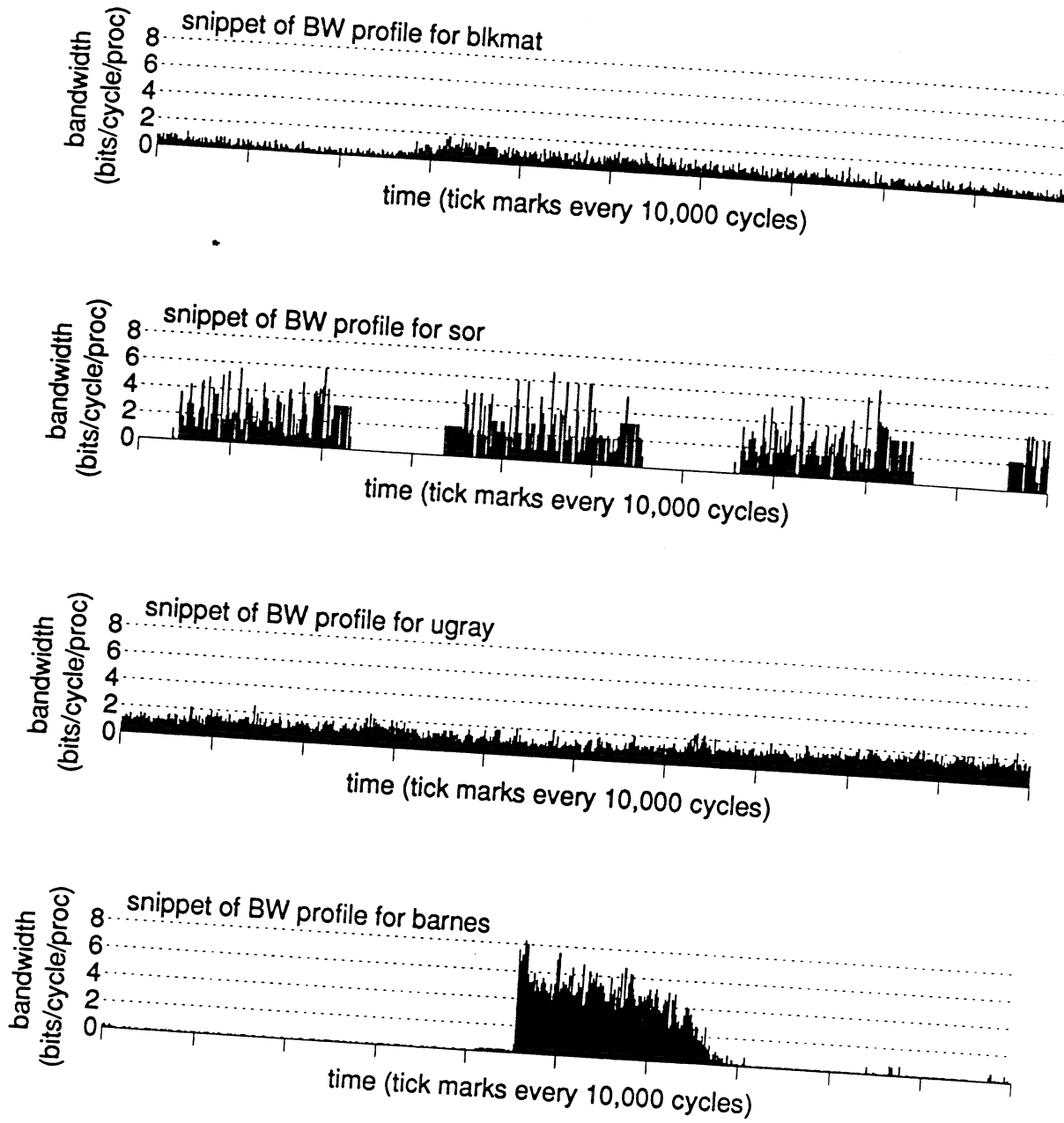


Figure 6.6: Snippets of remote memory bandwidth profiles.

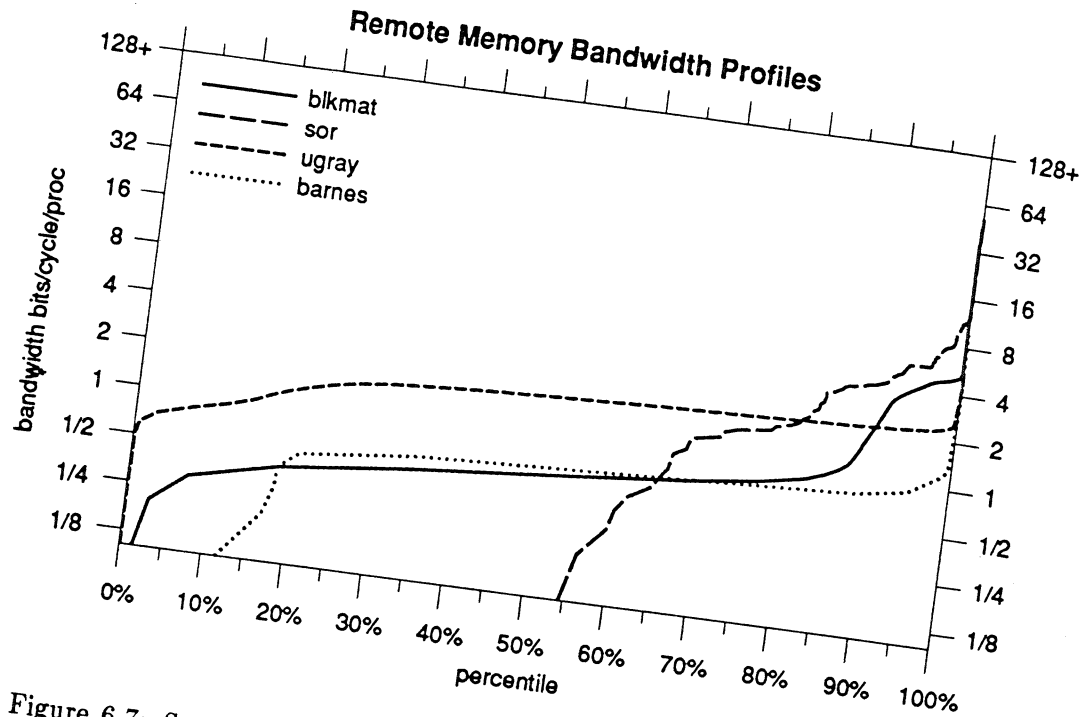


Figure 6.7: Sorted profiles of the applications' remote memory bandwidth usage.

cost of increasing the network bandwidth is warranted by the increase in performance. Clearly it is not worth doubling the size and cost of the network just for a 1% or 2% performance increase. Thus the proper remote memory bandwidth of the network is from 2 to 4 bits/cycle/proc.

### 6.2.2 Hot Spot Memory Modules

The next part of the bandwidth picture concerns how evenly references are spread across the memory modules. If a particular memory module is getting more traffic than other modules, it is dubbed a *hot spot*. A hot spot becomes a problem when traffic directed to it exceeds its capacity. At this point messages to it will be delayed and will back up into the network.

Not only will messages to the hot spot be delayed, but as congestion backs up into the nodes of the network, messages destined for other memories can get blocked as well. The congestion quickly compounds and can spread across the network in a process called

32				
16				
8	1.01	1.01		
4	1.02	1.07		
2	1.13	1.29	1.02	
1	1.37	1.85	1.71	1.01
bw	blkmat	sor	ugray	barnes

Table 6.8: Slowdown factors under various bandwidth limits.

*tree saturation*[PN85].

Hot spots occur for several reasons. We classify hot spots into three categories which we call *location*, *layout*, and *random*. Location hot spots occur when there are many accesses to a particular shared variable, such as a shared counter. They are called location hot spots because they involve a single location. Layout hot spots occur when data is spread across the machine in a regular fashion, but because of the particular access patterns of the applications, some memory modules are much more highly utilized than others. This is analogous to the stride problem for vector computers, and can be diminished by randomly spreading addresses across the memory modules. We have done this randomization of addresses in our simulations to avoid layout hot spots, but this accentuates the third category of hot spots which are due simply to random variation. At various points in time, simply through random chance, some memory modules will be more heavily utilized than others.

### A Pessimistic Performance Model

For a long lasting hot spot, it is clear that because of tree saturation the performance of the entire machine will be slowed down to match the service rate of the hot spot memory. However a short lived hot spot may only slow down those processors directly involved, or possibly because of synchronization and data dependencies, the delays may propagate to other processors as well.

Our model takes a pessimistic outlook and assumes that hot spots will slow down the entire machine. During each sample interval, the most heavily accessed memory module thus determines the machine slowdown for that interval. The total slowdown is calculated by applying the squeeze performance model to the links and traffic directly entering and exiting the hot spot memory module.

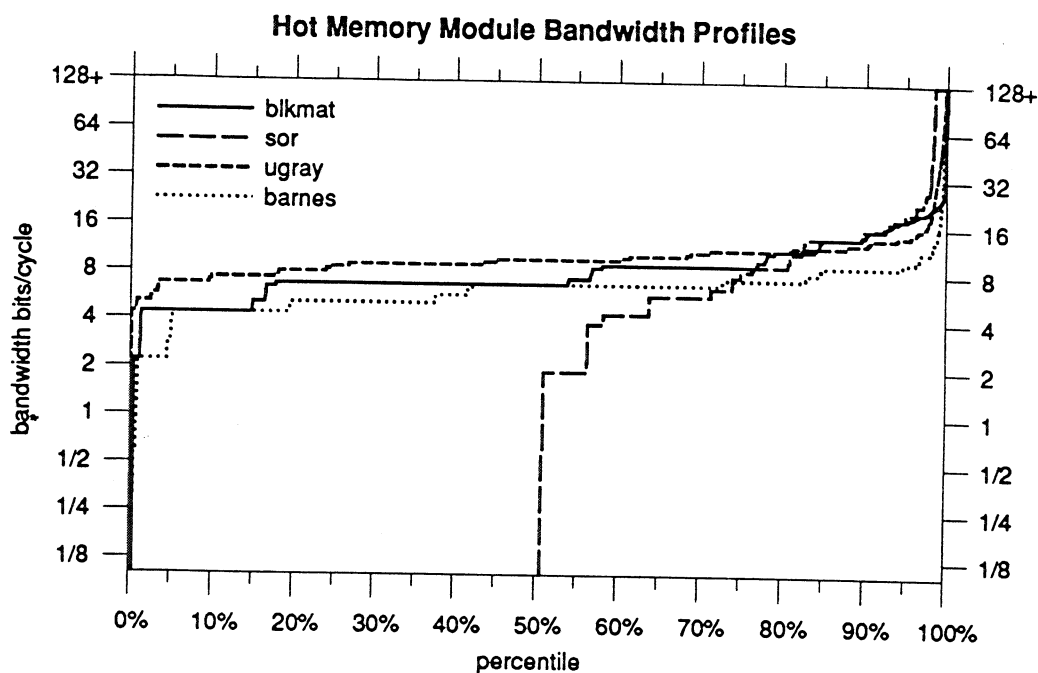


Figure 6.8: Sorted profiles of the applications' hot memory module bandwidth usage.

## Simulation Results

Figure 6.8 shows the simulation results for the hot spot memory module bandwidth. These are typically a factor of 4 to 8 larger than the remote memory bandwidths, and thus hot spots are an important part of the traffic picture.

Table 6.9 shows the application slowdowns computed using our performance model. Based on these results we can say that the network needs a memory module bandwidth of

32	1.01	1.08	1.02	1.01
16	1.03	1.19	1.05	1.03
8	1.25	1.55	1.36	1.10
4	2.23	2.41	2.65	1.77
2	4.44	4.26	5.31	3.49
1	8.88	8.02	10.61	6.97
bw	blkmat	sor	ugray	barnes

Table 6.9: Slowdowns factors based on hot spot memory module bandwidth.

Application	Average BW	Network		Memory Module	
		BW	over design factor	BW	over design factor
blkmat	0.98	4	4	16	16
sor	1.24	4	3	32	26
ugray	1.68	2	1.2	16	10
barnes	0.59	1	1.7	8	14

Table 6.10: Over design factors for the network and memory modules.

16 bits/cycle. We should qualify this by restating that this is a pessimistic model, and that it assumes either rapid onset of tree saturation or propagation of delays to processors not directly involved in the hot spot. We performed additional simulations with a sample interval of 500 cycles in order to gauge how sensitive our results are to this assumption. These simulation, which are probably optimistic, suggest that a memory bandwidth of 8 bits/cycle is adequate. Thus our conclusion is that memory module bandwidth should be from 8 to 16 bits/cycle.

Compared to our results in Section 6.2.1 indicating a remote memory bandwidth of 2 to 4 bits/cycle/proc, the memory module bandwidth is a factor of 4 higher. The direct implication is that networks having higher local bandwidths than bisection bandwidths are advantageous. For instance the fat tree network in the CM-5[LAD<sup>+</sup>92] was designed so that the lowest level of the network has four times the bandwidth of the upper levels. Another example is the networks of M. T. Raghunath[RR93] that provide higher local bandwidths as a means of getting high utilization of the bisection bandwidth. A third example is mesh networks that allows adaptive routing of traffic around the hot spot memories.

Another implication of the higher memory module bandwidths is that the memory modules must be over designed so that they have far more capacity than will be needed on average. We can calculate this over design factor by dividing our performance model's bandwidth suggestions by the average bandwidths actually used by the applications. Table 6.10 shows this calculation (for each application individually) for both the network and the memory modules. The network and memory module bandwidths used in this table were taken from Tables 6.8 & 6.9 at levels that allowed achieving slowdowns  $\leq 1.10$ . The network over design factors are moderate and range from 1.2 to 4. The memory module over design factors are much larger and range from 10 to 26.

Such large over design factors are required to service the hot spots in the memory access patterns. These hot spots arise because of the inevitable non-uniformity of the random message distribution. An analogous problem is the random distribution of  $n$  balls into  $n$  buckets. On average each bucket will receive 1 ball, but the worst case bucket will receive  $\Omega(\log n / \log \log n)$  balls.

### 6.2.3 Location Hot Spots

The elimination and reduction of location hot spots has been the subject of a large body of research[DK92, GGK<sup>+</sup>82, MCS91, PN85, Ran89, YTL86]. These involve either hardware combining of messages or software combining trees. Hardware support is typically for fetch-and-add operations, from which many highly parallel synchronization techniques can be built[GGK<sup>+</sup>82]. Software techniques have been devised for barriers and synchronous reductions[MCS91, YTL86].

Despite the large amount of research on combining techniques, there has been little previous work done on measuring how beneficial combining would be for real applications. This is partly a chicken and egg problem because without hardware support, programmers have little incentive to use fetch-and-add like operations. Although it has been suggested that fetch-and-add is useful even if not combined[MCS91] because it is a simple atomic operation that can be performed quickly at the memories. We have provided such a non-combining fetch-and-add operation in our simulation system, but we have used it only a few times.

In general, ordinary memory requests, such as several reads to a single location, can also be combined. In this section we determine an upper bound on the benefits of hardware combining. Our simulations measure (indirectly) the total number of accesses to each individual memory location during a sample interval. We then use these numbers as our upper bound on combining. In other words, we assume that all references to a single location during a sample interval can be combined. This is optimistic for two reasons. First, combining of different reference types (such as a read, a fetch-and-add, and a write) is very complex and unlikely to ever be implemented. And second, our sample interval of 100 cycles is long enough that in a real network messages will often pass through the routing nodes before their potential combining partners arrive.

Figure 6.9 shows the amount of traffic at the hottest (most heavily used) location



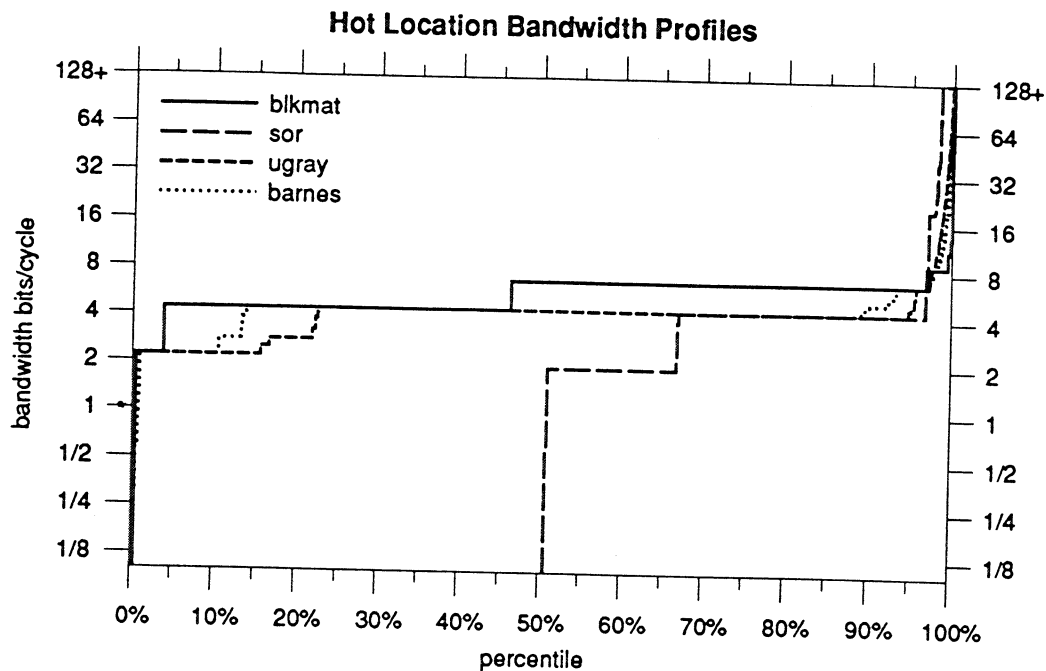


Figure 6.9: Sorted profiles of the applications' hot location bandwidth usage.

during each sample interval. As before, the samples are sorted from lowest to highest. To interpret this data we need to know the correspondence between bandwidth and the actual number of memory operations that occurred. Figure 6.3 showed our assumptions for message sizes. The messages needed to service a simple read miss constitute a total of 224 bits (7 words) of traffic. A read miss is the most common operation in the network and we will use it as our basis for calculation. When normalized to bits/cycle over a 100 cycle sample interval, as we have used, a single read miss uses a bandwidth of 2.24 bits/cycle.

Using *ugray* as an example, the lowest 16% of the intervals show a hot location bandwidth of 2.24 bits/cycle, which is equivalent to the traffic from one read miss message. This means that no location was referenced more than once during these intervals. Next there are some sample intervals that have slightly higher bandwidths. These most likely represent a single access that caused some invalidation traffic. After these, the rest of the intervals all the way up to the 94th percentile show a bandwidth of 4.48 bits/cycle, which equals 2 messages. These two messages might be combined, but such limited combining has little benefit and is not the motivation for combining hardware.

Chances for combining large numbers of messages are rare. For example intervals with 10 or more messages directed to a single location occur during only 2% of the intervals for *sor*, 1.1% of the intervals for *ugray*, and 0.6% of the intervals for *barnes*. For *sor* these hot spots are primarily due to barriers and could be eliminated with either a special purpose barrier network as on the CM-5[LAD<sup>+</sup>92] or perhaps with software barriers[MCS91, YTL86]. Since serious location hot spots are so infrequent, we do not believe that hardware combining is justified.

We should note that two of the applications were modified after the initial run of these simulations. *Sor* showed an unexpected hot spot which was caused by all processors resetting a shared flag when only one processor needed to reset it. This was easily amended.

For *ugray*, there was a problem with contention for a single lock that guarded a free list. This lock was a bottleneck, but had not been noticed before because *ugray* had never previously been run with so many processors. The bottleneck was eliminated by using a parallel free list.

### 6.3 Summary and Implications

For most applications, caching is effective at reducing the network bandwidth requirement. Thus despite the cost and complexity of providing coherent caches, caching is likely to be cost effective because of the reduction in network cost afforded by allowing a skinnier network than on systems without caching.

When burstiness is taken into account, our simulation results and performance model have shown that the networks for large shared memory cache coherent multiprocessors should provide a remote memory bandwidth of from 2 to 4 bits/cycle/proc and a memory module bandwidth of from 8 to 16 bits/cycle. The higher bandwidth requirement at the memory modules is needed to accommodate random hot spot traffic.

While these numbers have been normalized by the processor cycle time and therefore should be widely applicable, an adjustment is needed to account for differences in processors. Specifically, our results are based on the MIPS R3000[Kan89], a pipelined RISC processor operating at one operation per cycle. A superscalar processor operating at two operations per cycle, for example, would thus need to support twice the network bandwidth.

Network	Routing Nodes	Links <sup>4</sup> per Node	Link Width if constrained by:		Total Wires
			Remote Memory Bandwidth	Memory Module Bandwidth	
Ring	256	4	128	4	65536
2-D Torus	256	8	8	2	8192
3-D Torus	256	12	4	1.3	6144
Butterfly <sup>5</sup>	1024	8/2	2	8	36864
Fat-Tree <sup>6</sup>	120	24/4	8	4	15360
Hypercube	256	16	2	1	4096

Table 6.11: Example networks and the link widths they need to provide 4 bits/cycle/proc remote memory bandwidth or 16 bits/cycle/proc memory module bandwidth for a 256 processor machine. The larger link width is selected in order to satisfy both bandwidth requirements.

Table 6.11 lists an assortment of different network topologies and some key parameters. The link widths are based on the bandwidth values suggested by this research. They were calculated so that the networks would provide either a remote memory bandwidth of 4 bits/proc/cycle or a memory module bandwidth of 16 bits/cycle. The larger of these two link widths was then selected and used for calculating the total wire count, which is a rough estimate of network cost. The key observation to be made is that all of the networks except for the butterfly provide more local bandwidth than bisection bandwidth, and this extra local bandwidth is beneficial because it provides the higher bandwidth needed by the hot spot memory modules.

