Exploiting Thread-Level Parallelism on Simultaneous Multithreaded Processors

by

Jack Lee-jay Lo

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of W ashington

1998

Approved by ______________________________
(Co-chairperson of Supervisory Committee)

____
(Co-chairperson of Supervisory Committee)

Program Authorized
to Offer Degree ______________________________

Date ______________________________

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature ______________________________

Date ______________________________

© Copyright 1998

Jack L. Lo

University of W ashington

Abstract

Exploiting Thread-Level Parallelism on Simultaneous Multithreaded Processors

by Jack Lee-jay Lo

Co-Chairpersons of the Supervisory Committee: Professor Susan J. Eggers
Professor Henry M. Levy
Department of Computer Science and Engineering

This dissertation investigates techniques for promoting constructive sharing and efficient utilization of processor resources on simultaneous multithreaded processors. Simultaneous multithreading is an architecture that permits multiple threads to issue multiple instructions to the functional units each cycle. By dynamically sharing processor resources both temporally and physically, simultaneous multithreading can avoid the resource partitioning and inflexibility that plague other parallel architectures.

While dynamic resource sharing enables SMT to achieve performance gains, it also has some potential drawbacks. With its simultaneous multi-thread instruction issue, simultaneous multithreading can place much greater stress on the shared on-chip hardware. This dissertation explores this issue of dynamic, inter-thread resource sharing.

First, we demonstrate the importance of dynamic resource sharing by comparing an SMT processor to both wide-issue superscalars and single-chip multiprocessors. Using several metrics to identify resource waste, we provide a detailed analysis that quantifies the costs of static partitioning on an MP, as well as the performance gain due to dynamic resource sharing on an SMT. Our experiments demonstrate that an SMT achieves a 54% performance edge over a single-chip multiprocessor with comparable execution

resources, because of its ability to adapt to variable levels of ILP and TLP. These experimental results attest to the importance of efficient and flexible resource sharing.

We then investigate the effects of thread-level parallelism on shared processor structures. Although thread-level parallelism can placed greater stress on shared resources, we have found that inter-thread interference frequently results from software that is unaware of the underlying SMT architecture. By adopting the proper software policies, however, threads can utilize shared resources in an SMT-conscious manner, minimizing or even eliminating interference. We propose compiler, operating system, and application-level techniques that enable threads to share resources more efficiently, thereby increasing the performance of SMT processors.

Table of Contents

List of Figures vi

List of Tables viii

Chapter 1: Introduction 1

1.1 Potential costs of shared resources 4

1.2 Scope and contributions of this dissertation 5

1.3 Organization of the dissertation 7

Chapter 2: Simultaneous multithreading 8

2.1 Base processor architecture 8

2.2 Extensions to support SMT 11

2.2.1 Instruction fetch unit 11

2.2.2 Register file and register renaming 11

2.2.3 Other required modifications 13

2.3 SMT synchronization 13

2.4 Chapter summary 14

Chapter 3: Related work 15

3.1 Coarse-grain multithreading 15

3.2 Fine-grain multithreading 16

3.3 Simultaneous multithreading 17

3.3.1 Dynamic partitioning 17

3.3.2 Partitioned multiple-thread issue architectures 20

3.4 Other parallel on-chip architectures 21
3.4.1 Single-chip multiprocessors 22
3.4.2 Compiler-dependent thread-level parallel architectures 22
3.4.3 Trace processors 23
3.4.4 Dataflow architectures 23
3.5 Chapter summary 24

Chapter 4: Converting thread-level parallelism to instruction-level parallelism 25

4.1 Simultaneous multithreading and single-chip multiprocessors 27
4.1.1 SMT hardware configuration 27
4.1.2 Single-chip multiprocessor hardware configurations 27
4.1.3 Synchronization mechanisms and memory hierarchy 29
4.1.4 Comparing SMT and MP 30
4.2 Methodology 30
4.2.1 Why parallel applications? 30
4.2.2 Workload. 31
4.2.3 Simulation framework 32
4.3 Exploiting parallelism. 34
4.3.1 SMT and multiprocessing performance results 34
4.3.2 Measuring MP inefficiency. 34
4.3.3 Exploiting ILP. 36
4.3.4 Exploiting TLP 41
4.3.5 Effectively using parallelism on an SMT processor 41
4.4 Placing these results in context 45