<sup>4</sup>All of these network calculations are based on using pairs of unidirectional links to provide bidirectional routing. For the butterfly and fat-tree networks, two numbers are given for links per node. The first value applies to the routing nodes and the second value applies to the processor and/or memory nodes.

<sup>5</sup>This butterfly network calculation is based on 2 by 2 switches.

<sup>6</sup>This fat-tree calculation is based on 4 by 4 switches and on decreasing by a factor of 2 the total number of links at each higher level in the tree.

## Chapter 7

# Miscellaneous Studies

This chapter is a collection of miscellaneous studies and experiments that were not critical for the main presentation, but nevertheless are important details and issues that arose during this research.

### 7.1 Synchronization

The synchronization methods of current shared memory multiprocessors use spinning when it becomes necessary to wait for a synchronizing event to occur[GT90, MCS91]. For a single threaded processor spinning is acceptable because the processor does not have anything else it could do while waiting. For a multithreaded processor, however, spinning is wasteful because the processor could instead be gainfully executing its other threads. In this section we first review spin based synchronization techniques and then present on-spinning alternatives.

#### 7.1.1 Spin Waiting

##### Atomic Bus Transactions

On bus based shared memory multiprocessors, synchronization is facilitated by an atomic memory instruction such as: **test-and-set**, **swap**, or **compare-and-swap**. These instructions are called atomic because they allow reading and then writing a memory location in an indivisible fashion so that there are no intervening memory accesses.

These atomic instructions are then used to build all higher level synchronization

primitives. For example, a lock on the Sequent Symmetry is just a memory location and its value[GT90]. If the value is 0, the lock is free; if it is 1, the lock is taken. To obtain the lock a processor executes a `swap(addr,1)` instruction. This instruction atomically reads the old value at the address and writes the new value of 1. If the `swap` instruction returns an old value of 0, the lock has been obtained, but if it returns the value 1, then the lock is already taken. In this case, the processor spins: continuously reading the memory location until it changes to 0, at which point the processor retries the `swap` instruction.

When the lock is free, this is very efficient since there is just a single memory reference used to obtain the lock, and later just a single `write(addr,0)` instruction to release the lock. The problem arises when the lock is simultaneously desired by several processors. One processor will obtain the lock; the others will incessantly read the lock location waiting for its release. These reads would saturate the bus to shared memory except that most are filtered out by the caches. After the first read, the value is cached and subsequent reads simply spin on the value in the cache. When eventually the processor holding the lock relinquishes it by writing a 0 to the lock location, it triggers invalidations (or updates) of all the caches. At this point the remaining contenders all race to obtain the lock.

With  $n$  processors contending for the lock, each of these races involves  $O(n)$  bus accesses. By the time a group of  $n$  processors all get their turn with the lock, there will have been  $O(n^2)$  bus accesses to the lock location. There are a number of more sophisticated lock implementations which try to reduce this traffic[And90, GT90, MCS91]. The best of these reduce bus traffic to  $O(n)$  by building a software queue in which the waiting processors can all spin on local flags. Releasing a lock involves clearing the flag belonging to the next processor in line.

All of these lock implementations use spinning. The difference between them is simply how much of the traffic is local or global.

### Fetch-And-Add

The NYU Ultracomputer project[GGK+82] proposed an innovative synchronization instruction: `fetch-and-add (f&add)`. `F&add(addr, value)` reads the specified memory address, returns the contents as the operation's result, and then adds the specified value and stores the sum back into the addressed location.

A simple use of `f&add` is to have each of  $n$  processors execute `f&add(X, 1)`. If initially  $X = 0$ , the values returned will all be unique and go from 0 to  $n - 1$ . These values might then be used to select unique tasks for each processor in a parallel computation.

The power of `f&add` comes from the fact that multiple `f&add` messages can be combined in a tree like fashion as they proceed through a butterfly network. If combining works well, a group of simultaneous `f&add`'s will get fully combined so that only a single message actually reaches the memory module. The result message returned from the memory module will then be split apart and the correct return values computed as the message(s) return back down the combining tree. This can be done in such a way that the responses are the same as if the `f&add`'s had been performed sequentially. Combining allows congestion at the memories due to hot spots to be eliminated (although our results in Section 6.2.3 suggest that such congestion is rare in our applications).

Gottlieb[GGK<sup>+</sup>82] shows that data structures such as parallel queues can be designed using `f&add` and combining so that there is no serial bottleneck. In other words, hundreds of processor can simultaneously insert and remove entries from the queue without ever entering a critical section (where only a single processor has exclusive access to the internal queue data structures).

Although `f&add` based synchronization routines (along with combining) can eliminate memory hot spots (if they were a problem), these routines still use spinning in order to wait for a synchronizing event: such as the release of a lock or the insertion of an entry in a queue. Thus from the point of view of a multithreaded processor, cycles would still be wasted by threads waiting on synchronization operations.

### Full/Empty Bits

Synchronization on the HEP[Kow85] was done through full/empty bits associated with each memory location. For example, there was a write instruction that would set the full bit when it wrote a value, and there was a read instruction that would check that the full bit was set before reading the value. If the location was empty, the reading thread would wait until an appropriate write occurred. These full/empty bits allowed very fast and fine-grained synchronization.

Since HEP was a multithreaded processor, spinning the processor while waiting would be wasteful. Instead there was a separate unit called the Storage Function Unit that

did the spinning.

### 7.1.2 Non-Spinning Synchronization

All of the spin based synchronization techniques reviewed in the previous section treat synchronization as simply a “fancy” memory reference: a memory reference that also does a test or addition. The desired synchronization constructs are then built by using these “fancy” memory references in conjunction with spinning,

In fact, synchronization is really something more than this. Obtaining a lock is really a request to be notified when the lock becomes available, and releasing a lock should instigate the notification of a waiting thread. Likewise, checking in at a barrier is a request to be notified when all cohorts have also checked in, and the final thread to check in should instigate this notification.

Fundamentally, synchronization mechanisms such as locks and barriers involve a message from a thread to a synchronization agent(Sync-Agent) and then a subsequent message from the Sync-Agent to the appropriate synchronizing thread. For example, a lock is requested by sending a message to the Sync-Agent. If the lock is available, the Sync-Agent immediately sends a reply message granting the lock to the thread and marks the lock as taken. If another request for the lock then arrives while the lock is taken, the Sync-Agent queues this request, and later when the lock is eventually released, the Sync-Agent sends a message granting the lock to the first queued requester. This message based synchronization provides a mechanism for eliminating spinning. Queue-On-Sync-Bit[GVW89] and Queue-On-Lock-Bit[Gus92] are similar mechanisms for building queues of waiting processors, but they were designed without consideration of multithreading and thus use local spinning.

In this section we present a collection of synchronization operations which can be built upon this understanding of synchronization as message passing and therefore directly implementing it as such. In fact, in a large shared memory multiprocessors, memory references are also messages as well: a message is sent to a memory modules to read a value, and a reply message is returned to deliver the result. Because everything is actually implemented with messages, our message based synchronization will turn out to have a simple implementation that fits naturally into the design of our multithreaded processor.

## Messages

What was described above as the synchronization agent, is really just the interface to a memory module. For normal memory operations the interface receives messages, does the memory operation, and sends replies. It also must send and receive any messages needed to maintain cache coherency. For synchronization operations, the memory interface sends the messages dictated by those operations.

A synchronization variable is a memory location just like any other variable, the only difference is that it is accessed via synchronization instructions rather than normal memory access instructions.

We have provided the following synchronization operations in our simulator:

**lock:** A message is sent to request a lock, and when the lock becomes available, a message is returned granting the lock. An unlock message is used to release a lock.

**barrier:** Each thread sends a message when it is ready to check in at a barrier, and when all threads have checked in, barrier completion messages are returned to each of the threads<sup>1</sup>.

**fetch-and-add:** This is the same as for the NYU Ultracomputer except that there is no combining. A **f&add** message is sent to the memory, the addition is performed there, and the reply message returns the fetched value.

**wait:** This is similar to the full/empty bits on the HEP except that it is just a synchronization: there is no associated data transfer. A message is sent to wait for a specified flag to be set. If the flag is already set, a completion message is immediately returned, otherwise the response occurs later when the flag is set. There are also messages for setting and resetting the flag.

The messages formats and their sizes are shown in Figure 7.1. These messages are the same format as the messages used for regular memory operations, except that no data is associated with most of the synchronization operations. They are therefore compact messages; most are just one or two words long.

---

<sup>1</sup>In our general simulations we have not modeled the limited network bandwidth and the serialization at the memory module of the many barrier messages. In Section 6.2 we addressed the network bandwidth limits and found that barriers are infrequent enough in our applications that the occasional congestion they cause has only a minor performance impact. Later in this section we will mention some congestion free alternatives.



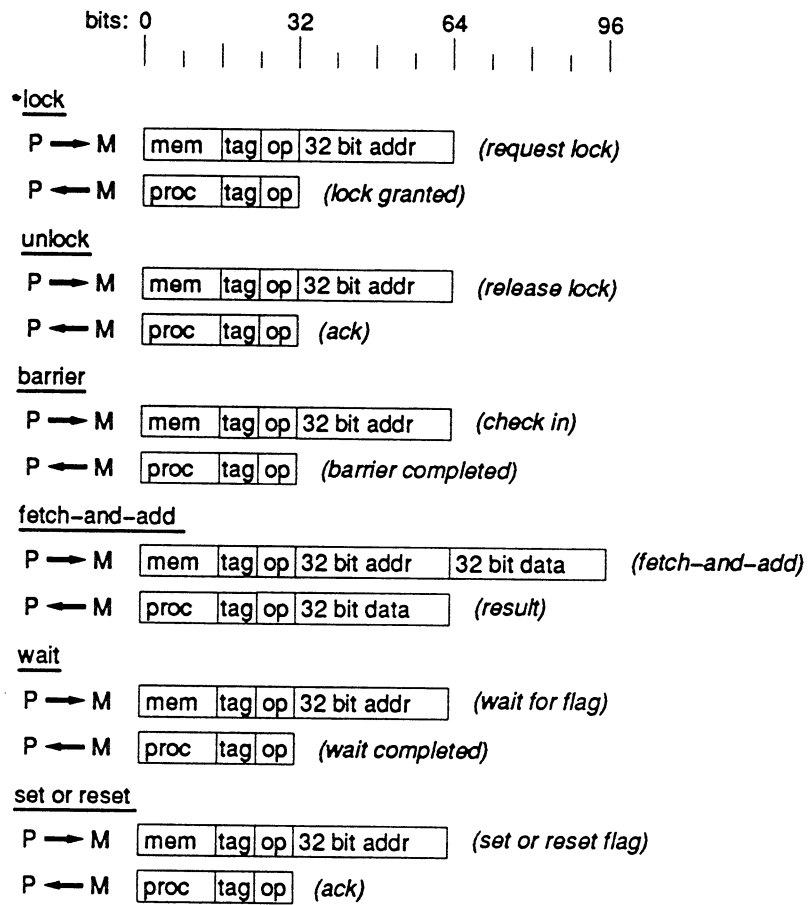


Figure 7.1: Messages for synchronization operations.

Application	cycles/sync/processor			
	lock	barrier	f&add	wait
sieve	—	240K	480K	—
blkmat	17K	1.7M	—	—
sor	—	24K	—	—
ugray	115K	—	8K	—
water	5K	2.9M	—	—
locus	19K	78M	—	—
mp3d	—	117K	7K	—
barnes	41K	1.7M	340K	176K

Table 7.1: Frequency of use of various synchronization operations. These are expressed in terms of the average number of cycles between synchronization operations of a particular type, on a per processor basis. K = one thousand cycles. M = one million cycles.

### Frequency of Use

Table 7.1 shows the frequency of use of these synchronization operations under **switch-on-miss**<sup>2</sup>. Locks and barriers were used by most of the applications, **f&add**'s by some, and waits were used only by **barnes**. In fact, the original codes for **mp3d** and **barnes** did not explicitly use **f&add** or wait operations. We inserted these operations when we replaced spin based equivalents.

Spinning was eliminated from most of the applications, but **ugray** and **locus** still have spinning in their parallel task queues. Most of the spinning occurs for these applications as the computation is completing and threads are waiting for any new tasks to be made available. Because of this spinning, it was useful to have **spin-switch** as part of the scheduling policy (see Section 5.2.1).

All of these applications are coarse grained in their use of synchronization. The heaviest synchronization uses are: locks in **water** are obtained at rates of once every 5,000 cycles, barriers in **sor** are used once every 24,000 cycles, and **f&add**'s in **mp3d** are used once every 7,000 cycles. Since the use of **f&add**'s (and/or any other type of synchronization) is infrequent, this corroborates our results in Section 6.2.3 that suggested that message combining in the network would provide little benefit.

<sup>2</sup>The experimental parameters are the same as in Table 6.4.

## Implementation

Our message based synchronization fits nicely with the design of a multithreaded processor. When a thread issues a normal memory access into the network, that thread is then context switched out and it waits for the response message. The exact same things occur for synchronization. For example a thread requests a lock by sending a message into the network, and then it waits for the response message saying that the lock has been granted. The only difference between the two cases, as far as the processor is concerned, is that memory references have fairly uniform latencies (a few hundred cycles), but locks have varied latencies (depending upon how many other threads are also waiting for the lock and how long they each hold it).

Because issuing a memory reference or a synchronization reference both cause descheduling of a thread, and their responses both cause its reactivation, both types of references are treated identically by the processor. However because synchronization references may have long latencies, strictly round robin scheduling is inappropriate. The processor should not wait on a specific thread if other threads are ready to execute. First-in-first-out (FIFO) is a simple scheduling policy, and the policies looked at in Section 5.2.1 were all just slight modifications to FIFO.

At the memory modules, memory references and synchronization references *do* have to be treated differently. There will be some sort of memory interface unit whose job is to extract messages from the network, process them, and send the replies. For normal memory references, this processing will be simple: read the requested memory block (line) and send it. However for some memory references, there will be coherency operations, such as invalidations, that also must be performed. These will involve the sending and receiving of additional messages. For synchronization operations, the memory interface unit must check if the synchronization event has occurred and queue waiting requests.

The main complexity in implementing these synchronization operations is maintaining the queue of waiting processes. Figure 7.2 shows one possible implementation<sup>3</sup>. A lock variable is just the address of a small structure in shared memory. The first location contains a flag which says if the lock is free or busy (taken). The next two locations contain pointers to the head and tail of a list of waiting threads. This three word lock structure should reside on a single memory module. Also each memory module will need a local array

---

<sup>3</sup>The acknowledgement responses to unlock messages were not shown in this figure in order to make it simpler.

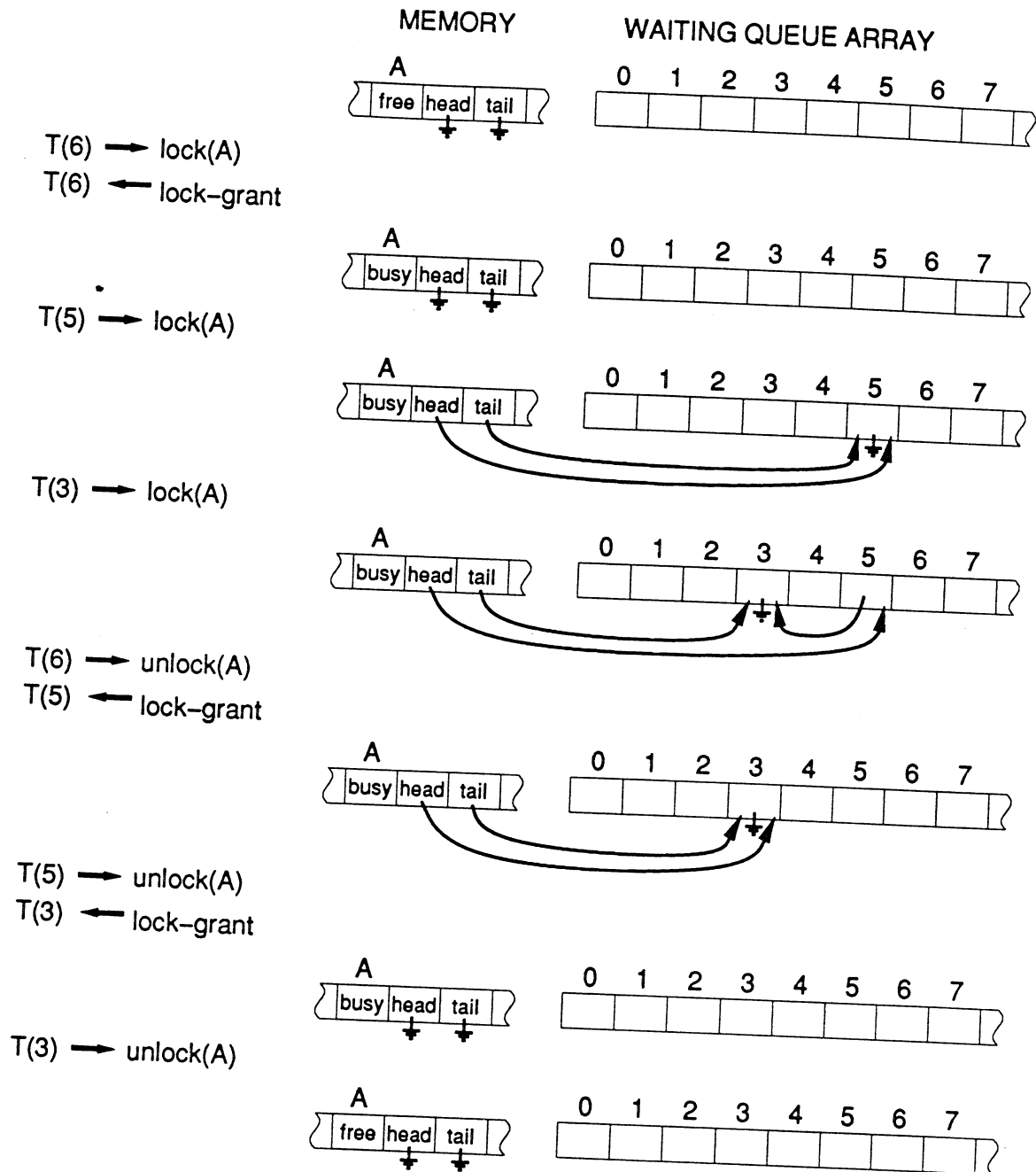


Figure 7.2: Operation of the waiting queue for a lock.

in which it can build waiting queues as linked lists. This array contains one location for each thread, but only one such array will be needed per memory module, regardless of the number of locks.

In the example in Figure 7.2, *A* is a lock variable which is initially set to free. Thread 6 is the first to request the lock and is therefore immediately granted the lock. Threads 5 and 3 then request the lock and are queued up in FIFO order. Next thread 6 releases the lock, which causes the lock to be granted to the thread at the head of the waiting list: thread 5. When thread 5 releases the lock, the lock is granted to thread 3. And finally when thread 3 releases the lock, the lock returns to the free state.

Since the three word lock structure and the waiting queue array all reside on the same memory module, they can easily be updated atomically. The memory interface unit simply performs all its operations for the lock variable before servicing the next incoming message.

If an application has multiple lock variables, they should be spread across the memory modules to avoid unnecessary hot spots. Since the number of lock variables is unlimited, there may be several lock variables on each memory module. These can all share a single waiting queue array since the threads in each waiting queue will be distinct. This is true because it is only possible for a thread to be waiting on one lock at a time, and thus it could never be on more than one queue. It is perfectly valid, however, for a thread to obtain nested locks. They just must be obtained one at a time.

Because of the complexity of synchronization (and cache coherency) it is likely that the memory interface will be some sort of programmable device. In fact, each memory module will likely be connected to one of the processors, and the synchronization and cache coherency might be handled by a quick interrupt to the processor. The Wisconsin Wind Tunnel[HLRW92] uses a CM-5 in this fashion; the processor manages the cache coherency protocol. We suggest letting the processor handle synchronization operations in the spirit of active messages[vECGS92]. This allows the synchronization operations to be changed and supplemented rather than being designed into the machine.

Besides locks, other synchronization operations such as barriers, waits, and queues can also be built with messages so that no spinning is required. In addition to eliminating spinning, performing complex synchronization operations at the memory modules has the advantage of being faster than building them out of several simpler synchronization operations that each involve a network traversal delay[BR90].

If contention is a problem, many synchronization operations can be implemented in a distributed fashion. Barriers can be implemented with software combining trees[YTL86] or a potentially faster technique called the dissemination barrier[MCS91]. Index distribution and work queues can be implemented using the low contention techniques of Herlihy[HLS92]. These are based on counting networks and do not use spinning if there is an atomic memory operation such as fetch-and-add.

## 7.2 Line Size for Minimizing Bandwidth

In this section we study the affect of cache lines sizes on the network bandwidth needs of our applications. A large cache line has the potential of decreasing network bandwidth because the headers for routing and specifying a memory access are of fixed size (see Figure 6.3), but the data payload varies with the cache line size. With a larger line, the fraction of bandwidth used for data is higher.

In practice, a larger line size might might actually increase the bandwidth requirement for several reasons. First, the requesting processor may not use all of the locations in a cache line. These unused locations use bandwidth when they are brought across the network, but do not otherwise affect performance. Second, a large line size increases the likely hood of false sharing. This is the case where two different processors access different parts of a single cache line, and the cache line is ping-ponged back and forth between the processors even though they are not actually sharing any variables. Third, larger line sizes imply fewer total lines in the cache and thus increase the probability that useful data will get replaced with unwanted data (cache pollution).

Figure 7.3 shows the bandwidth usage of the applications when run with cache line sizes ranging from 8 to 128 bytes. The experimental parameters are shown in Table 7.2. These experiments did not use multithreading. However, we expect that the same relative relationship between bandwidth and line size will continue to hold for multithreaded systems.

Most of the applications have increasing bandwidth needs with larger line sizes. The best choices for minimizing bandwidth usage are either 8, 16, or 32 byte cache lines, depending upon the application. A 16 byte line size is the best overall choice, and that is the value we have used throughout the studies in this thesis.

O’Krafka[O’K92] also looked at traffic as a function of cache line size. He studied

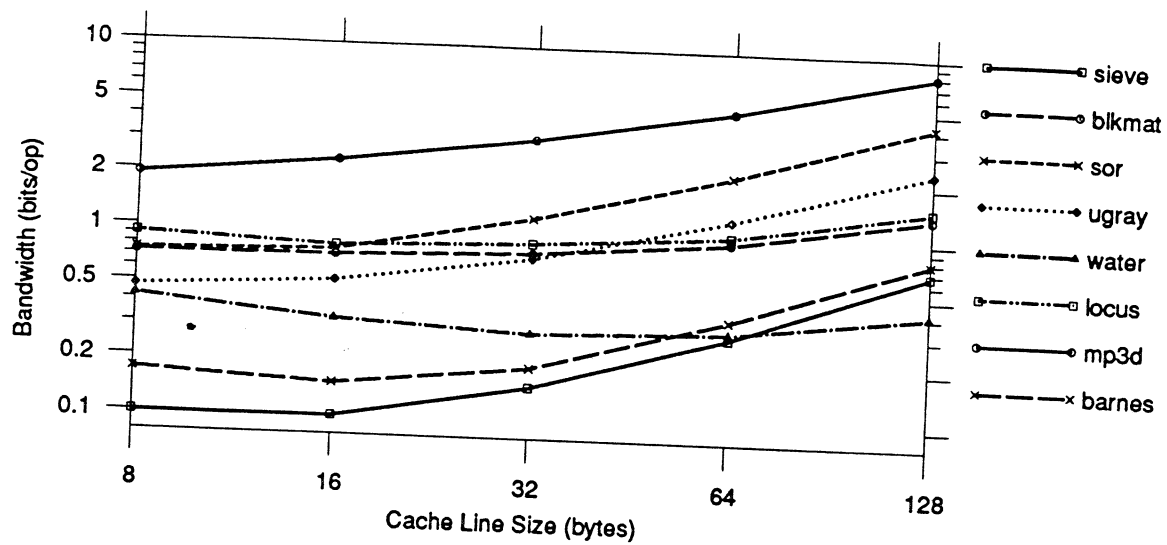


Figure 7.3: Bandwidth as a function of line size.

Experiment: Bandwidth versus Line Size			
Application	Processors	Multithreading	
sieve	128	1	<ul style="list-style-type: none"> <li>• Latency = 200 cycles</li> <li>• Each processor has a 64K byte cache with 4 way set associativity. The line size was a parameter of the experiment.</li> </ul>
blkmat	64	1	
sor	16	1	
ugray	32	1	
water	29	1	
locus	10	1	
mp3d	32	1	
barnes	32	1	

Table 7.2: Experimental parameters for measurements of bandwidth versus line size.

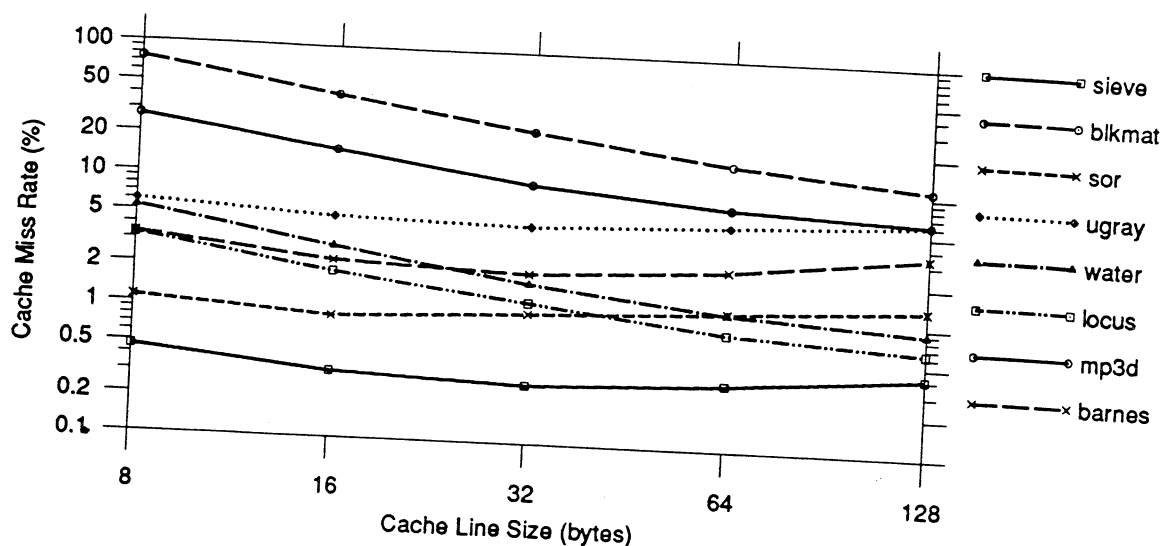


Figure 7.4: Miss rates as a function of line size.

three application: **verf**, **ugray**, and **locus**, and three different lines size: 4 bytes, 16 bytes, and 64 bytes. Traffic was minimized for **verf** with a 4 byte line, for **ugray** with a 16 byte line, and **locus** with a 64 byte line. He also compared invalidation based protocols with update based and competitive protocols. He found that update and competitive protocols “are only appropriate for multiprocessors with 100 or fewer processors”, and we therefore limited our studies to invalidation based protocols.

The rising bandwidth usage with large line sizes suggests that the problems of unused data, false sharing, and cache pollution do occur for many of the applications. Figure 7.4 shows the miss rates as a function of line size. These would be expected to decrease with larger line size, and do for some of the applications (**blkmat**, **mp3d**, **water**, and **locus**). The diminishing rates of decrease with larger line sizes suggest some of the data fetched in large lines is unused. The other applications (**ugray**, **barnes**, **sor**, and **sieve**) shows increasing miss rates with large line sizes, which suggest false sharing and cache pollution are occurring for these applications.

For those configurations showing decreasing miss rates, the performance should be higher since fewer stalls on remote references are occurring. Figure 7.5 shows the processor utilizations as a function of line size and confirms this inverse relationship between miss rates and performance. However, since miss rates do not diminish with increased line size



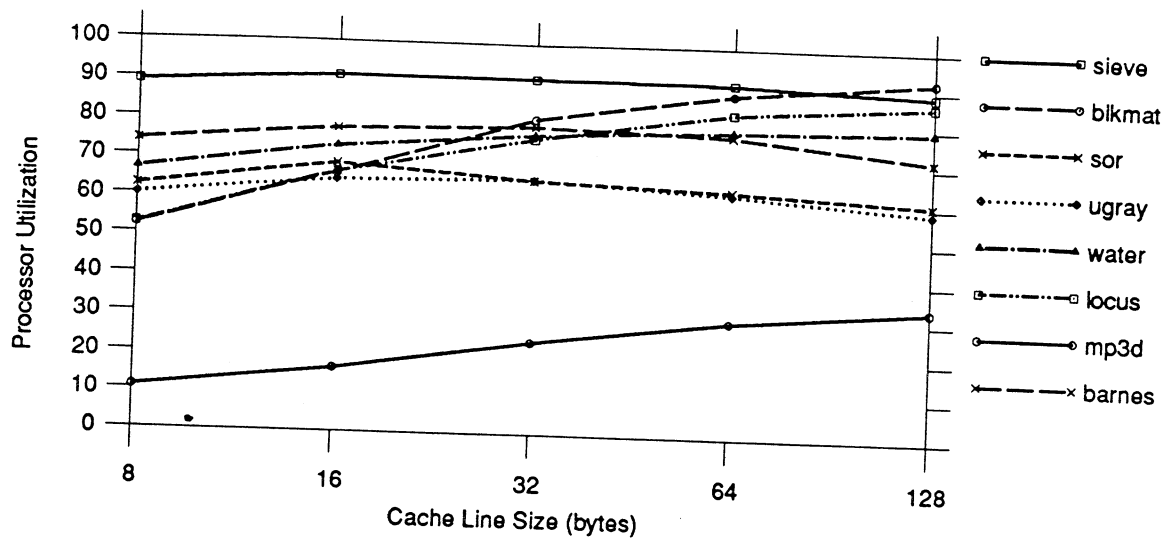


Figure 7.5: Processor utilization as a function of line size.

for all of the applications, performance increases for some applications but declines for others.

The goal of this section is to determine an appropriate line size for minimizing the network bandwidth requirement. Simply looking at the bandwidth as a function of line size can be misleading because bandwidth usage might be low simply because the processors were idle most of the time. Figure 7.6 adjusts for this possibility by normalizing the bandwidth results from Figure 7.3 by the processor utilization results from Figure 7.5. (This sort of normalization actually occurs when multithreading is used to increase performance to a higher level.) The earlier conclusion that a 16 bytes line size is the best overall choice for minimizing bandwidth still holds, although several applications now have slightly lower bandwidth requirements with a 32 byte line size.

Network bandwidth is not the sole issue in choosing a line size. Another major factor is the overhead in terms of cache tag space and directory storage that is needed to locate lines and manage coherency. The number of lines is directly proportional to the amount of this storage, and thus doubling the lines size halves the amount of tag and directory storage. Because of the large amount of directory storage, this may be a more important factor than bandwidth. The KSR1[Ken92], for example, has chosen a 128 byte line size in order to limit the size of its distributed directories.

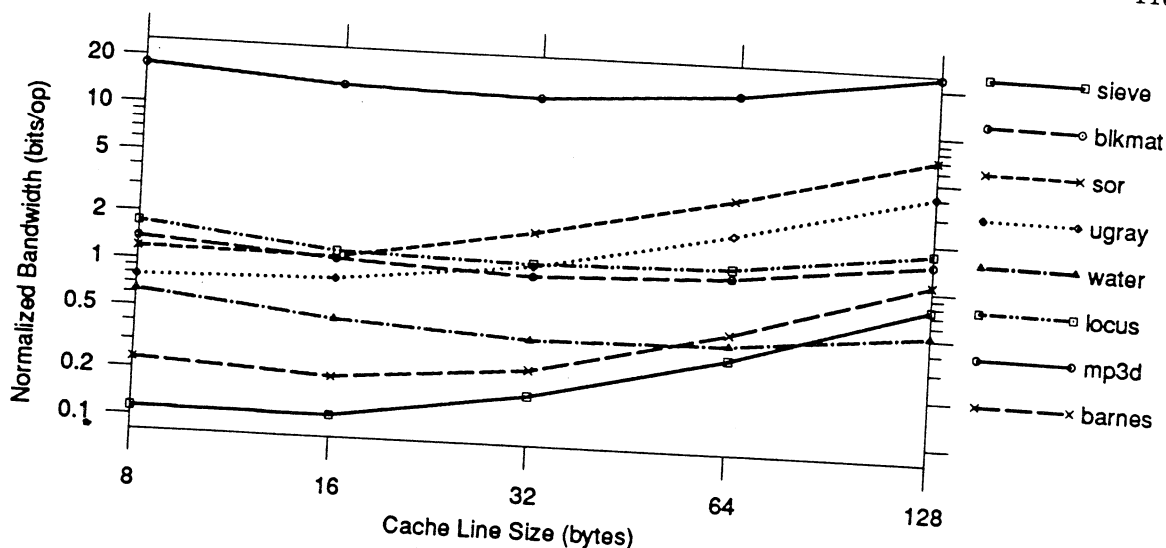


Figure 7.6: Bandwidth (normalized to 100% processor utilization) as a function of line size.

### 7.3 Cache Degradation due to Multithreading

One concern with multithreaded processors is that they may have poor cache behavior if the threads on a processor compete for space in the cache and knock out data used by each other [ALKK90, CGL92, SBCvE90, WG89]. The interaction between threads as they access the cache can be destructive or constructive. The destructive case occurs when a thread knocks out data that *would have been used* by other threads. The constructive case occurs when one thread brings in data that *is latter used* by other threads.

We think of the processor as conceptually having three caches: one for instructions, one for local data, and one for shared data. Throughout this research we have only considered the shared data cache. We expect that the instruction cache should have constructive interaction between the threads since they are all from a single SPMD (Single Program Multiple Data) program. For many of the applications the threads are synchronized so that they execute in same region of code simultaneously. For local data, the cache interactions will be destructive, and caches should clearly be larger in order to accommodate the multiple threads. Figure 7.7 shows the cache miss rates for the shared data cache as a function of multithreading<sup>4</sup>.

<sup>4</sup>The simulation parameters are the same as in Table 5.6.

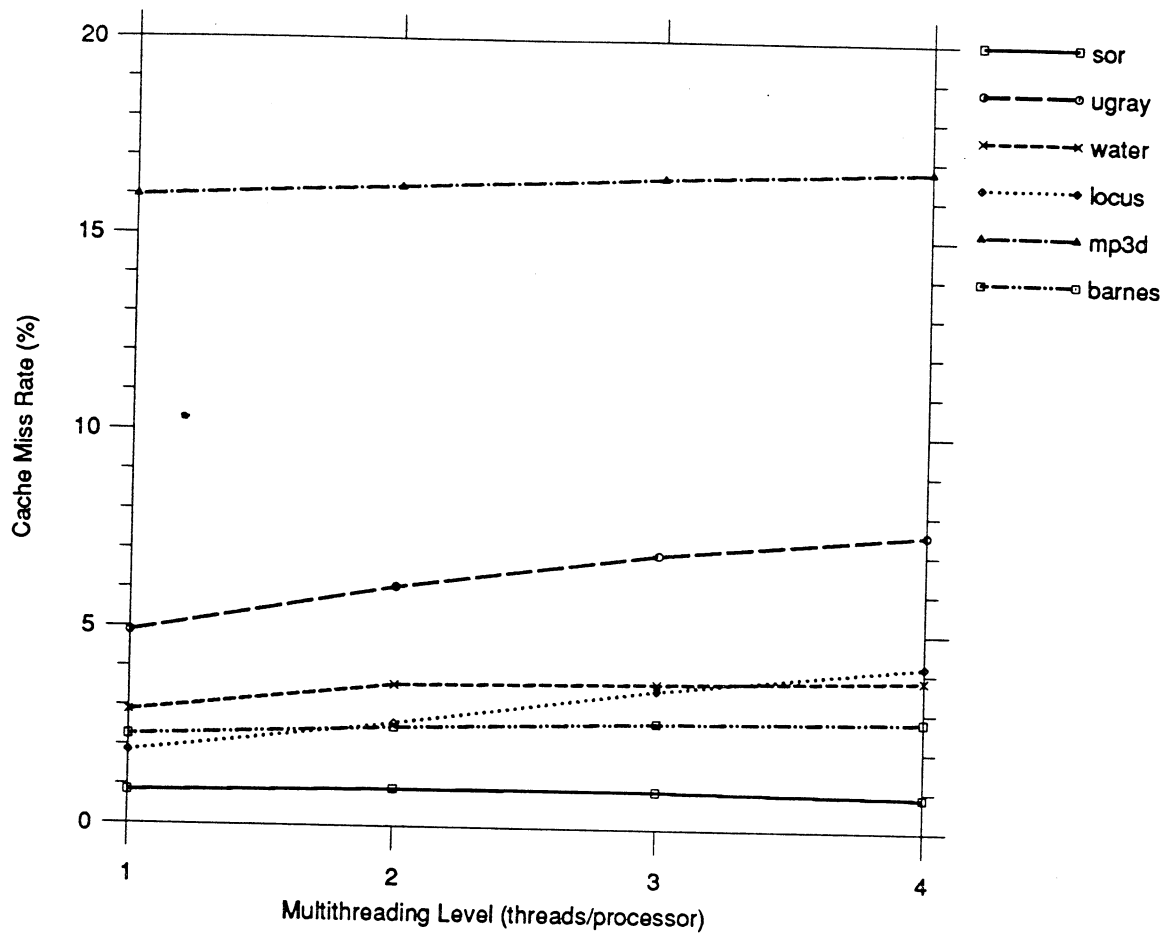


Figure 7.7: Cache miss rates as a function of multithreading.

Generally the miss rates increase slightly with multithreading. Saavedra-Barrera [SBCvE90] modeled this increase in miss rates based on the assumption that with multithreading of  $N$ , the cache would behave as if it were partitioned into  $N$  subsets, each of size  $1/N$ th of the original cache. Then based on an analytic model of cache behavior, he derived an expression for the miss ratio as a function of multithreading:  $m(N) = m(1)N^K$ , where  $m(N)$  is the miss ratio with multithreading  $N$ , and  $K$  is a constant that depends on the behavior of the applications. This model does fit the behavior of many of the applications, but the values of  $K$  vary widely.

A major factor in this variation in cache behavior is whether the threads interact constructively or destructively. For some applications this depends on the order in which

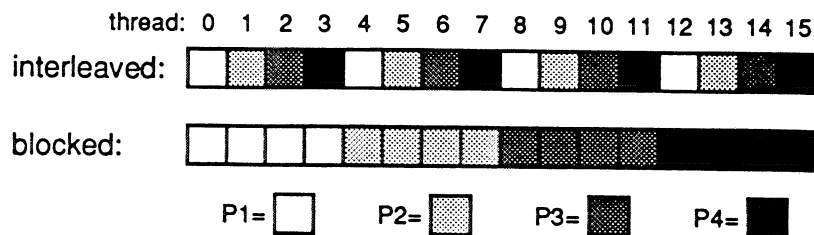


Figure 7.8: Assignment of threads to processors.

threads are assigned to processors. For example, the `sor` application works on a two dimensional array that is partitioned into rectangular blocks with one block per thread (see Section 2.2.3). Interaction between threads occurs for the data along the edges of these blocks, and if abutting blocks (threads) are assigned to the same processor, the common edges will interact constructively in the cache.

Figure 7.8 shows two possible orders for assigning threads to processors. The blocked order is better for `sor`, `barnes`, and `blkmat` because they partition the problem by thread id numbers; and thus threads working on neighboring regions are assigned to the same processor. Blocked ordering is actually worse for `water` because the particles are not isotropically distributed; and thus blocked assignment aggravates load imbalance on the processors. For the other applications, work scheduling is less structured and the thread ordering is unimportant. For the simulations in this thesis, we used blocked assignment for all of the applications except for `water`, for which we used interleaved assignment.

## 7.4 Longer Latencies

The final issue which we address in this chapter is what happens when latencies are longer than 200 cycles. We should expect that *without* multithreading the increased latencies will mean longer waits for remote references and thus lower efficiencies. *With* multithreading, we should expect that more threads will be needed to hide the longer latencies.

Figures 7.9 & 7.10 show simulation results for the applications at latencies of 200, 500, and 1000 cycles<sup>5</sup>. As expected, single-threaded efficiencies drop with increased

<sup>5</sup>The simulation parameters are the same as in Table 5.6 except that the latency has been varied and higher multithreading levels were used for some applications.

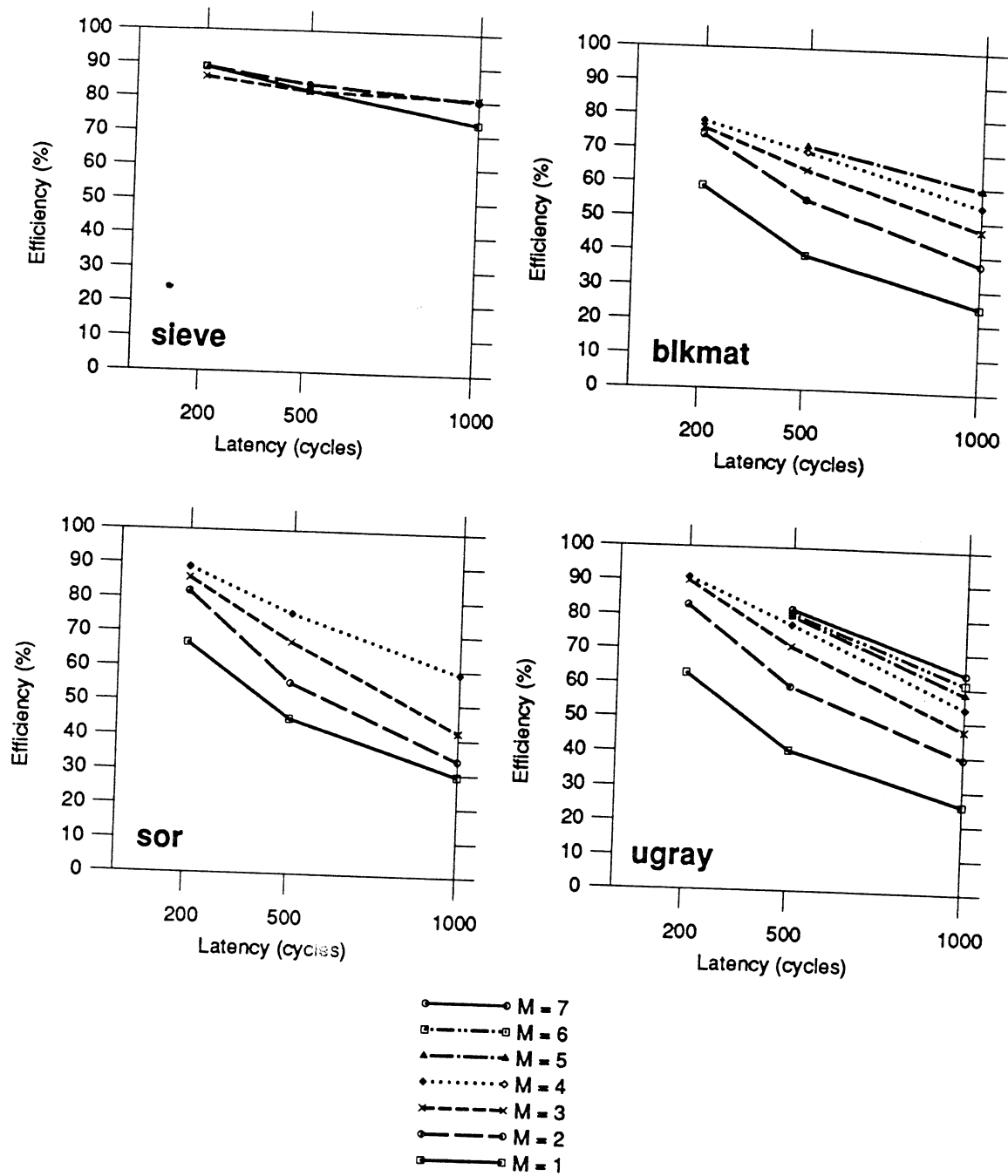


Figure 7.9: Efficiencies with longer latency.