4.5 Related work . . . 47
4.5.1 Previous SMT work . . . 47
4.5.2 Single-chip multiprocessors . . . 50
4.5.3 Related SMT work . . . 51
4.6 Chapter summary . . . 52

Chapter 5: Tuning compiler optimizations for SMT **54**

5.1 Rethinking compiler optimizations . . . 55
5.1.1 Inter-thread data sharing . . . 55
5.1.2 Latency-hiding capabilities and the availability of instruction issue slots . . . 56
5.2 Methodology . . . 56
5.2.1 Compiler framework . . . 58
5.2.2 Simulation framework . . . 58
5.3 Loop iteration scheduling . . . 60
5.4 Software speculative execution . . . 66
5.5 Loop tiling . . . 70
5.6 Other compiler optimizations . . . 74
5.7 Compiler related work . . . 77
5.7.1 Loop iteration scheduling . . . 77
5.7.2 Code scheduling optimizations . . . 77
5.7.3 Optimizations for improving locality . . . 78
5.8 Chapter summary . . . 78

Chapter 6: Managing cache interference in database workloads **80**

6.1 Methodology . . . 81

6.1.1 SMT processor model . . . 81

6.1.2 Simulating database workloads. . . . 82

6.2 Database workload characterization . . . 85

6.2.1 OLTP workload characterization . . . 86

6.2.2 DSS workload characterization. . . . 91

6.2.3 Summary of workload characterization . . . 92

6.3 Multi-thread cache interference . . . 95

6.3.1 Misses in a database workload . . . 96

6.3.2 Page-mapping policies . . . 97

6.3.3 Application-level offsetting . . . 99

6.3.4 Constructive interference . . . 99

6.3.5 Summary of multi-thread cache interference . . . 100

6.4 SMT performance on database workloads . . . 101

6.4.1 The effect of I/O . . . 103

6.4.2 Architectural analysis of SMT performance gains . . . 103

6.5 Related work . . . 106

6.5.1 Database workload performance studies . . . 106

6.5.2 Cache interference in multithreaded processors . . . 108

6.6 Chapter summary . . . 110

Chapter 7: Software-directed register deallocation **112**

7.1 Methodology . . . 113

7.2 Register renaming 114
7.2.1 Register renaming hardware inefficiency 116
7.2.2 Register inefficiencies in SMT processors 120
7.3 Compiler support for dead-register deallocation 122
7.3.1 Five compiler-based solutions 122
7.3.2 Implementation issues. 125
7.3.3 Experimental results 127
7.4 Operating system support for dead-register deallocation 132
7.4.1 Implementation issues. 133
7.4.2 Experimental results 134
7.5 Related work 137
7.6 Chapter summary 139

Chapter 8: Conclusions **140**

8.1 Future work 141
8.1.1 Dynamic policies 142
8.1.2 Single-thread priority vs. overall throughput 142
8.1.3 Parallelization of sequential applications 143