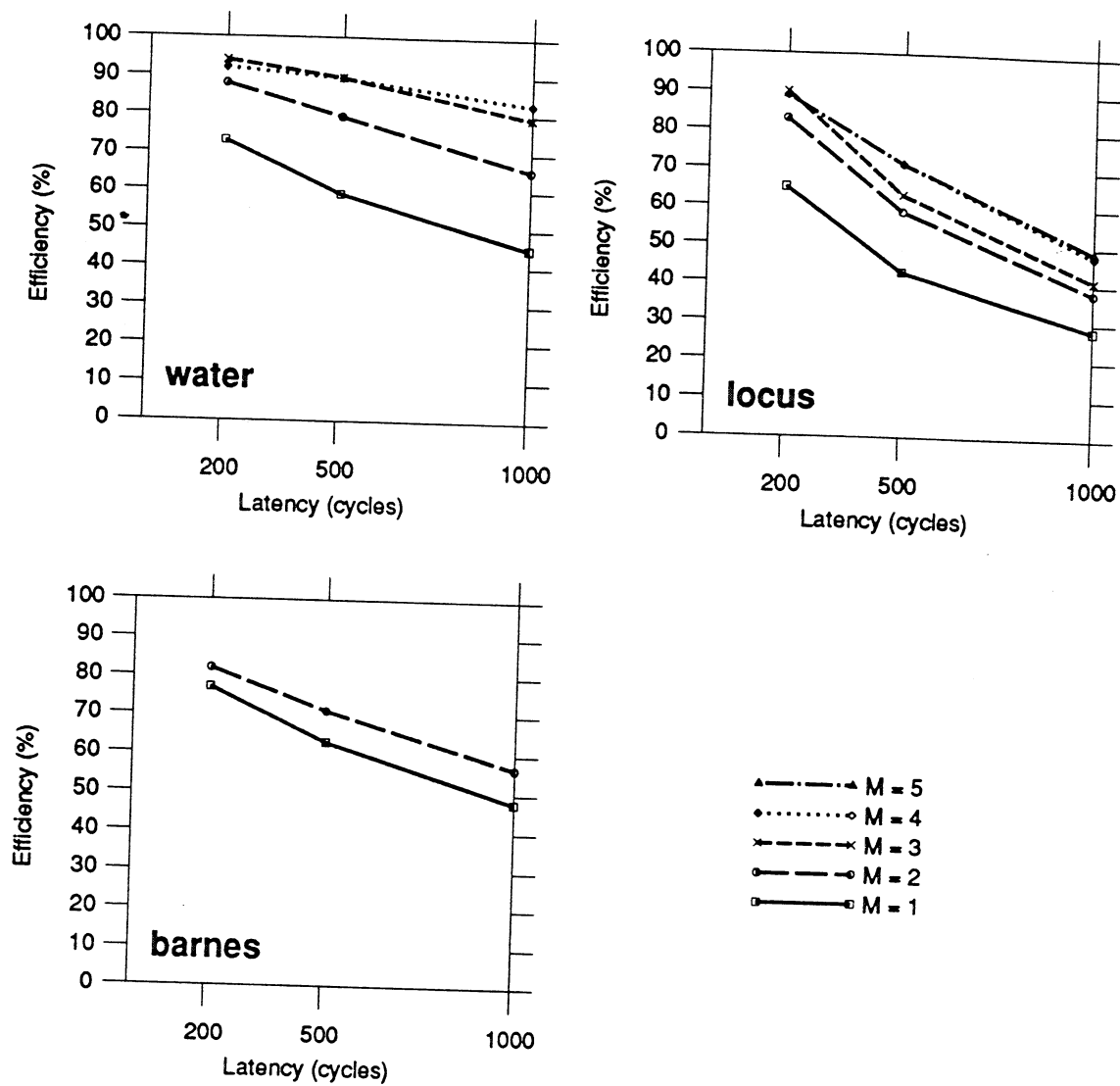


Figure 7.10: Efficiencies with longer latency.

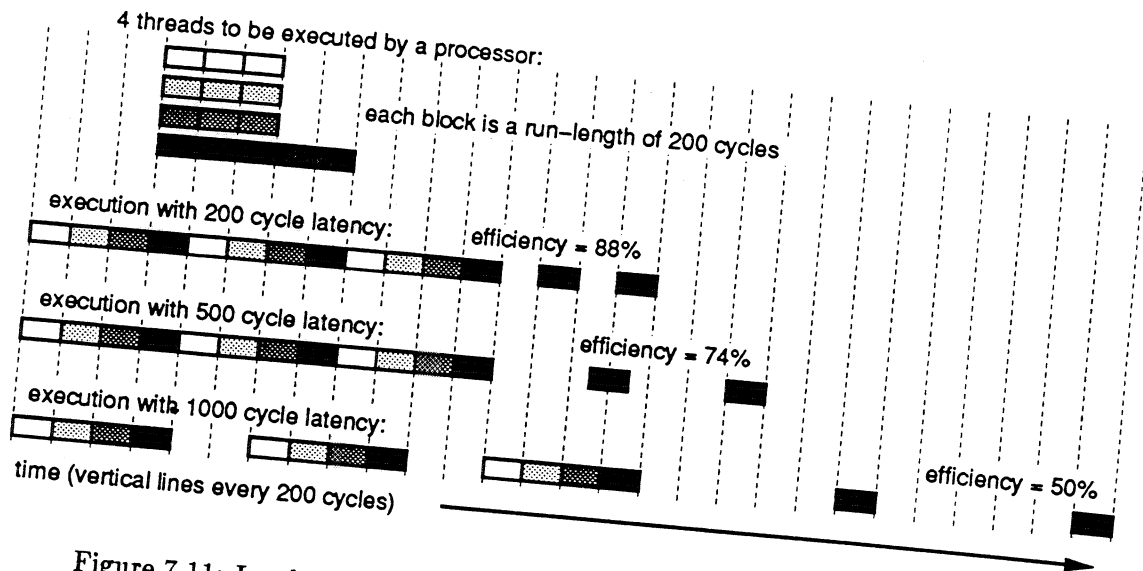


Figure 7.11: Load imbalance between threads has an increasing performance impact with longer latencies.

latencies. Multithreading, however, is not able to bring efficiencies back up to the same levels achieved under smaller latencies. Thus multithreading is unable to completely hide the affects of increased memory latency.

There are a number of factors in this decrease in attainable performance under longer latencies:

**Short Run-Lengths:** When several threads have coinciding short run-lengths (and thus the latency is not hidden) the stalls last longer because the latency is longer.

**Load Imbalance:** When threads finish a phase of computation (reach a barrier) at different times, the last thread to finish takes a longer time to finish because of the longer waits for each of its memory references. This is shown in Figure 7.11, where four thread are shown. The first three thread each execute for three run-lengths of 200 cycles, but the fourth thread executes for five run-lengths of 200 cycles. As the network latency increases, completion of the unbalanced thread takes a larger fraction of the total execution time.

**Lock Contention:** A typical critical section involves: obtaining a lock, doing a few operations and references, and then releasing the lock. When latencies are longer it takes longer for an individual thread to complete a critical section and thus locks are held

for longer periods of time, increasing lock contention.

**Finer Granularity:** As more threads are demanded from a fixed sized problem, the problem must be divided into smaller pieces or work. This finer partitioning leads to more communication and thus shorter run-lengths.

There also several ways to ameliorate the impact of longer latencies:

**Larger Problem Sizes:** When the applications are run with larger problem sizes, the granularity of tasks can be increased and/or more threads will be available to hide latency.

**Larger Caches:** Larger caches can decrease the miss rates, which both increases average run-lengths and decreases the number of points at which stalling might occur.

**Better Load Balancing:** More careful load balancing of threads will help limit the duration of the slow completion process of the last thread.

Despite the lower efficiencies achieved with multithreading under very long latencies, multithreading provides larger net performance gains. The typical application (such as `sort`) has a 100% performance improvement (from 30% to 60% efficiency) under multithreading with a 1000 cycle latency. Whereas with a 200 cycle latency, typical performance improvements were 33% (from 60% efficiency to 80% efficiency).



## Chapter 8

# Hardware Support

Most previous research into multithreading machines has involved complex hardware to support the switch-every-cycle model[ACC<sup>+</sup>90, HF88, Kow85, PC90]. These machines were built to switch every cycle so as to allow a fast pipeline implementation without concern for data dependencies. In addition, this multithreading mechanism is used to tolerate long unpredictable memory latencies. These machines typically also provide sophisticated synchronization (using full/empty bits on memory), and support powerful programming models allowing rapid and dynamic creation of threads.

Unfortunately, every hardware capability has its cost. TERA[ACC<sup>+</sup>90], for instance, allows fast dynamic creation of threads, and it provides 128 banks of 32 registers to hold these threads' register values. This huge amount of hardware complicates the machine because it slows down the access time to the register file. Alternatively, Monsoon[PC90] limits the size of the register file by severely restricting the number and lifetimes of registers, which undoubtably has a negative performance impact. It remains unclear if these machines offer advantages in either performance or ease of programming compared to the simple multithreaded shared memory model studied in this dissertation.

We have studied multithreading models which we feel have a balance between computational flexibility and implementation simplicity. Only a small number of threads are allowed per processor and thus the register file can be kept reasonably small. The programming model involves a static set of threads that is used for the lifetime of the program and thus support for fast dynamic thread creation is not needed. The synchronization mechanisms are simple and similar to remote memory references. Finally, thread scheduling uses simple policies and only switches threads at special events. In this chapter we will present

the hardware mechanisms necessary to support the multithreading models studied in this dissertation

Our goal is to show that the hardware is simple enough for a single chip implementation. In fact the MIT Alewife project is using a slightly modified SPARC processor as a multithreaded processor by using its register windows to hold multiple contexts instead[ALKK90]. We will present the ideas in this chapter as modifications or additions to an ordinary pipelined RISC processor, and we expect that these ideas can be incorporated into existing processor designs. These ideas should also extend to superpipelined and superscaler processors and thus the considerable development effort into these processors can be leveraged in the development of multithreaded processors.

## 8.1 Hardware for Explicit-Switch

This section describes the hardware mechanisms needed on a multithreaded processor that provides the **explicit-switch** multithreading model. We focus on **explicit-switch** because this was shown in Chapter 4 to be superior to **switch-on-load**. The hardware support required for **switch-on-load** is virtually identical.

Figure 8.1 shows a simplified data path for a RISC processor. A real processor would be more complex, but this diagram is sufficient for explaining the ideas developed in this section.

Figure 8.2 shows a revised data path that has been modified to support the explicit-switch multithreading model. This diagram shows support for a multithreading level of at most 4 threads per processor. There are 4 register sets, 4 program counters, and 4 instruction registers. Earlier simulations suggested that an actual processor should be built to allow perhaps 10 threads per processor if a latency of 200 cycles is expected.

In addition to the extra register sets, PC's, and IR's, we have shown the addition of a scheduler block, a larger control unit because of the increased complexity, and a path from memory directly into the register file. Each of these changes will be discussed below.

### 8.1.1 Pipelined Context Switch

While switch-every-cycle machines have no lost cycles due to a context switch, most research on other multithreading models has assumed non-zero context switch times. For example, Gupta *et al.*[GHG<sup>+</sup>91] assumed a cost of 4 cycles for flushing the processor

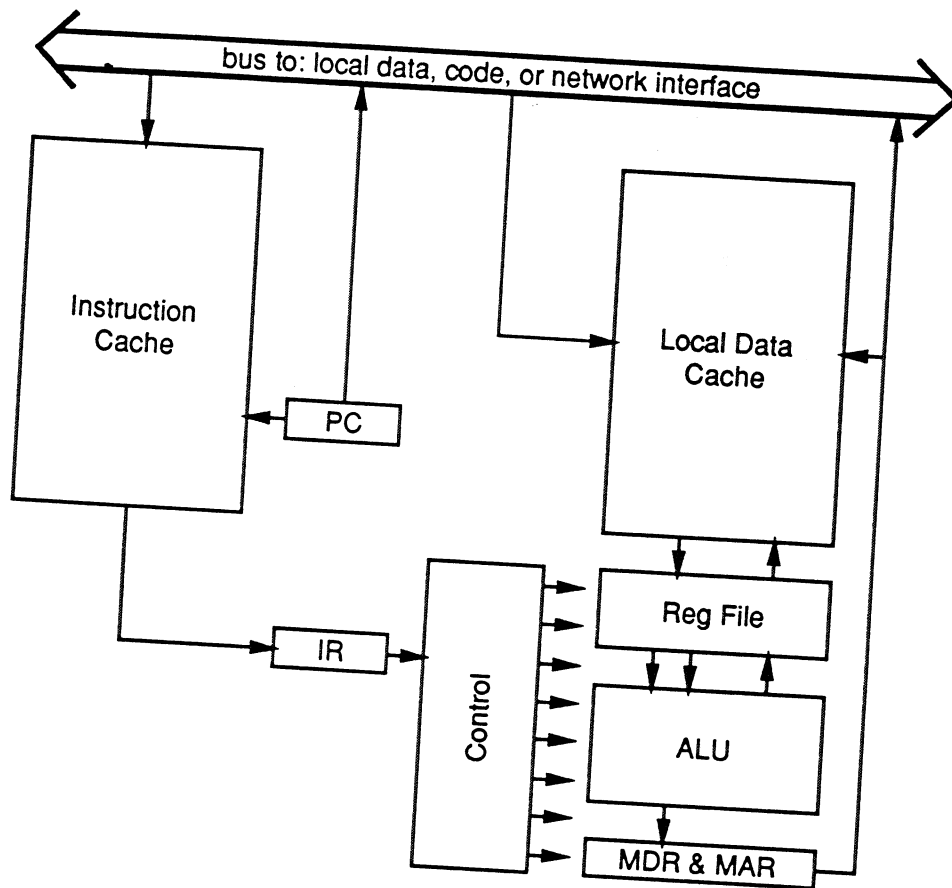


Figure 8.1: Datapath for RISC processor.

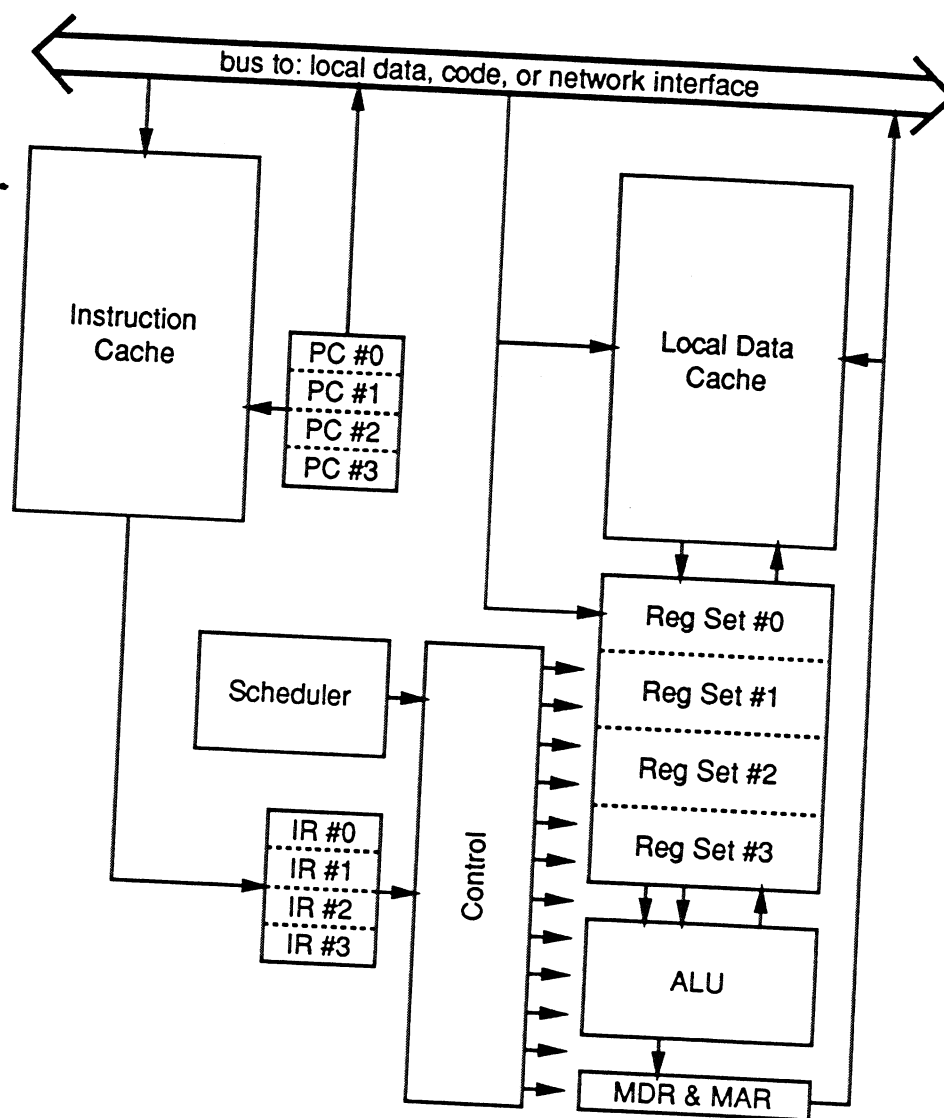


Figure 8.2: Datapath with changes for explicit-switch multithreading.

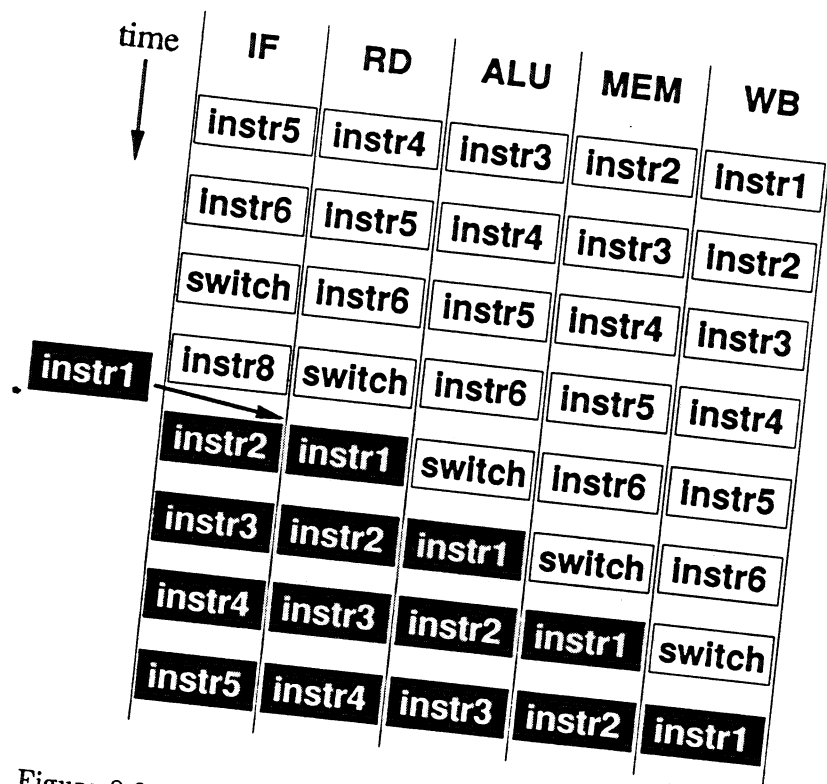


Figure 8.3: Pipelined context switch for explicit-switch.

pipeline, or 16 cycles for a less aggressive implementation. For the **explicit-switch** model considered in this section, we will show that flushing the processor pipeline is unnecessary and thus a single cycle context switch is possible.

The key to the single cycle context switch is to pipeline the context switch just as other operations are pipelined. Figure 8.3 shows the content of the processor pipeline<sup>1</sup> as it progresses from running one thread (shown in white), through a context switch, and the start of the next thread (shown in black). Each horizontal line represents a cycle. Time progresses down the page one cycle at a time, and the progress of instructions through the pipeline is shown by their movement to subsequent pipeline stages.

The seventh instruction for the white thread is a switch instruction. When the switch instruction is in the the decode stage (RD), the processor detects that the thread is to be context switched. At this point the processor has already fetched instruction eight of the white thread. However, instead of passing instruction eight through the pipeline, the processor can substitute the first instruction from the black thread if it is available. This is

<sup>1</sup>The pipeline stages are those of the MIPS R3000[Kan89].

Application (multithreading)	Efficiency switch = 1 cycle	Efficiency switch = 9 cycles	Loss
sieve (mt = 5)	86	86	0%
blkmat (mt = 3)	80	77	3%
sor (mt = 9)	83	67	16%
ugray (mt = 10)	76	68	8%
water (mt = 5)	90	89	1%
locus (mt = 8)	80	72	8%
mp3d (mt = 11)	89	73	16%
barnes (mt = 5)	87	83	4%

Table 8.1: Explicit-Switch: Performance loss with 9 cycle context switch.

the purpose of the additional instruction registers in Figure 8.2: to hold the prefetched first instruction from each thread. The multiple program counters are needed to allow immediate commencement of instruction fetching from the new thread.

The cost of the context switch for this method is just one cycle, which is the switch instruction itself. Notice that during the transition from the white thread to the black thread, instructions from both threads occupy the pipeline concurrently. This means that the black thread will start reading results from the black register set while the white thread is still writing results into white register set. To allow different pipeline stages to access different register sets, the processor must pass the register set number down the pipeline with the instructions. This transition between threads is similar to the transition between register windows in the SPARC processor.

The importance of a fast context switch depends upon the frequency of context switching. If context switches occur at intervals of hundreds of thousands of cycles, as they do in uniprocessor operating systems, the context switch time is not critical. But if context switches occur every 20 cycles, as they might on a multithreaded processor, even just a single cycle spent context switching decreases performance by 5%.

A slower context switch would result if the CPU pipeline were drained before starting a new thread [GHG<sup>+</sup>91]. For a pessimistic value we assume an 8 cycle pipeline as on the MIPS R4000 [MIP91], plus 1 cycle for the switch instruction. This gives a context switch cost of 9 cycles.

Table 8.1 shows the performance loss that would result from a 9 cycle context switch instead of a 1 cycle switch as has been studied in this dissertation. Simulation parameters are the same as they were in the studies of **explicit-switch** with inter-block

grouping (refer to Table 4.4) except that the context switch time has been increased. The performance loss varies by application. The applications that have long average run-lengths (such as `sieve`, `blkmat`, `water`, and `barnes`) lose just a few percent of their performance. The other applications (`sor`, `ugray`, `locus`, and `mp3d`) have shorter run lengths, context switch more frequently, and thus incur a larger performance loss from the slow context switch. The worst performance loss is 16% which occurs for `sor` and `mp3d`.

The performance losses are lower than we expected for two reasons. First, average run-lengths for the explicit-switch model with inter-block grouping (as seen in Section 4.3.2) range from 30 cycles to more than 200 cycles. With longer run-lengths, the context switch time has less impact on performance. Second, some of the cycles lost to the slower context switch would have otherwise been lost to memory latency. This is a small effect, since multithreading is usually effective at hiding most of the memory latency.

This section has shown that pipelining the context switch is feasible and can provide as much as 16% performance improvement over a slower context switch that waits for the pipeline to drain before starting the next thread.

### 8.1.2 Result Matching

Multithreading allows issuing multiple memory references into the network to hide network latency. A difficulty arises because most networks do not preserve message order and thus the responses must be matched with the requests.

A simple solution is to send a small tag along with each message. This tag is later used to identify the returning result message. The tag should contain the thread number of the issuing thread and the register in which to put the result. This allows writing the result directly into the register file through a second write port as shown in Figure 8.2. By storing results directly into the register file, no special storage is needed and they are immediately available upon resumption of the thread. Alternatively, the second write port can be eliminated if the results are buffered and later written into the register file during cycles in which the processor does not write to the register file.

### 8.1.3 Scheduling

The main task of the scheduler is to determine when a thread is ready. In the multithreading models we have looked at, a thread becomes ready when all of its shared

memory accesses have returned from the network. To keep track of outstanding references the scheduler will need a counter for each thread. The counter is incremented on each shared load issued into the network and is decremented upon its return. When the counter reaches zero, the thread becomes ready.

The ready threads may be scheduled with any sort of scheduling policy. For **explicit-switch** we used first-come-first-serve (which is the same as round robin when accesses return in order.) This is simple and fair. Other policies, such as those studied in Section 5.2.1, might provide some additional benefit, for instance, by causing timeouts on long run-lengths. However since long run-lengths are uncommon under **explicit-switch**, we expect the benefits of more complex policies will be small.

#### 8.1.4 Multiple Register Sets

The largest change to the processor design in terms of chip area is the addition of multiple register sets. These multiple register sets are essential for fast context switching because without them it would take at least 128 cycles to save the register set<sup>2</sup> for the current thread out to memory and then load in the register set for the next thread. This overhead for context switching would overshadow any gains made from hiding the memory latency.

In a typical RISC processor design, the register file only occupies a few percent of the chip area. On the Stanford MIPS processor[PGH<sup>+</sup>84], for example, the register file occupied 8.3% of the chip area. For more recent processors with large on chip caches, the percentage of chip area used for the register file is even less. At this size, providing 10 register sets on chip, as was found to be sufficient to support the **explicit-switch** multithreading model, is conceivable. However, multithreading designs that allow hundreds of threads per processor, such as TERA, have so far been prohibited from considering single chip implementation because of the size of their register files. Other multithreading designs, such as Monsoon and  $\star T$ [NPA92], do not provide a separate register set for each thread.

The precedent for increasing the register file size has already been set by the SPARC[Fuj88] chip and the Am29000[Man92]. SPARC has 120 integer registers, and the Am29000 has 192. Most of these are used to provide register windows to help speedup procedure call and return by shifting to a new register set rather than saving and restor-

---

<sup>2</sup>Here we assume that the register set is 32 general purpose and 32 floating registers.



ing registers to memory. However, the benefit of these register windows is small because compilers have been able to do a good job of avoiding most register saves and restores. Some researchers have therefore proposed using the register windows for multiple contexts instead[APRIL]. Unfortunately, as far as multithreading is concerned, the SPARC architecture does not provide register windows for the floating registers.

### 8.1.5 A Denser Register File

If the multithreading level and the number of register sets supported is small (say 4), the chip area used for the registers will be comparable to that used on the SPARC or AM29000 chips. Supporting  $M = 4$  is thus clearly reasonable. However as the number of threads and register sets is increased, the chip area will become more of a concern and we therefore propose the following design which can be used for a denser implementation of the register file for a multithreaded processor.

The key to this design is that only the register set of the currently active thread is used by the processor. The other register sets sit idle until their thread is scheduled by the processor. Rather than keep all register sets in large multiported register cells, the inactive register sets can be kept in smaller single ported register sets until they are needed. If the single ported cells are implemented as dynamic memory, then on a VLSI chip they will require less than one twelfth the area of a regular multiported static cell<sup>3</sup>.

The main obstacle in implementing this is being able to switch to a new active register set quickly at a context switch. At a context switch, the entire contents of the active register set must be saved into the inactive storage area, and the register set of the next thread must be loaded. For a register file of 32 registers, each of which is 32 bits, this constitutes 1024 bits that must be saved and another 1024 bits to be loaded. If done quickly, i.e. in parallel, this can be done in two cycles. In the first cycle all 1024 bits are transferred out of the active register file into the inactive register file, and in the second cycle the new 1024 bits are transferred in. Moving 1024 bits into (or out of) the register file in a single cycle would require 1024 wires, and this would be unwieldy.

This obstacle can be overcome by interlacing the inactive register file within the active register file as shown in Figure 8.4. This shows a single block that can be used in an array of 32 by 32 blocks to implement the full collection of active and inactive register sets.

---

<sup>3</sup>A static cell with 2 read and 1 write ports can be implemented in  $64\lambda \times 41\lambda$ . A dynamic cell in  $10.5\lambda \times 18\lambda$ [Waw91].

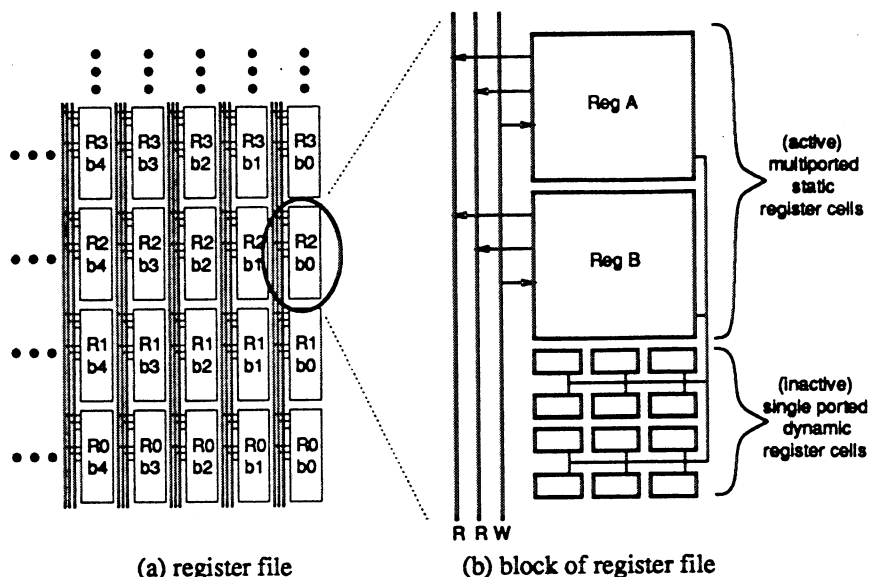


Figure 8.4: One bit of register file supporting 12 threads per processor.

The figure shows 12 single ported dynamic register cells for the inactive register sets and 2 multiported register cells for the active register sets. Usually just one of the two active register sets is used, except that when the processor is in transition from one thread to the next, instructions from both threads are in the pipeline and thus both register sets are needed.

The multiported register cells are used in an alternating fashion as shown in Figure 8.5. This example shows the transition from running one thread to the next and then to a third (shown in white, grey, and black respectively). At the start, the white thread has been executing out of the A-registers, and the B-registers have been loaded with the register set for the grey thread. When the white thread context switches, the registers for the grey thread are available and it can start executing immediately. The A-registers are retained until all of the white thread's instructions have exited the pipeline (except the switch instruction which does not use any registers). At this point, the A-registers are written into the dynamic memory storage area used for the inactive threads. The A-registers are now loaded with the register values of the black thread. By alternating between the A-registers and the B-registers, the active registers can always be kept available.

This technique allows enough registers for twelve contexts to be implemented in the space that would normally be needed for three. This compact register file keeps the bus

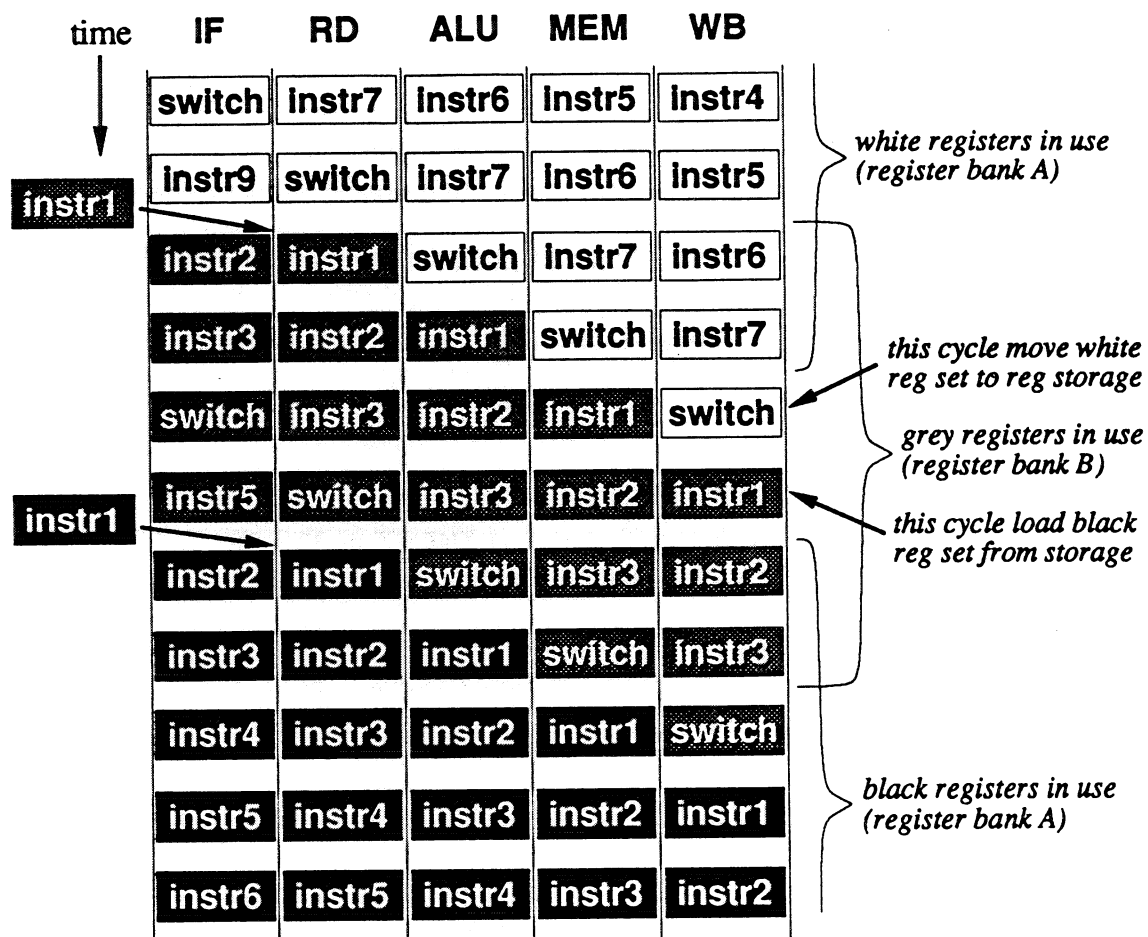


Figure 8.5: This shows the operation of the denser register file design. During the transition between threads, both the A register bank and the B register bank are in use simultaneously.

loading and lengths of the register read and writes busses within acceptable bounds. As the number of register sets is increased, this design becomes more desirable because additional register sets can be added using only one twelfth the area that would be required for adding a multiported static register set.

A potential complication arises because dynamic registers are often difficult to use. In a dynamic register, the value is set by trapping a charge on a small capacitor. This small capacitor has limited driving capacity, its charge is destroyed when it is read, and it slowly leaks and therefore must be refreshed periodically. These are characteristics that are all acceptable for use in our register file design. The limited driving capacity is acceptable because only a small number of cells are on any wire and these wires are short. Destructive read is acceptable because register values are re-written when a thread completes its active phase. And refreshing is not needed because the threads and their registers are constantly being cycled through the processor.

A minor limitation of this design is that there is a minimum period after a context switch before the processor can context switch again. This minimum period is 4 cycles for a 5 stage pipeline like the MIPS R3000 and is shown in Figure 8.5. The grey thread (which uses register bank B) executes for the minimum period of 4 cycles, during which the A register bank is in use every cycle either by the white or black thread or for saving and restoring registers. This 4 cycle minimum on the context switch interval should not pose a problem since context switches rarely occur this frequently.

## 8.2 Hardware for Switch-On-Miss

This section describes the additions and changes to our multithreaded processor that are needed to support the switch-on-miss multithreading model. A revised processor datapath is shown in Figure 8.6. The main change is the addition of a cache for shared data and support for cache coherency. Since the processors are multithreaded, the caches must be *lock up free* so that they can continue operating while misses are being serviced in the network.

### 8.2.1 Cache Coherency

Supporting cache coherency is complex [HLRW92], and some machines, such as CRAY's parallel vector computers and the TERA computer, choose to put their complexity

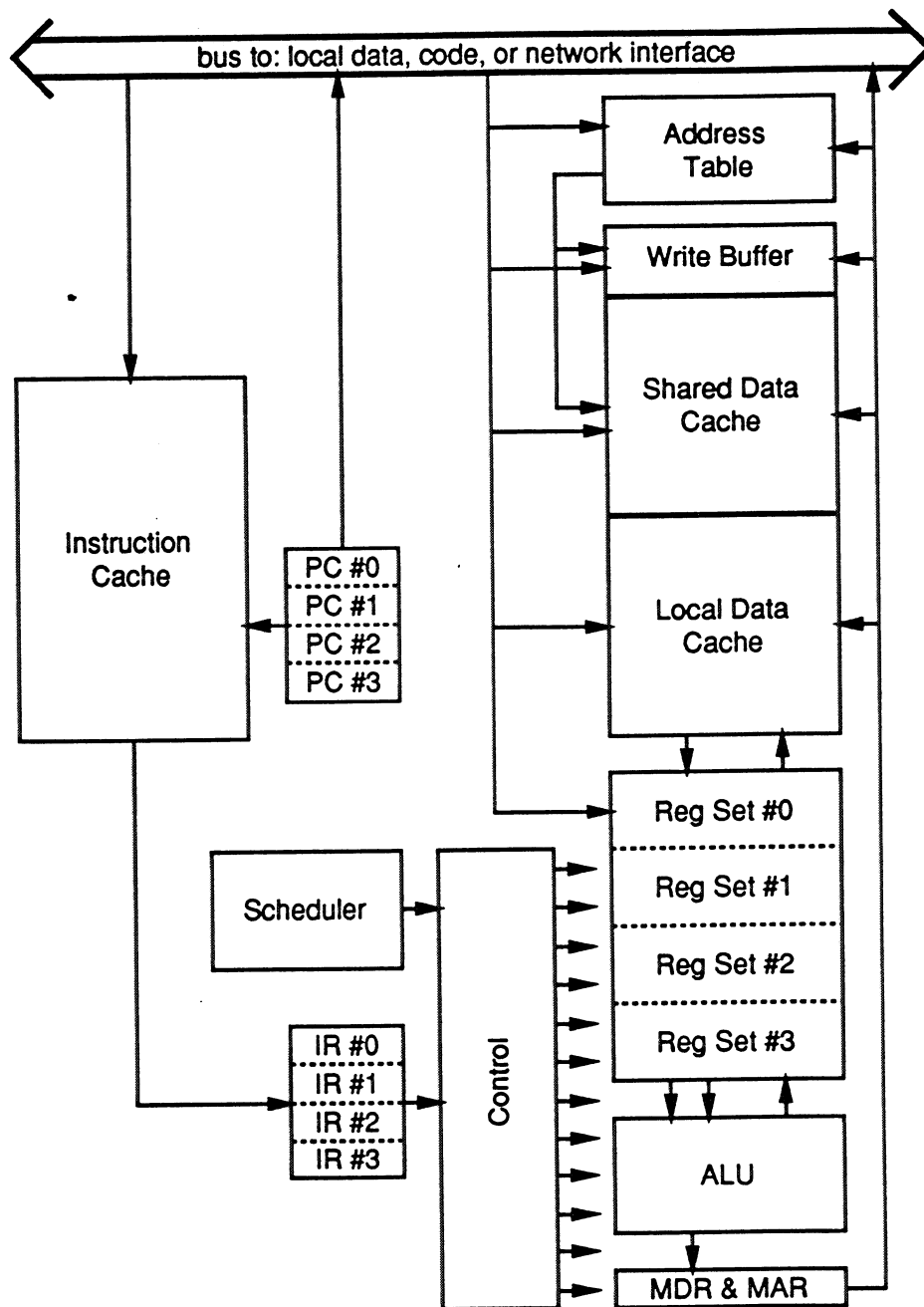


Figure 8.6: Datapath with changes for switch-on-miss multithreading.

elsewhere and build machines without caching of shared data. However, we suggested in Chapter 6 that it may be more cost effective to provide shared data caches and cache coherency than to build networks that provide enough bandwidth to satisfy the needs of large cacheless multiprocessors.

Throughout this dissertation we have used the Censier & Feautrier[CF78] (C&F) invalidation protocol. This is a conceptually simple protocol, and a slight variation was used in the 64 processor DASH prototype[LLJ+92]. However, for a large parallel machine, C&F is impractical to implement in hardware because each memory line must be accompanied by a full directory vector that indicates which caches hold a copy of that line. For a 1024 processor machine, the overhead would be 1024 directory bits for each 16 byte (128 bit) data line.

More recent studies have suggested that variants of the C&F protocol can obtain nearly the same performance with reasonable hardware cost[ASHH88, ON90]. Since the performance is nearly the same, we chose to simulate the conceptually simpler C&F protocol.

### 8.2.2 Result Matching and the Address Table

In Section 8.1.2 we introduced the idea of sending a small tag along with each remote reference, and upon the return of the result, using that tag to put the result directly into the register file. This direct storage of results into the register file makes the implementation of cache coherency slightly easier by eliminating a possible race condition. This race occurs when a result is placed only in the cache and thus must remain there until the requesting thread has a chance to use it. If some other thread somehow causes the result to be evicted from the cache, the requesting thread will find it missing and have to fetch it again. It is possible that this could result in deadlock[GHG+91]. To avoid deadlock, the results must be locked into the cache until they are used by the requesting thread. Thus the simplification afforded by placing results directly into the register file is that the cache is relieved of its obligation to lock data until it is used.

These tags are also important in conserving network bandwidth. Normally an outgoing read message would contain the address, and the return message would contain both the address and the data. The address in the return message is needed to store the data into the cache. The returning of the address can be eliminated, however, by saving the address in a local table and retrieving it using the message tag upon the result's return.

This table is the address table shown in Figure 8.6. A similar idea was proposed by Dickey & Kenner[DK92] where the tag was called the *Outstanding Request Index*.

### 8.2.3 Stalling On Writes

We have assumed throughout our simulations that the processor never stalls on writes. For a machine without caching, writes are simply issued into the network and the processor proceeds. Since writes do not return a result, there is usually no reason to stall (or context switch to another thread).