References **144**

List of Figures

1.1 A comparison of issue slot utilization in various architectures 2

2.1 Organization of the dynamically-scheduled superscalar base processor 9

2.2 Comparison of the pipelines for a conventional superscalar processor and our SMT processor..10

2.3 Organization of the SMT processor used in this dissertation. 12

4.1 Speedups normalized to MP2.T1 35

4.2 Frequencies of partitioning inefficiencies for MP2.T2. 37

4.3 Frequencies of partitioning inefficiencies for MP4.T4. 38

4.4 MP2 and MP4 speedups vs. one-thread MP2 baseline 42

5.1 A blocked and cyclic loop distribution example...61

5.2 Speedups over one thread for blocked parallelization. 62

5.3 Speedup attained by cyclic over blocked parallelization. 65

5.4 Speedups of applications executing without software speculation over with speculation 69

5.5 Code for blocked and cyclic versions of a tiled loop nest 71

5.6 Tiling results with the larger memory subsystem and separate tiles/thread...........73

5.7 A comparison of blocked and cyclic tiling techniques for multiple threads..........75

5.8 Tiling performance of 8-thread mxm..76

6.1 Locality profiles for OLTP 89

6.2 Locality profiles for DSS 93

6.3 Comparison of throughput for various page-mapping schemes on a superscalar and 8-context SMT 102

6.4 SMT throughput with 16 server processes, bin-hopping and application-level offsetting 104

7.1 Register renaming hardware 115

7.2 Register renaming example....... 116

7.3 Logical register file configurations 121

7.4 Register file utilization with idle contexts 121

7.5 Register deallocation example 123

7.6 Register deallocation implementation example 126

7.7 Register deallocation performance for the large cache size....... 128

7.8 Register deallocation performance for the small cache size 129

7.9 Comparison of execution time with and without Free Register Bit for a range of register file sizes, on the large cache size 130

7.10 Comparison of execution time with and without Free Register Bit for a range of register file sizes, on the small cache size....... 131

7.11 Execution time for the larger memory hierarchy 135

7.12 Speedups when freeing registers in idle contexts for the larger and smaller memory hierarchies, at different register file sizes 136

List of Tables

2.1 Processor instruction latencies....10

4.1 Per-processor configurations....28

4.2 Benchmark suite....32

4.3 Memory hierarchy details....33

4.4 Throughput comparison of MP2, MP4, and SMT....35

4.5 Average frequencies for inefficiency metrics (MP2 configurations)....39

4.6 Average frequencies for inefficiency metrics (MP4 configurations)....39

4.7 Speedups for MP2 and MP4 configurations relative to 1 thread MP2....40

4.8 Benchmark throughput for SMT.T1....43

4.9 SMT speedups relative to 1 thread MP2....43

4.10 SMT throughput in instructions per cycle....44

4.11 SMT throughput in the computation phase of SPLASH-2 benchmarks....44

4.12 Throughput for parallel and multiprogrammed workloads....48

4.13 Branch and jump misprediction rates....48

4.14 Comparison of memory system interference for parallel and multiprogrammed workloads....49

5.1 Benchmarks....57

5.2 CPU parameters used in our simulator....59

5.3 Memory hierarchy parameters....59

5.4 TLB miss rates for a blocked distribution and a 48-entry data TLB....60

5.5 Improvement in TLB miss rates of cyclic distribution over blocked....64

5.6 Percentage increase in dynamic instruction count due to profile-driven software speculative execution 67

5.7 Throughput with and without profile-driven software speculation for 8 threads 68

6.1 Memory behavior characterization for OLTP on a single-threaded uniprocessor 87

6.2 Memory behavior characterization for DSS on a single-threaded uniprocessor 92

6.3 Proportion of total misses due to each segment on an 8-context SMT....96

6.4 Global and local L2 cache miss rates for 16 threads running on an SMT with 1-8 contexts....98

6.5 Local L1 cache miss rates for 16 threads running on an SMT, with and without offsetting of per-process PGA data....100

6.6 OLTP throughput with 16 and 32 server processes, bin-hopping, and application-level offsetting....104

6.7 Architectural metrics for superscalar and 8-context SMT on OLTP and DSS workloads....105

7.1 Configuration and latency parameters of the SMT memory system used in this study 113

7.2 Register unavailability for the large cache hierarchy 117

7.3 Register unavailability for the small cache hierarchy. 118

7.4 Average number of dead registers per cycle when executing 8 threads for the large cache hierarchy....118

7.5 Average number of dead registers per cycle when executing 8 threads for the small cache hierarchy 119

7.6 Dead register distance for 264 registers and the smaller cache hierarchy 119

7.7 The opcodes used in Free Opcode 125

Acknowledgments

I owe many thanks to a number of people who have assisted me throughout my graduate studies. I'd like to thank my advisors Susan Eggers and Hank Levy, who have taught me a tremendous amount about conducting research and have guided my growth as a graduate student. I would also like to thank Jean-Loup Baer for serving on my reading committee.

I have been fortunate to have had the opportunity to work with many excellent people during graduate school. I would like to thank my colleagues on the simultaneous multithreading project: Dean Tullsen, Sujay Parekh, and Brian Dewey. I would also like to thank Joel Emer and Rebecca Stamm from Digital Semiconductor, and Luiz Barroso and Kourosh Gharachorloo, from Digital's Western Research Laboratory. Special thanks to Luiz for hosting me during my summer internship at WRL. I would also like to thank Marc Fiuczynski, Dennis Lee, Matthai Phillipose, and Xiaohan Qin for the many fun and stimulating discussions we had.