The one case where one might need to stall the processor is to satisfy the requirements of the memory consistency model. For example, to provide sequential consistency[Lam79], all accesses must appear as if they were executed in some global order that is consistent with a sequential interleaving of the thread executions. If the network does not preserve the ordering of accesses, then to be safe a processor must stall after each remote access until that access is completed. Most networks do not preserve ordering of accesses, but some such as the Fluent[Ran89] network do.

Alternatively, the processor can avoid stalling by providing a weaker consistency model such as weak consistency[DSB88] or release consistency[GLL<sup>+</sup>90]. These are applicable to programs that use explicit synchronization primitives (such as locks) to coordinate the interchange of data among threads. This class of programs (to which all of our applications belong) are called data race free[AH90]. For these programs, waiting for accesses to complete is only required at synchronization points. This effectively eliminates almost all stalls on writes.

An additional cause for stalling occurs on systems with caching. When a processor tries to write a value, the processor must first have exclusive ownership of the value's cache line. If the cache does not contain the line (or exclusive ownership status), it must be obtained (over the network) before the write can be performed. In order to allow the processor to continue executing, the pending write can be put into a write buffer[PH90]. This buffer holds the write operation until the cache line is obtained, and then it performs the write.

For the simulation studies in this dissertation, we have assumed a write buffer of sufficient size to hold all pending writes, and thus we never stall or context switch the processor because of a write to shared memory.

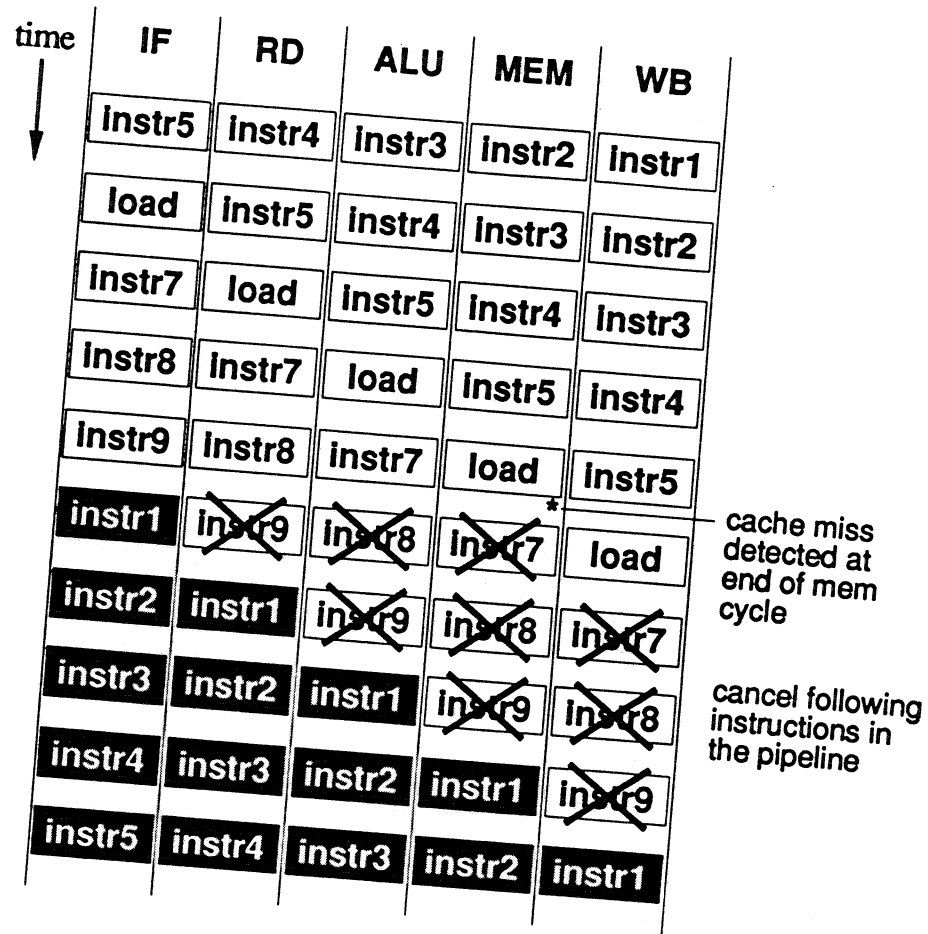


Figure 8.7: Pipelined context switch for switch-on-miss.

#### 8.2.4 Context Switch Delay

For the **switch-on-miss** multithreading model, a context switch can no longer be performed in a single cycle as it was for the **explicit-switch** model as described in Section 8.1.1. The difference is that now the context switch decision is based on the detection of a cache miss, which occurs deep in the processor pipeline, rather than during the decoding of the instruction opcode, which occurs near the start of the pipeline.

Figure 8.7 shows a pipelined context switch that is caused by a cache miss. The sixth instruction of the white thread is a load instruction that cause a miss, and this miss is detected at the end of the MEM stage. At this point three subsequent instruction have already entered the pipeline. These three instruction will be canceled, and thus the cost of the context switch is three cycles.



Application (multithreading)	Efficiency switch cycles: on miss = 3 on timeout = 0	Efficiency switch = 8 cycles	Loss
sieve (mt = 1)	89	86	3%
blkmat (mt = 3)	79	77	2%
sor (mt = 4)	88	85	3%
ugray (mt = 3)	88	85	3%
water (mt = 3)	91	90	1%
locus (mt = 2)	83	78	5%
mp3d (mt = 11)	86	77	11%
barnes (mt = 2)	82	80	2%

Table 8.2: Switch-On-Miss: Performance loss with 8 cycle context switch.

If, however, the context switch is caused by a timeout rather than a cache miss, the context switch can be performed at the start of the pipeline rather than from deep within it. This means that a context switch caused by a timeout can be fully pipelined and thus without any wasted cycles.

Table 8.2 shows the performance loss if all context switches take 8 cycles instead of either 3 or 0 as just explained. The experimental parameter are otherwise the same as in the studies of **switch-on-miss** (refer to Table 5.6. Excluding mp3d, which does not cache well, the performance loss due to the longer context switch is typically 2% or 3% with the worst case being 5% for locus. In fact, the performance loss is somewhat overstated here because of the timeout switches. Other scheduling policies that introduce fewer spurious context switches would be able to mitigate the performance loss further.

These results show that a fast pipelined context switch provides only small performance gains over a less aggressive implementation that drains the pipeline before starting the next thread. This smaller performance impact of context switch time, compared to that under explicit-switch, results from the longer run-lengths between context switches.

### 8.3 Conclusions and Extension to Multiprogramming

In this chapter we have indicated that the hardware mechanisms needed to build an **explicit-switch** or **switch-on-miss** multithreaded processor are reasonable. We have presented them as modifications and additions to a simple RISC processor, and we have suggested that multithreaded processors might be designed by modifying current microproces-

sor designs. Some researchers have recently proposed that in the future even uniprocessors should be multithreaded[CGL92, FP91, LGH92] because of increasing memory latencies and the increased difficulty of scheduling deeper and wider pipelines.

Some of the simplicity of the multithreading support hardware presented in this chapter comes from the fact that we have only considered the parallel machine to be running a single application at a time. If instead we had tried to provide a more general parallel processor, such as TERA[ACC<sup>+</sup>90], where each processor could be shared by threads from several different programs, there would have been additional hardware complexities. TERA allows up to 128 threads per processor and 16 simultaneously executing programs. The large number of threads requires a very large register file. And the simultaneous execution of multiple programs requires memory protection to protect all the threads from each other.

Multiprogramming is a very desirable attribute of a parallel machine. At times the machine must clearly be devoting 100% of its resources to solving a single large problem, for otherwise such a large machine would not be needed. But often the machine must support the simultaneous development and testing of applications by many programmers. We believe the approach taken by the CM-5 is a good compromise. The CM-5 allows the machine to be partitioned into smaller machines for independent use, and it also allows the entire machine to be time sliced between applications. This time slicing can be done at intervals in the range of seconds, to limit the throughput loss due to stopping the machine, draining the network, and switching to a new process.

Partitioning the machine allows a real time user to collar a portion of the machine, whereas time slicing allows multiple application developers to share the machine while testing their applications at the full machine size. Under these policies, the individual processors are always executing just a single program at a time, and thus the processor model presented in this chapter is adequate for building real parallel machines.

An additional benefit of multithreading over current parallel machines is that the number of processors assigned to an executing program can be easily changed. For example, consider a program that is being run at a multithreading level of  $M = 2$  and is using all 1024 processors. If a second program is started, the first program can be compressed down to use 512 processors at  $M = 4$ , or 256 processors at  $M = 8$ . Without multithreading it is difficult to change the number of processors since somehow the running program must be reconfigured to use fewer threads.

## Chapter 9

# Conclusions and Future Directions

In this dissertation we have identified long memory latency and limited network bandwidth as the main problems impeding the development of large shared memory multiprocessors. It appears inevitable that any large interconnection network will have long latencies because of the many nodes, links, and delays which messages will encounter on their transit through the network. Also any network that provides some level of scalable bandwidth will represent a large fraction of the hardware in a large system, and thus cost effectiveness will dictate that bandwidth be limited.

We do not expect that any technological innovation in network design will be able to eliminate the latency and bandwidth problems, so instead we have focused on making the processor tolerant of long latencies and reducing and quantifying the amount of network bandwidth needed.

### 9.1 Conclusions

The main focus of this research has been on models of multithreaded processors whose design is similar to current state-of-the-art microprocessors except for the addition of support for fast context switching. These multithreaded processors will execute instructions in the same pipelined (and superscalar) fashion as current microprocessors except that at certain instructions (determined by the particular multithreading model) they will quickly context switch to another thread rather than wait for completion of a remote reference.

For machines without caching of shared data, we have looked at the **switch-on-load** and **explicit-switch** multithreading models. **Switch-on-load** performs poorly

because context switches occur frequently and often very close together. This means that a large number of threads will be needed to hide the latency, and because of the many short run-lengths, sometimes the latency still will not be completely hidden.

**Explicit-switch** improves performance by introducing an explicit context switch instruction. This instruction can be used by an optimizing compiler to group together independent shared memory references. This grouping allows a single thread to issue multiple references into the network before switching to another thread. The run-lengths are increased, and the distributions are significantly improved by the elimination of most short run-lengths.

Our results show that **explicit-switch** is able to tolerate latencies of 200 cycles by using a multithreading level of 10 threads (or less) per processor, and that this is sufficient to allow all of the applications studied to obtain efficiencies of 80%.

However, since there is no caching of shared memory in these systems, all shared references are sent across the network, and thus the resulting network bandwidth demands can be quite high. These bandwidth demands vary considerably across the applications and range as high as 30 bits/operation. We expect that providing such high bandwidth will be expensive, and thus conclude that although multithreaded systems without shared data caching can achieve high execution efficiencies, they may not be cost effective because of their high bandwidth requirements.

Caching was effective for all but one of our applications (which has since been rewritten to improve its caching behavior). For the rest of the applications, caching was able to reduce the average bandwidth requirement to under 2 bits/operation. This large reduction in bandwidth suggests that the cost and complexity of maintaining coherent caches on a large machine will be justified by the savings afforded from use of a skinnier network.

Caching is also beneficial in that it filters out many of the remote references and thus eliminates many potential long latency operations. Typical miss rates ranged from 1% to 4%. With so many fewer remote references, the impact of long latency is diminished. Our results show that with latencies of 200 cycles, execution efficiencies of 60% to 70% can be achieved without multithreading, and that with a multithreading level of 3 threads per processor, efficiencies can be raised to 80% to 90%.

Multithreading systems both with and without caching are able to achieve efficiencies of 80%. The advantage of caching is that it reduces the multithreading level and

the amount of network bandwidth that is required.

While most of our experiments have assumed that there will be adequate network bandwidth available, we also looked at the impact on performance of having limited bandwidth, as will be the case for real networks. Our results for large (256 processor) systems (using both multithreading and caching) show that network traffic will be bursty. If the system is to have at most minor performance degradation, then the network will need to supply a remote reference bandwidth of 2 to 4 bits/operation and a memory module bandwidth of 8 to 16 bits/operation. The higher memory module bandwidth is necessary because of random hot spot congestion.

## 9.2 Future Directions

We expect in the future that the latency problem will continue to increase. Processors will continue to get faster, and because of both higher clock rates and superscalaring, they will issue remote references at higher rates. Ever larger parallel machines will also be desired and these will have larger networks and longer latencies. These faster processors and larger networks will both contribute to an increased need for latency tolerance.

Without caching, our simulations show that tolerating a 200 cycle latency requires 10 threads per processor. We expect that the number of threads needed will grow at least linearly with latency, and thus at significantly larger latencies, the number of threads needed may grow prohibitively large. An important aspect of these machines that needs further research is the amount of inter-block grouping that can be obtained by a smart compiler. Our estimates in Section 4.3.2 suggests that research in this area should be successful.

With caching, our simulations in Section 7.4 show that with longer latencies the performance of a single threaded processor drops to 30% efficiency at a latency of 1,000 cycles. At this point multithreading is very beneficial and can double the performance to 60% efficiency, while still using just a moderate number of threads per processor.

In the introduction of this dissertation we listed six mechanisms for reducing the impact of memory latency: caching, multithreading, weak consistency, prefetching, layout, and aggregation. In this dissertation have focused primarily on the first three techniques. These are the most hardware oriented and the most broadly applicable, but there is potential benefit from all of these mechanisms, and we suggest future research should look at exploiting all of these techniques.

## Bibliography

- [ACC<sup>+</sup>90] Robert Alverson, David Callahan, Daniel Cummings, et al. The TERA Computer System. In *1990 Int. Conf. on Supercomputing*, pages 1–6, 1990.
- [AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Co., 1989.
- [Aga92] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, pages 525–539, September 1992.
- [AH90] Sarita V. Adve and Mark D. Hill. Weak Ordering — A New Definition. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 2–14, 1990.
- [ALKK90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiawicz. APRIL: A Processor Architecture for Multiprocessing. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 104–114, 1990.
- [And90] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 6–16, January 1990.
- [ASHH88] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *The 15th Annual Int. Symp. on Computer Architecture*, pages 280–289, 1988.
- [BBN89] BBN Advanced Computers Inc. *TC2000 Technical Product Summary*, 1989.

- [BDCW91] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.
- [Bel85] C. Gordon Bell. Multis: A New Class of Multiprocessor Computers. *Science*, 228:462–467, April 26 1985.
- [BJS88] F. Baskett, T. Jermoluk, and D. Solomon. The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons per Second. In *Proc. Compcon Spring 88*, pages 468–471, 1988.
- [Boo89] Bob Boothe. Multiprocessor Strategies for Ray-Tracing. Master's thesis, U.C. Berkeley, September 1989. Report No. UCB/CSD 89/534.
- [BR90] Roberto Bisiani and Mosur Ravishankar. PLUS: A Distributed Shared-Memory System. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 115–124, 1990.
- [BR92] Bob Boothe and Abhiram Ranade. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 214–223, May 1992.
- [C+88] R. C. Covington et al. The Rice Parallel Processing Testbed. In *Proc. 1988 ACM SIGMETRICS*, pages 4–11, 1988.
- [CF78] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.
- [CGL92] David E. Culler, Michial Gunter, and James C. Lee. Analysis of Multithreaded Microprocessors under Multiprogramming. Technical Report 92/687, Computer Science Division, University of California, Berkeley, 1992.
- [CSS+91] 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 *ASPLOS-IV Proceedings*, pages 164–175, 1991.

- [Dal90] William J. Dally. Virtual-Channel Flow Control. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 60–68, 1990.
- [Del91] Chrysanthos N. Dellarocas. A High-Performance Retargetable Simulator for Parallel Architectures. Technical Report MIT/LCS/TR-505, Massachusetts Institute of Technology, June 1991.
- [DGH91] Helen Davis, Stephan R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing using Tango. In *Proc. 1991 Int. Conf. on Parallel Processing*, pages II 99–107, 1991.
- [DK92] Susan R. Dickey and Richard Kenner. Hardware Combining and Scalability. In *Proc. of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 296–305, 1992.
- [DM93] Jack Dongarra and Hans Meuer. Top500 computers - a new competition. posted to comp.parallel, May 1993.
- [DRPS87] F. Darema-Rogers, G. F. Pfister, and K. So. Memory Access Patterns of Parallel Scientific Programs. In *Proc. 1987 ACM SIGMETRICS*, pages 46–58, 1987.
- [DSB88] Michel Dubois, Christoph Scheurich, and Favé A. Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessors. *Computer*, pages 9–21, February 1988.
- [Enc87] Encore Computer Corporation. *Multimaz Technical Summary*, 1987.
- [FP91] Matthew K. Farrens and Andrew R. Pleszkun. Strategies for Achieving Improved Processor Throughput. In *The 18th Annual Int. Symp. on Computer Architecture*, pages 362–369, 1991.
- [Fuj88] Fujitsu. *SPARC MB86901 (S-25) High Performance 32-Bit RISC Processor*, September 1988.
- [GGK<sup>+</sup>82] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer — Designing a MIMD, Shared-Memory Parallel Machine. In *Conf. Proc. of the 9th Annual Symposium on Computer Architecture*, pages 27–42, 1982.



- [GHG<sup>+</sup>91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *The 18th Annual Int. Symp. on Computer Architecture*, pages 254–263, 1991.
- [GLL<sup>+</sup>90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 15–26, 1990.
- [GT90] Gary Graunke and Shreekanth Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *Computer*, pages 60–69, June 1990.
- [Gus88] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, pages 532–533, May 1988. Technical Note.
- [Gus92] David B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, pages 10–22, February 1992.
- [GVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *ASPLOS-III Proceedings*, pages 64–75, 1989.
- [HF88] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *The 15th Annual Int. Symp. on Computer Architecture*, pages 443–451, 1988.
- [Hig93] High Performance Fortran Forum. High Performance Fortran Language Specification. Technical Report CRPC-TR 92225, Center for Research on Parallel Computation at Rice University, May 1993.
- [HLRW92] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *ASPLOS-V Proceedings*, pages 262–273, 1992.
- [HLS92] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Low Contention Load Balancing on Large-Scale Multiprocessors. In *Proc. of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 219–227, 1992.

- [Kan89] Gerry Kane. *MIPS RISC Architecture*. Prentice Hall, 1989.
- [Ken92] Kendall Square Research Corporation. *Kendall Square Research Technical Summary*, 1992.
- [Kow85] Janusz S. Kowalik, editor. *Parallel MIMD computation : the HEP supercomputer and its applications*. MIT Press, 1985.
- [KS93] R. E. Kessler and J. L. Schwarzmeir. CRAY T3D: A new Dimension for Cray Research. In *COMPCON SPRING '93*, pages 176–181, 1993.
- [LAD+92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, et al. The Network Architecture of the Connection Machine CM-5. In *Proc. of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.
- [Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, pages 690–691, September 1979.
- [Lar90] James R. Larus. SPIM S20: A MIPS R2000 Simulator. Technical report, C. S. Dept., University of Wisconsin-Madison, 1990.
- [Lei85] C. E. Leiserson. Fat-Trees: Universal Network for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, pages 892–901, October 1985.
- [LGH92] James Laudon, Anoop Gupta, and Mark Horowitz. Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors. Technical report, Stanford University, 1992. Technical Report CSL-TR-92-523.
- [LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 148–159, 1990.
- [LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolk-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *Computer*, pages 63–79, March 1992.

- [LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 92–103, 1992.
- [LM92] Thomas J. LeBlanc and Evangelos P. Markatis. Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. In *Fourth IEEE Symp. on Parallel and Distributed Processing*, 1992. to appear in SPDP 92.
- [LS91] Steven Lucco and Oliver Sharp. Parallel programming with coordination structures. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 197–208, Orlando, Florida, January 1991.
- [LT88] Tom Lovett and Shreekanth Thakkar. The Symmetry Multiprocessor System. In *Proc. 1988 Int. Conf. on Parallel Processing Vol. 1 Architecture*, pages 303–310, 1988.
- [LYL87] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Data Prefetching in Shared Memory Multiprocessors. In *Proc. 1987 Int. Conf. on Parallel Processing*, pages 28–31, 1987.
- [Man92] Daniel Mann. UNix and the Am29000 Microprocessor. *IEEE Micro*, pages 23–31, February 1992.
- [Mar87] Don Marsh. UgRay - An Efficient Ray-Tracing Renderer for UniGrafix. Master's thesis, U.C. Berkeley, May 1987. Report No. UCB/CSD 87/360.
- [MCS91] John M Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, pages 21–65, February 1991.
- [MIP86] MIPS Computer Systems. *MIPS language programmer's guide*, 1986.
- [MIP91] MIPS Computer Systems, Inc. *MIPS R4000 Microprocessor User's Manual*, 1991.
- [ML92] Evangelos P. Markatos and Thomas J. LeBlanc. Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance. Technical Report 420, University of Rochester, May 1992.

- [MPS87] Theodore E. Mankovitch, Val Popescu, and Herbert Sullivan. CHoPP Principles of Operation. In *Proc. Supercomputing '87 Vol. I*, pages 2–10, 1987.
- [NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, August 1991.
- [NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. \*T: A Multithreaded Massively Parallel Architecture. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 156–167, 1992.
- [O'K89] Brian W. O'Krafka. An Empirical Study of Three Hardware Cache Consistency Schemes for Large Shared Memory Multiprocessors. Technical report, Electronics Research Laboratory, University of California, Berkeley, May 1989. Tech Report UCB/ERL M89/62.
- [O'K92] Brian Walter O'Krafka. *Design and Evaluation of Directory-Based Cache Coherence Systems*. PhD thesis, U. C. Berkeley, January 1992.
- [ON90] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 138–147, 1990.
- [Ost89] Anita Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems*. Prentice Hall, 1989.
- [PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: an Explicit Token-Store Architecture. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 82–91, 1990.
- [PGH<sup>+</sup>84] Steven A. Przybylski, Thomas R. Gross, John L. Hennessy, Norman P. Jouppi, and Christopher Rowen. Organization and VLSI Implementation of MIPS. Technical Report 84-259, Computer Systems Laboratory, Stanford, 1984.
- [PH90] David A. Patterson and John L. Hennessy. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.
- [PN85] G. F. Pfister and V. A. Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. In *Proc. 1985 Int. Parallel Processing Conf*, pages 790–797, 1985.

- [Ran89] Abhiram Gorakhanath Ranade. *Fluent Parallel Computation*. PhD thesis, Yale University, May 1989.
- [RL92] Anne Rogers and Kai Li. Software Support for Speculative Loads. In *ASPLOS-V Proceedings*, pages 38–50, 1992.
- [RR93] M. T. Raghunath and Abhiram Ranade. Customizing Interconnection Networks to Suit Packaging Hierarchies. Technical Report UCB/CSD-93-725, Computer Science Division, University of California, Berkeley, CA 94720, January 1993.
- [RT86] R. Rettberg and R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *Communications of the ACM*, 29(12):1202–1212, Dec 1986.
- [SBC91] Rafael H. Saavedra-Barrera and David E. Culler. An Analytic Solution for a Markov Chain Modeling Multithreaded Execution. Technical Report 91/623, Computer Science Division, University of California, Berkeley, 1991.
- [SBCvE90] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *ACM Symp. Paral. Alg. Arch.*, July 1990.
- [SHG92] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of Hierarchical N-Body Techniques for Multiprocessor Architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.
- [Smi82] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.
- [Smi92] Burton Smith, 1992. personal communication.
- [SWG92] Jaswinder Pal Singh, Wolk-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, pages 5–44, March 1992.
- [TD91] Manu Thapar and Bruce Delagi. Cache Coherence for Large Scale Shared Memory Multiprocessors. *Computer Architecture News*, pages 114–119, March 1991.

- [TKB92] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *Computer*, pages 10–19, August 1992.
- [vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 256–266, 1992.
- [Waw91] John Wawrzynek, 1991. personal communication.
- [WG89] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *The 16th Annual Int. Symp. on Computer Architecture*, pages 273–280, 1989.
- [Wol89] Ronald W. Wolff. *Stochastic Modeling and the Theory of Queues*. Prentice Hall Inc., 1989.
- [YTL86] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-spot Addressing in Large-Scale Multiprocessors. In *Int. Conf. on Parallel Processing*, pages 51–58, 1986.

## Appendix A

# Distribution Function Histograms

Normally, distribution functions are shown either as cumulative distribution functions or as histograms, where both axis have linear scales. These turned out to be inadequate for showing the properties of the distributions which arose in this research, so as an alternative, we have developed a new type of histogram that represents data primarily by area rather height.

### A.1 Area Proportional to Value

Figure A.1 shows an example of our new histogram with various data points ranging from 50% to 0.05% of the total data.<sup>1</sup> The data points are represented as two-dimensional triangular piles whose area is proportional to their value. You can think of

<sup>1</sup>Please excuse the fact that the piles do not sum to exactly 100% in this example.

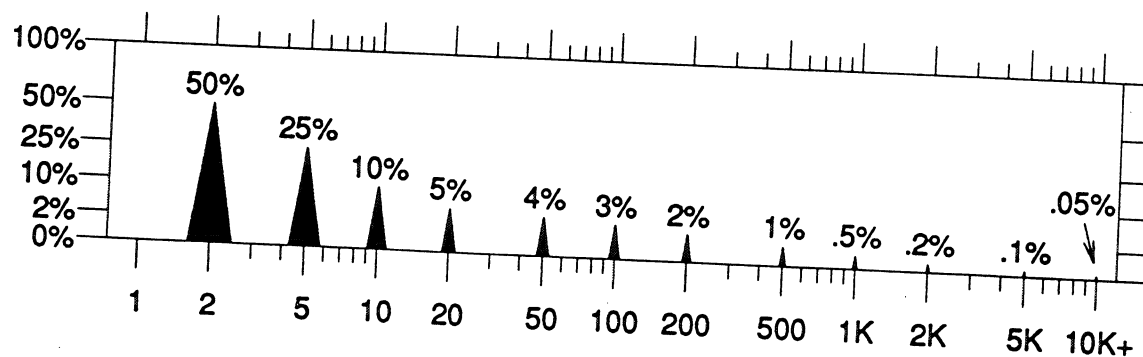


Figure A.1: Example histogram showing piles of various sizes.

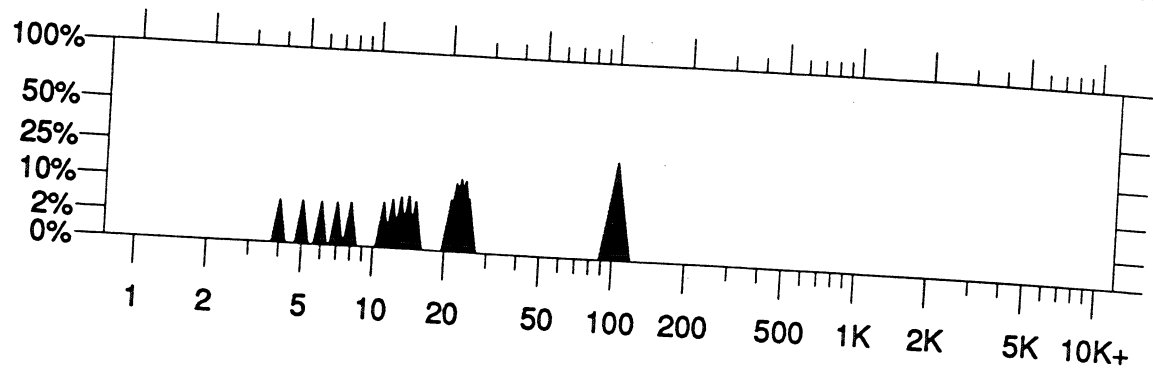


Figure A.2: Example histogram showing aggregation of adjoining piles.

these as piles of sand, where the proportion of sand corresponding to each data point was dropped from above that point. The triangular shape of the piles comes from the critical slope of the sand. The height of a pile will be equal to the square root of its area (value), and thus a square root scale is marked on the vertical axis. The advantage of representing data by area rather than height is that values ranging over three orders of magnitude can all be distinguished from each other, yet there is no distortion as there would have been if we had tried to use a logarithmic vertical axis. For some distributions that come up, it is important to see contributions of less than one percent because if they occur at large values, they can have a large impact on the mean.

## A.2 Aggregation of Adjoining Piles

Another important property of these histograms is the ability to combine nearby piles together when appropriate. For example, if 1% of the run-lengths were spread out evenly over the range from 2000 to 2100, each point in this range would have a value of less than 0.01%, and the individual points would be indistinguishably small. Together, however, they constitute a sizable amount and should definitely appear as such. Since the points from 2000 to 2100 are all so close together on the logarithmic scale, it is appropriate to draw just a single pile constituting 1% which is centered over the range 2000 to 2100.

Using the sand analogy, if several piles are close enough together that they would overlap, the sand should simply pile up higher and the piles should start to merge together. Figure A.2 shows an example of this merging. There are four clusters of five piles each, and each pile constitutes 5% of the data. The first cluster is the points {4,5,6,7,8}, and these



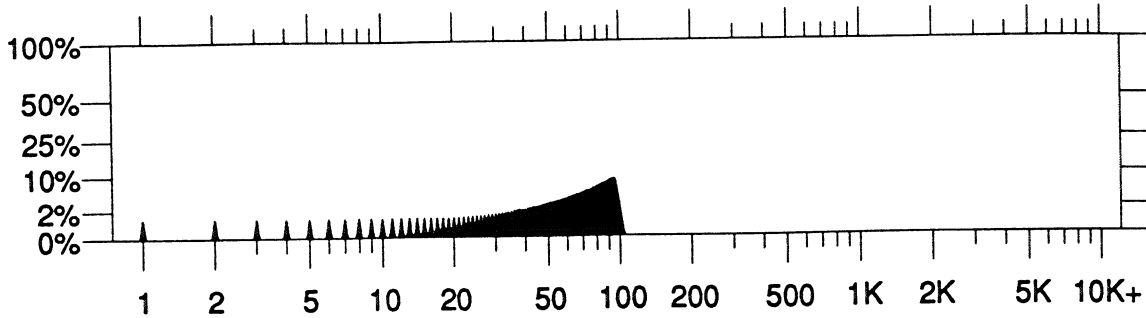


Figure A.3: Example histogram showing a uniform distribution over the domain  $[1, 100]$ .

points are far enough apart on the logarithmic scale that the piles remain independent. The second cluster of points is  $\{11, 12, 13, 14, 15\}$ . These piles are starting to merge, but are still distinguishable. The third cluster of points is  $\{21, 22, 23, 24, 25\}$ , and these are close enough together that they have almost merged into a single pile. Finally, the fourth cluster of points at  $\{101, 102, 103, 104, 105\}$  are so close together that the merged pile looks identical to a single pile of 25%. The important point is that each of the four piles comprises a total of 25% and uses the same amount of ink on the page. Where there is room to distinguish the individual components, this is done, and when there is not enough room, the pile are aggregated into a larger pile.

### A.3 Mismatch with Old Intuition

Unfortunately, if we are familiar with the look of a distribution when plotted on linear axis, it will appear distorted when plotted in this new format. A strong example of this is shown in Figure A.3. Here we show a uniform distribution where each point from 1 to 100 has 1% of the run-lengths. On linear axis this would appear flat. But in our new format, the tighter spacing at the high end does not allow enough room to show the many small piles independently. These adjoining piles are aggregated together and the graph rises.

Despite this mismatch with our old intuition, in most cases these graphs provide a compact and clear understanding of the distributions that we will see in this dissertation.

## Appendix B

# Simulator

### B.1 Introduction

Simulation is an essential tool in the process of computer design. While the speed of simulation has always been a concern, it is of critical concern when simulating parallel machines because of the increased computational power of these machines. The arithmetic is obvious: simulating one second of execution of a one MIP uni-processor requires simulating one million instructions, but simulating one second of execution of a thousand processor parallel machine requires simulating one billion instructions. Most simulation based research reports being limited in scope and accuracy by the speed of their simulators[BR92, GHG<sup>+</sup>91, ON90]. Faster simulators allow larger and more realistic simulations to be performed and help speed up the experimental process by allowing more rapid feedback of simulation results.

Our simulation system, **FAST** (Fast Accurate Simulation Tool), has a simulation slowdown ranging from 10 to 100. This *slowdown* factor is the average number of cycles it takes to simulate a single cycle of execution for a single processor. It varies based on the application program being simulated. Applications with more frequent references to shared memory interact with the simulator more frequently and therefore take longer to simulate. Comparable simulation systems such as that of O'Krafska[O'K89] or Tango[DGH91] have reported slowdowns of 2,000 and 500-6,000 respectively.

FAST was developed for the purpose of studying large shared memory multiprocessors with hundreds or thousands of processors, and to run real applications on these simulated machines. To support our simulation studies of such large systems, we needed a

simulator that was orders of magnitude faster than the other simulators that were available at the time of its development.

The technique of *execution driven simulation*[C+88] is the foundation of FAST. We are not concerned with the simulation of a new instruction set, but rather we are concerned with higher level aspects of the simulated machine. Because of this, we can accept the instruction set of the host machine on which we are performing our simulations. This allows us to directly execute most instructions instead of spending hundreds of cycles to interpret each instruction individually[Lar90]. The assembly code of the application program is augmented with additional instructions which keep track of a thread's execution time and return control to the simulator at special events such as references to shared memory. The net result is that most instructions are directly executed in a single cycle, and only the small fraction of instructions which interact with the rest of the system need to be simulated.

In this research we have extended the idea of execution driven simulation with several new techniques that have allowed us to build a simulator that is both faster and more accurate than previous comparable simulators.

### B.1.1 Overview

The remainder of this appendix is broken into five sections. Section B.2 discusses several previous simulators and their tradeoffs in performance, accuracy, and other concerns. Section B.3 presents an overview of FAST. Section B.4 explains our extensions to the ideas of execution driven simulation. Section B.5 reports performance results. And section B.6 summarizes and suggests directions for future research.

## B.2 Previous Simulators and Tradeoffs

There have been an enormous number of simulation systems written for various purposes. Here we focus on a few recent simulators that have all been designed for basically the same purpose: simulating large shared memory multiprocessors at the instruction level.

We compare their performance in terms of their slowdown factors, and we also look at two aspects of accuracy. One is the degree to which instruction timings reflect that of an actual processor, and the second is the degree to which shared memory references are interleaved and simulated in an accurate global order.

### B.2.1 Cycle-by-Cycle Simulators

The most straight forward type of simulator to build is one that cycles through the parallel processors, simulating one instruction at a time from each of the processors. Two examples are the simulator by O'Krafka [O'K89], which we are more familiar with since this was done at Berkeley, and ASIM (refer to the description in [Del91]) developed at MIT as part of the Alewife project. These simulators are slow because they are essentially assembly language interpreters. The reported slowdown factor for O'Krafka's simulator is 2,000, and for ASIM it is reported as ranging from 200–5,000. Cycle-by-cycle simulators are accurate in interleaving global events since they simulate the entire machine one cycle at a time, but they may be inaccurate in instruction timing (as is O'Krafka's simulator) because it is complex and time consuming to accurately model the processor's pipeline.

The performance of these cycle-by-cycle simulators is dominated by instruction interpretation since this is done for every single cycle of the executed program. Interesting events, like shared memory references, occur less frequently.

### B.2.2 Execution Driven Simulators

Execution driven simulation can be substantially faster than a cycle-by-cycle simulator because it eliminates the instruction interpretation portion of the simulator. Instead, control is handed over to the augmented program which executes for several cycles before encountering an event of interest and returning control to the simulator. The simulated processor has now advanced its private clock past those of other simulated processors. Accurate event interleaving dictates that the event should not be processed immediately, but rather it must be scheduled and executed once the entire global state has advanced to the event's time step. This means that instead of cycling between the simulated processors on a cycle by cycle basis, it is sufficient to cycle between them at each event (as long as the events are then queued and later executed at their proper times).

The Tango simulator [DGH91] developed at Stanford is an execution driven simulator. It is based on Unix shared memory and uses Unix context switches in order to switch from executing one processor to another. These heavy weight context switches however require thousands of cycles, and thus they slow the simulator substantially if it switches at every event in order to accurately interleave them. For accurate simulations they report slowdown factors ranging from 500 to 6000. Because of this large cost of context switching,

they provide an option to tradeoff accuracy for faster execution by letting the individual processor clocks get out of sync and not trying to accurately interleave the shared memory references. They have recently rewritten their simulator to use a light weight thread package, which should significantly reduce the magnitude of their context switch overhead problem.

The Proteus simulator developed at MIT[BDCW91, Del91] is another execution driven simulator. It does use a light weight thread package, and is substantially faster than Tango. They report typical slowdown factors ranging from 35 to 100. However they have a substantial accuracy problem in their instruction timing because they do not apply code augmentation at a consistent low level. They replace shared memory references in the C source code with calls to the simulation routines (and optionally also insert statistics gathering calls.) They then compile this modified code and apply code augmentation for timing on the assembly language. Because each shared reference (which should be just a single instruction) is replaced with a procedure call, the compiler optimizations that can be applied and the object code produced are substantially changed from that which would have been produced if the original code were compiled directly. In fact, their good performance is partially due to the fact that their insertion of procedure calls causes the compiler to save away important registers, and thus allows them to “exploit ‘partial’ context switches” in which they only save a limited amount of the register file. This is good for performance, but bad for timing accuracy.

### B.2.3 Tradeoffs

We have identified the following five tradeoffs in simulator design:

**Performance:** Execution driven simulation is the most important factor in building a fast simulator because otherwise the interpretation of individual instructions is the dominant cost. The next most important factor is fast context switching between the simulated processors because frequent context switching is required to accurately order global events.

**Accuracy:** Performing all code augmentation at the assembly language level is necessary for accurate instruction timing. Any source code modifications that change the code generated by the compiler affect the compiler’s optimization ability and the thus accuracy of instruction level timing. Switching between simulated processors at all

globals events is required in order to obtain a correct global ordering. If context switching is expensive, then the simulator writer or user is tempted to trade accuracy for performance by context switching less often.

**Source Alteration:** Ideally the source code should be compiled and optimized in its original form as it would be written for a shared memory multiprocessor. However, all of these simulators require some source changes. Proteus is the most egregious and requires new operators be used for all shared memory references. O'Krafka's simulator and Tango both disallow static shared variables, and thus all such variable must be allocated dynamically and referenced indirectly through pointers. FAST only requires minor syntactic changes<sup>1</sup> that have no affect on the instructions generated by the compiler.

**Modularity:** All of the simulators have similar modularity. Each allows selecting and mixing different modules for different aspects of the machine: such as the cache and the interconnection network. Normally this is done by linking the modules together, but Tango also has the option (at substantial performance cost) of using distinct Unix processes.

**Portability:** Portability is poor for all of these systems because they are tied to the instruction set that they are designed for. Direct execution simulators must be run on that specific type of machine, but cycle-by-cycle simulators, since they are interpreters, can use cross-compiled applications and be run on any machine. Porting execution driven simulators to a new machine involves changing the code augmentation to understand the new machine's instruction set. The actual simulators are all written in high level languages and should presumably be portable.

Based on an understanding of these tradeoffs, we have built our FAST simulation system so that it is faster, more accurate, and uses less mutative source alterations. It has similar modularity and portability as in the other simulators discussed.

## B.3 Simulator

---

<sup>1</sup>Most of our applications were originally written for the Sequent[Ost89]. Their syntax for declaring a shared variable is: `shared int x;`. Our syntax is: `int shared_x;`. All uses of the variable `x` are also changed to `shared_x`. These changes allow shared variables in the object file to be identified by using symbol table information that is normally used for linking.

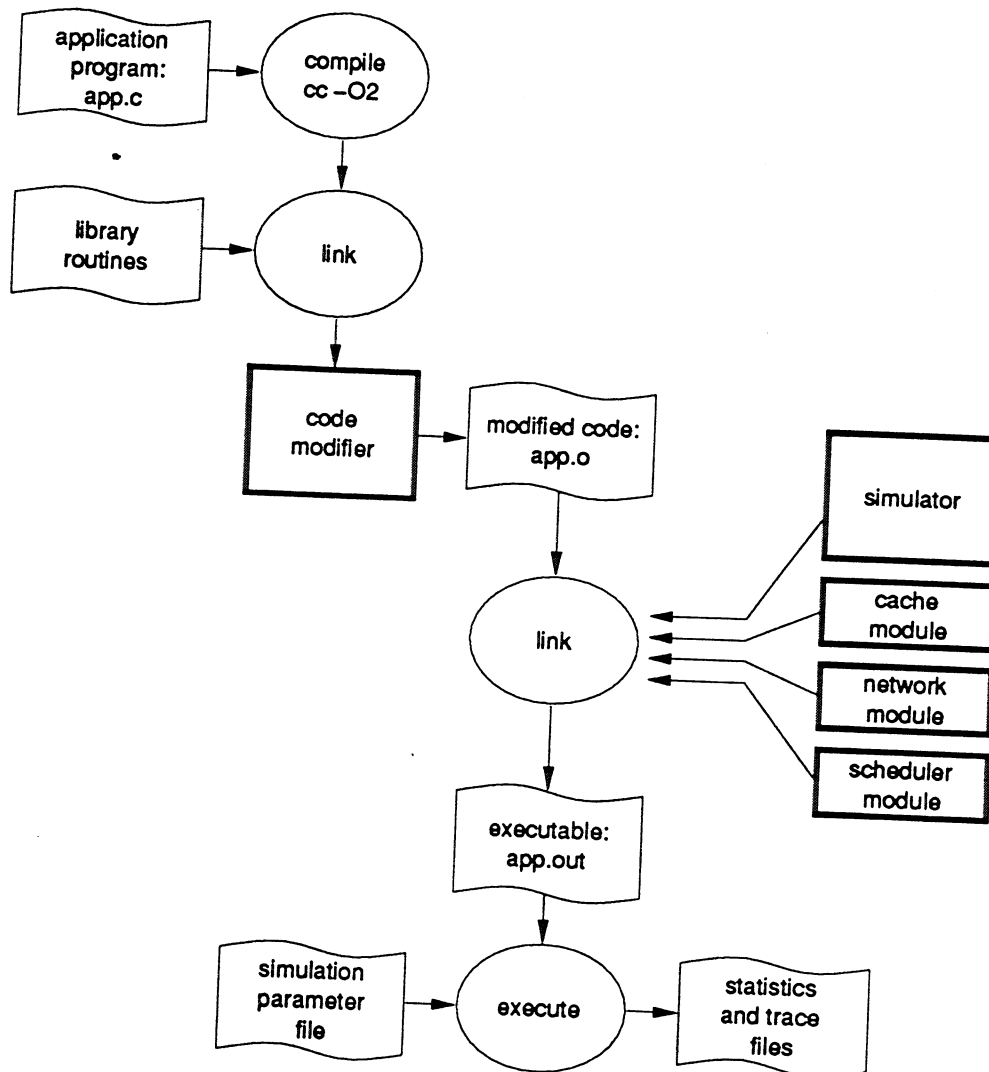


Figure B.1: Diagram of using FAST.

Figure B.1 shows a diagram of using FAST. First the application program to be simulated is compiled with full optimization just as it would be for a real parallel processor, and then it is linked with any libraries that it uses, such as math routines.