I'd also like to thank Frankye Jones for answering all of my questions about graduate student issues and helping guide me through all the graduation requirements.

I owe many thanks to Nancy Burr for her help in maintaining our workstations, which I used and abused while running my endless processor simulations. Thanks also to Melody Kadenko-Ludwa for assisting me with graduate school paperwork and travel arrangements.

To my roommates Colin Bryar, John Tippett, and David Rubenstein, thank you for making my stay in Seattle so enjoyable.

I owe thanks to Intel Corporation for sponsoring my research with a graduate fellowship.

My family has played an important role throughout my life. To my late father, Chienming Lo, thank you enough for the guidance you gave me and for showing me the virtues of dedication, hard work, and sacrifice for others. To my mom, Billie, thanks for the lifelong support and encouragement you have given. To Wayne and Elaine, I couldn't have asked for a better pair of siblings.

Last, but certainly not least, I'd like to thank Miyako Takahashi, whose love and companionship I cherish dearly.

Chapter 1

Introduction

As on-chip transistor density continues to increase, processor designers face the challenge of identifying new architectures for higher-performance next-generation chips. In recent years, high-end processors have relied on multiple-issue superscalar technology to boost performance. These superscalar processors exploit instruction-level parallelism (ILP) by providing multiple functional units and the ability to issue multiple instructions each cycle. The increase in processor issue width, however, has not seen a commensurate increase in processor throughput, because single applications often lack sufficient ILP to exploit larger amounts of parallel, on-chip hardware. This lack of instruction-level parallelism has led to a vast under-utilization of processor resources; further attempts to increase the issue width of traditional superscalar processors will only exacerbate this trend.

For next-generation architectures, though, parallelism need not be limited to fine-grain ILP within a single application. In many workloads, additional coarser-granularity parallelism may exist in several forms. A multiprogrammed workload may consist of several independent applications, while a parallel program may contain several parallel processes or threads. Parallel applications not only include traditional scientific programs, but also include database and web servers. Even desktop applications have shown an increasing reliance on coarse-granularity multithreading. This trend should continue as programmers take advantage of the multithreaded language constructs and interfaces provided by mainstream programming languages (Java [44]) and operating systems (Windows NT [23]). We can broadly classify all of these forms of coarse-granularity parallelism as *thread-level parallelism*.

Many researchers have proposed architectures that use this thread-level parallelism