The linked object code module is then read into the code modifier which performs the various code augmentations (which will be discussed in the next section). It is important that augmentation be done on library functions since some applications use these extensively. System calls are not handled, but these usually do not occur in the parallel computation phases of the parallel applications that we have studied.

The modified code is then linked with the simulator and selected modules that simulate the caches, network, and scheduler. A large number of these modules have been written, and they can be selected based on what is of interest to the user. For caching there are modules for various cache configurations and protocols, or for no caching at all. For networks the simulator is usually used with a simple constant time network approximation, but it has also been used with a detailed simulator of packet switched networks. The scheduler module is used for multithreading studies and implements simple scheduling policies such as FIFO, or more complex policies like priority scheduling or timeouts.

The single executable file produced includes the simulator, the various modules, and the modified application code. When it is run, the simulator starts first. It reads in a simulation parameter file that specifies the number of processors, level of multithreading, network latency, and other parameters. It then calls initialization routines for the various modules, and then starts up and manages the execution driven simulation of the application program.

The core of the simulator is a simple time wheel scheduler. This is just a linear array with one slot per time step (modulo the array size), where each slot points to a linked list of events that will occur at that time step. The simulator operates by removing an event at the current time step, simulating it (using execution driven simulation), and then placing the resulting event into the proper slot to be executed in the future. This is very efficient since there is no polling to test for ready events. For simulations of large parallel machines, there are so many events that typically every slot has one or more events in it. The average cost of scheduling an event is thus very small.



## B.4 Code Augmentation

Code augmentation is the process of taking an original piece of code and adding to it and/or modifying it so that it can perform additional functions. Traditionally it has been used for the following three purposes:

**Time Counting:** Instructions are added to the each basic block so that when that block is executed, the extra instructions increment a time counter with an amount corresponding to the number of cycles required for the processor to execute the original basic block. This is the basic code augmentation that is used in all execution driven simulators.

**Statistics Gathering:** Instructions are added to gather statistics such as counts of the number of times that certain pieces of code are executed. This is the basis of execution driven profilers, such as the MIPS *pixie* program[MIP86].

**Event Call-Outs:** At special events, such as shared memory references, code is inserted to call out to the simulator in order to let the simulator regain control and process the event. This is used in a simplified form when debuggers create breakpoints by replacing the instruction at the breakpoint with a trap instruction.

In this section we extend the idea of augmentation with several new uses:

**In-line Context Switching:** The augmented code typically runs for just a small number cycles before reaching an event and returning control to the simulator. During this execution only a small subset of the register file is ever accessed, and therefore it is wasteful to actually load and store the entire register set. We use code augmentation to load and store register values at basic block boundaries so that only the used and modified registers are loaded and stored.

**Reference Indirection:** For a single threaded program, which the compiler thinks it is compiling, static local variables are assigned to fixed memory addresses. However, for a parallel program, each thread needs its own copy of these variables. Our code augmenter converts these references into indirect references into the executing thread's context block which contains the thread's local state: register values, local variables, and stack.

**Dynamic Reference Discrimination:** We suggested in Section 2.1.3 that a compiler, with proper language support, should be able to statically identify all memory accesses as going to either local or shared memory. Since we do not have languages and compilers that support this, we have added code augmentation to check address ranges at execution time and determine if a pointer is to shared or local memory. Optionally, this reference classification information can be collected as a trace file on the first run of an application and then fed back into the code modifier to do complete static classification.<sup>2</sup>

**Re-Optimization:** During our studies of multithreading we found it important to group shared memory load instruction together. We implemented this within the code modifier by reordering instruction and percolating shared memory load instructions up towards the tops of basic blocks.

**Extended Instruction Sets:** For the most part we accepted the instruction set of the processor on which simulations were being executed: the MIPS R3000[Kan89]. However we did want to add a number of new instructions such as: double word load and stores, local and shared memory versions of all loads and stores, an explicit thread switch instruction, fetch-and-add, and other special synchronization instructions. These were all added by having the code modifier convert these into calls to special simulator routines.

**Virtual Registers:** One of the most useful new code augmentations is virtualization of the register file. This simplified implementation of the other code augmentations because it eliminated concerns about remapping registers. This will be discussed more fully at the end of this section.

#### B.4.1 An Example

Figure B.2 shows an example of code augmentation for a small code fragment which will be used to demonstrate several of the code augmentations described above. The original assembly language instructions are shown in part (a); the modified code is shown in part(b).<sup>3</sup> These instructions were generated by the compilation of the expression

<sup>2</sup>This accurate static classification is required for our re-optimization of the code.

<sup>3</sup>The instruction set is approximately that of the MIPS R3000[Kan89], but it has been simplified slightly to make the example clearer.

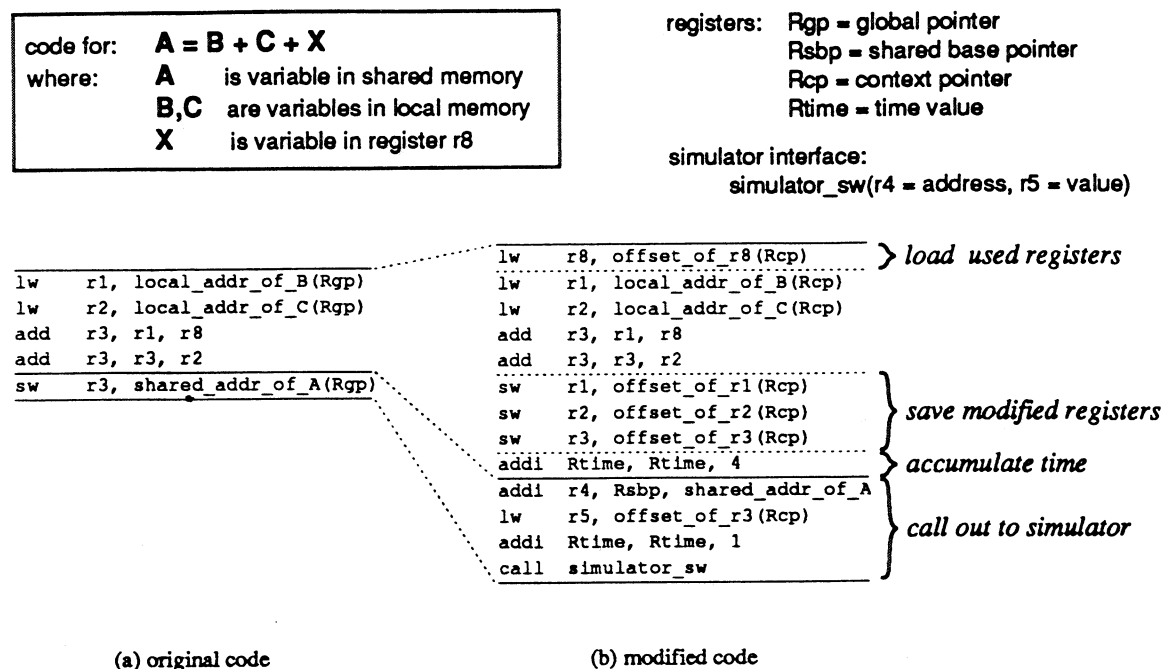


Figure B.2: Example of code augmentation

$A = B + C + X$ , where the variables  $B$  and  $C$  will be loaded from local memory, the variable  $X$  is already in register  $r8$ , and the result  $A$  will be stored in shared memory. Assume for this example that this expression by itself forms a basic block. Basic blocks are the granularity at which we perform analysis and code augmentation, and thus this small basic block can serve as a complete example.

The first step is to identify which instructions can be directly executed by the host processor and which instructions will require call-outs to the simulator. In this example the last instruction references shared memory and thus will be replaced with a call-out. The other four instructions are local to the processor and can be directly executed. For ease of manipulation, the call-out instruction is isolated into its own basic block, as indicated by the horizontal lines separating the instructions.

The second step is to calculate the timing of the basic blocks. The first block has four instructions and takes four cycles. The second block has one instruction and takes one cycle<sup>4</sup>. The timing of each basic block is computed statically and is used in the inserted

<sup>4</sup>In general determining accurate timing is somewhat more complicated because the processor pipeline must be modeled. Usually looking just within a basic block is adequate, but sometimes long latency floating point operations continue executing past the end of a basic block and affect the timing of subsequent blocks.

instructions which accumulate the running execution time in register **Rtime**.

The third step is reference indirection. The loads of local variables **B** and **C** are originally relative to the global pointer (register **Rgp**). These are changed to be thread relative by indexing off of the thread context pointer (register **Rcp**).<sup>5</sup>

Step four involves adding code for in-line context switching. In our implementation, we maintain the invariant condition that between basic blocks all register values should be correctly stored in the context block of the executing thread. In our system this context block is pointed to by the **Rcp** register, and thus register load and stores are relative to this pointer.

At the start of each basic block we insert code to load the registers whose values will be used. In the example, only the value in register **r8** is used. The registers **r1**, **r2** and **r3** also appear, but they do not need to be loaded since their original values are not used. At the end of each basic block we append code to store any registers whose values have been redefined. In the example this is **r1**, **r2** and **r3**.

This completes the code augmentation for the first basic block. The second basic block is the save word instruction (**sw**) that originally saved the value in register **r3** to an address in shared memory. It is replaced by a sequence of instructions which load parameters and then call-out to a simulation routine to perform the shared memory operation. The address and data values are loaded into the argument registers (**r4** and **r5**), and the time counter (**Rtime**) is incremented by 1 (the time taken by the original instruction). If the simulator finds that more time would be needed by this instruction, for instance if the memory network is clogged or there is a cache miss, the simulator would add the additional time.

This completes the code augmentation. The code has now been converted so that it is context block relative. The simulator can now switch threads by changing the context pointer and time counter and then jumping into the new thread to be executed.

---

If these subsequent blocks are selected by conditional branches, the exact timing will depend upon the branch paths taken at execution time. These cases are rare, and for our simulator we use timings based on the statically predicted most likely branch paths.

<sup>5</sup>Here reference indirection is simply changing from **Rgp** to **Rcp** and possibly changing the offset. It is more involved when the original reference is not relative to **Rgp**.

### B.4.2 Virtual Registers

The technique of in-line context switching usually leaves most register values in the context block, and this motivated the idea of virtualizing the register file. When register `r8` was loaded and later used in Figure B.2(b), it could have been loaded into any physical register as long as the register later used in the add instruction was also changed to the same register. Thus the *virtual* registers used in the original code need not be the same as the *physical* registers used in an expanded basic block. Different basic blocks could choose to use different physical registers to hold the virtual register `r8`.

The benefit of this is that we can now have more virtual registers than there are physical registers. For instance we have used virtual registers `Rtime`, `Rcp`, and `Rsbp` in our modified code. The mapping between virtual and physical registers is possible as long as each individual basic block does not use more virtual registers than there are physical registers to map into. Mapping problems are rare and occur only for large basic blocks, and they are easily handled by splitting these large blocks into multiple smaller blocks.

This virtualization of the register file actually simplifies other code augmentations. For instance in the old style of code augmentation, some specific physical register, say `r30`, is used for time counting. Thus wherever `r30` is used in the original code, the code must be modified to work around the usurpation of this register.

Virtual register have many potential uses. One example use was in a research project that tried to improve memory reference patterns by re-optimizing basic blocks in order to group together shared memory load instructions. This re-optimization needed a few extra temporary registers to allow reordering of instructions while still preserving all data dependencies, and these extra registers were made available as extra virtual registers.

## B.5 Performance

In this section we discuss three aspects of the performance of our simulator: the cost of in-line context switching, the slowdown factors of basic simulations, and the affects on slowdown when simulating multithreading or caching.

### B.5.1 Cost of In-line Context Switching

Application	Description	Context switch cost		Average interval between switches	Amortized cost per instruction
		switch in	switch out		
sieve	finds primes	9.8	7.9	7.0	2.5
blkmat	blocked matrix multiply	47.7	50.3	48.0	2.0
sor	solves Laplace's equation	8.5	5.5	4.2	3.3
ugray	ray tracing renderer	11.8	9.1	10.1	2.1
water	system of water molecules	27.7	22.2	33.1	1.5
locus	standard cell router	8.0	5.2	4.0	3.3
mp3d	rarefied hypersonic flow	8.1	6.3	4.7	3.1

Table B.1: Context Switch Costs

Table B.1 shows the effectiveness of in-line context switching. It gives the context switch frequency and the average context switch costs for the applications that we have used in our simulation studies.

The *switch in* cost listed in the table is the average number of registers loaded per context switch into the application from the simulator. The *switch out* cost is the average number of registers saved per context switch from the application out to the simulator. Recall that these register loads and stores do not all occur at the points of context switching between the simulator and threads, but are spread among the prefixes and suffixes of the sequence of basic blocks executed between context switches. Also included in these context switch costs are the overheads incurred by the simulator in saving and restoring reserved registers such as the program counter, time counter, stack pointer and context pointer.

The column labeled *average interval between switches* shows the average number of simulated cycles between context switches. For those applications that context switch most frequently, the context switch cost is less than 10 cycles. The *locus* program, for example, accesses shared memory very frequently and thus context switches at an average rate of once every four cycles. The average cost of these context switches is 8.0 cycles to switch in and 5.2 cycles to switch out. In all cases, the context switch cost is less than the size of the register set<sup>6</sup>. In comparison, the light-weight thread package used in Proteus[Del91] loads and stores the entire register set and takes 135 cycles per context switch.

In our system, the cost of context switching is roughly proportional to the frequency of occurrence. The longer an application executes, the more registers it is likely to

<sup>6</sup>On a Mips processor there are 29 integer, 32 floating point and 3 special purpose registers in the usable register set.

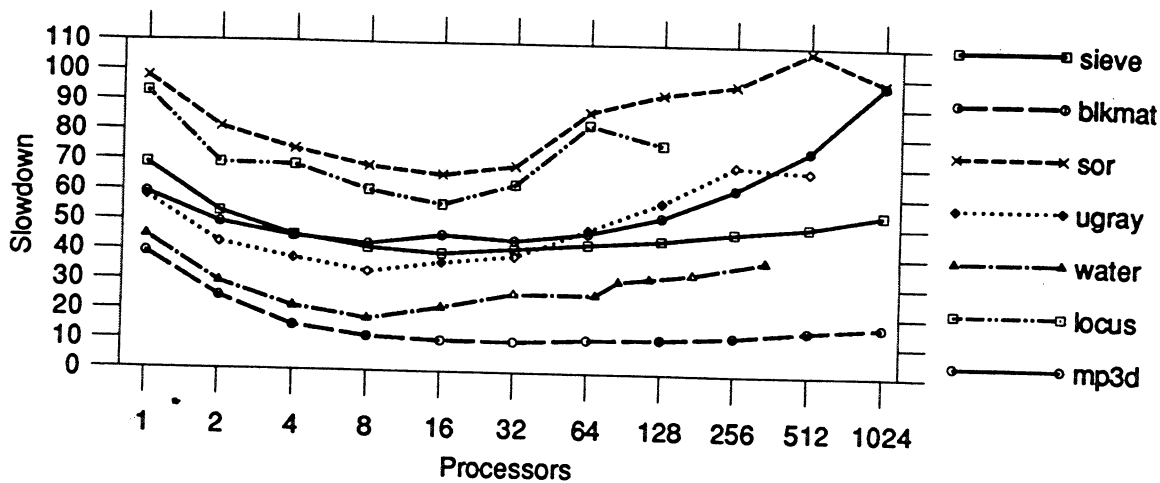


Figure B.3: Simulation slowdown

use. The `blkmat` and `water` applications, for example, context switch less frequently than the other applications and their average context switch costs are higher. However since they do not context switch as frequently, the higher context switch costs are amortized over a longer period. Overall, the total context switch overhead ranges from 2 to 3 cycles per simulated cycle.

### B.5.2 Slowdowns Factors for Basic Simulations

Figure B.3 shows the performance of the FAST simulator on the various benchmark applications. Results are shown with the number of processors varied from 1 to 1024. The slowdown factors shown in this graph are the number of cycles taken to simulate a single cycle of a single thread. Since most instructions are directly executed and the context switching cost has been reduced to just 2 to 3 cycles per simulated cycle, one might expect slowdown factors of 3 or 4. The slowdowns are larger because of the remaining overhead which comes from the scheduling mechanism within the simulator, the simulation of shared references, the memory simulator, and statistics gathering. For this graph the memory model is a simple ideal memory that has 0 latency and no contention.

Two interesting trends can be observed from this graph. First, the slowdowns vary for different programs. Programs such as `blkmat` and `water` have typical slowdowns from 10 to 30, while programs such as `locus` and `sor` have typical slowdowns from 60 to 100. The difference comes from the different frequencies at which the applications interact with

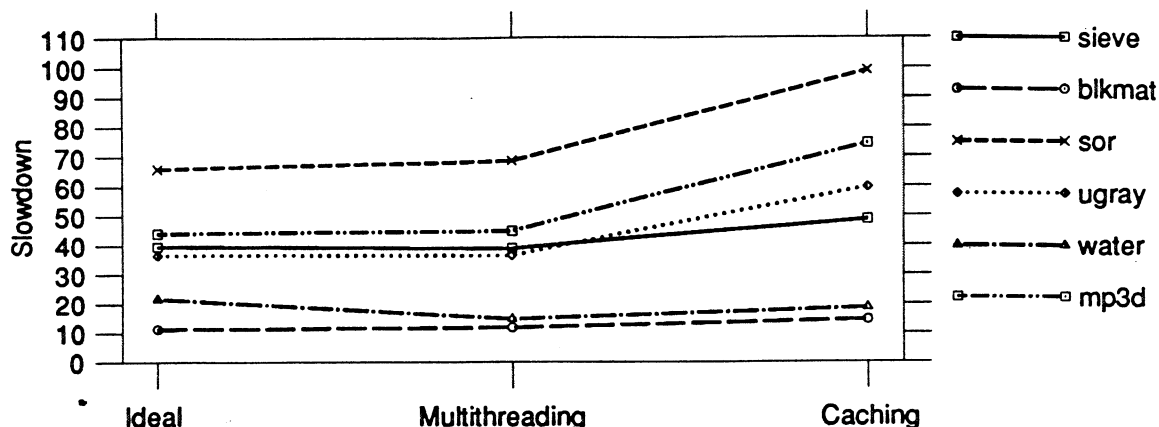


Figure B.4: Simulation slowdowns under different configurations.

the simulator. *Sor* and *locus* had context switches every 4 cycles compared to *blkmat* and *water* which have context switches only every 30 to 50 cycles and thus require much less scheduling by the simulator. The cost of simulated events is amortized over a larger number of instructions, and thus the overall slowdown factors for *blkmat* and *water* are lower than those for the other applications.

The second interesting trend is that as the number of processors is increased, the slowdown factor initially drops and then slowly rises. The initial decrease in slowdown is due to the time wheel algorithm used to schedule threads and events. It works best when there are many processors and thus there are many events per cycle. The later increase in the slowdown factor occurs because the applications use more synchronization operations as the number of processors is increased. Synchronization operations, especially spinning on locks or barriers, involve many shared accesses and thus increase the work of the simulator.

### B.5.3 Multithreading and Caching

FAST was designed in a modular fashion and can be configured to perform a wide variety of different simulations depending upon what is of interest to the researcher conducting the simulation studies. The main uses of the simulator have been for studies of multithreading under long memory latencies and for performance studies of cache coherency protocols.

Figure B.4 shows the performance of the simulator under three configurations: the *ideal* case which has 0 latency, the *multithreading* case which has 200 cycle latency and



several threads per processor, and the *caching* case which uses a cache simulator of the Censier and Feautrier[CF78] directory based cache coherence protocol. The ideal case and the multithreading case have roughly the same performance. This occurs because studying multithreading was one of the primary intended uses of FAST, and thus multithreading support was built in from the start. Single threaded execution is simply a special case of multithreading in which there is just one thread per processor. The cache simulator typically takes hundreds of cycles per reference to check and manipulate the caches' states, and this extra overhead slows the simulations. The change in performance is moderated by the fact that the cache simulation cost is amortized over the total number of simulated cycles.

## B.6 Summary and Future Research

We have used FAST to perform a large number of architectural simulations. Its fast speed has allowed us to simulate larger problems and larger machines than would have been possible with previous comparable simulators. Execution driven simulation is the most important technique for obtaining high performance.

However speed is just one important aspect of FAST. By carefully understanding the tradeoffs in design choices, we have been able to build a simulator that is also more accurate than previous instruction level simulators. The most important point is that code augmentation must be applied at a low level since source code alterations can perturb the object code produced and thus the accuracy of instruction level timings. A second point is that accurate interleaving of global events requires frequent context switching between simulated processors, and thus fast context switching is desirable.

In building FAST, we have extended the idea of code augmentation into a number of new areas such as in-line context switching, re-optimization, extended instruction sets, and virtualization of the register file. These extensions have been important in making the right design tradeoffs so as to obtain both high performance and high accuracy, and in making a simulator that is flexible enough to be used for a large variety of experiments.

There are several possible directions for future research with FAST or similar simulators. First, since we are simulating a shared memory multiprocessor, it should be possible to speed up the simulator by executing it in parallel on today's small shared memory multiprocessors in order to simulate tomorrow's larger machines. The main problem that

will arise is synchronizing and correctly interleaving the concurrent simulations of multiple processors.

Second, FAST would be a good foundation for a parallel program development and debugging system. Simulators are useful for debugging because they can reproduce identical timing races on subsequent runs. The Proteus[Del91] simulator provides a powerful monitoring facility by inserting monitoring code into the source code of applications, and we would like to see if similar mechanisms could be built without modifying the applications' source code.

Third, our new augmentation techniques of virtualizing the register file and extending the instruction set could be used along with a modified compiler to study various architectural changes such as larger register files or new instructions.