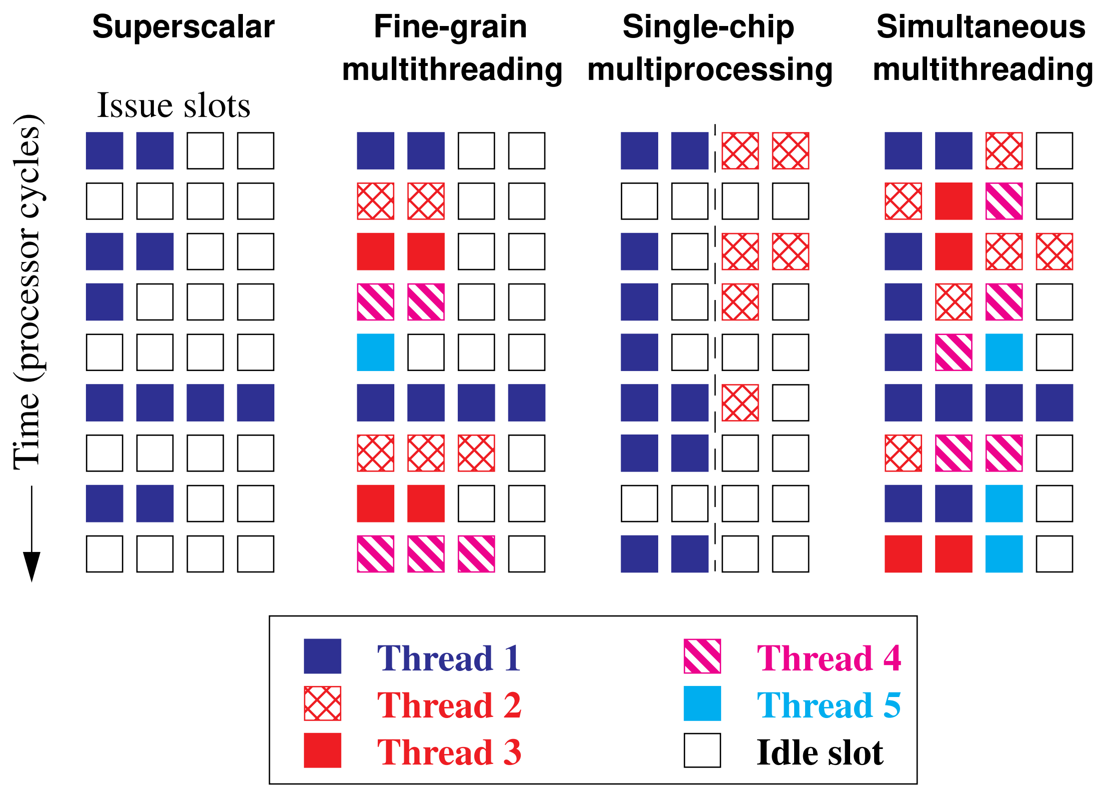


Figure 1.1: A comparison of issue slot (functional unit) utilization in various architectures, assuming each has four functional units.

Each square corresponds to an issue slot, with white squares signifying unutilized slots. Hardware utilization suffers when a program exhibits insufficient parallelism or when available parallelism is not used effectively. A superscalar processor achieves low utilization because of low ILP in its single thread. Multiprocessors physically partition hardware to exploit TLP, and therefore performance suffers when TLP is low (for example, in sequential portions of parallel programs). In contrast, simultaneous multithreading avoids resource partitioning. Because it allows multiple threads to compete for all resources in the same cycle, SMT can cope with varying levels of ILP and TLP; consequently, utilization is higher and performance is better.

(TLP) to exploit the greater amount of on-chip processing hardware. Fine-grained multithreading [5, 52, 68, 72, 112, 124], single-chip multiprocessing [53, 93], and simultaneous multithreading [31, 133, 134] are three such architectural techniques. Each of these architectures provide multiple program counters and register sets, enabling them to execute along several threads of control. In Figure 1.1, we compare these architectural paradigms with superscalar processing.

In a fine-grain multithreaded processor (Figure 1.1b), such as the Tera MTA [5], the CPU can context switch between threads in a single cycle. This hardware support enables multiple threads to execute on a single processor, thereby exploiting thread-level parallelism to hide long-latency operations. A single-chip multiprocessor (Figure 1.1c) also exploits thread-level parallelism, but executes a separate thread on each of its multiple CPUs.

In both architectures, hardware unnecessarily differentiates thread-level parallelism from instruction-level parallelism. In a fine-grain multithreaded processor, functional unit resources are partitioned *temporally*. During each cycle, functional units can only be used by a single thread. If insufficient ILP exists, multiple issue hardware will be wasted. In a single-chip multiprocessor, on-chip resources are partitioned *physically*; the processor resources are divided among each of the multiple CPUs on the chip. Consequently, a single thread executing on a particular CPU only has access to the resources on that CPU, even if the resources on another CPU are idle.

This inflexibility is disconcerting for chip designers, because hardware resources must be statically allocated at design-time. Given a fixed amount of hardware, architects must determine how much should be used to exploit ILP and how much should be dedicated for TLP. If the design does not match the ILP and TLP characteristics of the application, then performance suffers. Unfortunately, the levels of instruction-level and thread-level parallelism are highly variable both within an application and across a workload. It therefore becomes very difficult to hit this performance “sweet spot.”

Simultaneous multithreading (Figure 1.1d) addresses these problems with an architecture that allows instruction-level and thread-level parallelism to be used interchangeably. Simultaneous multithreading achieves this flexibility by allowing multiple threads to issue multiple instructions to the functional units each cycle. The multiple executing threads *dynamically share* processor resources both temporally and physically; each cycle, ready instructions can be issued from any thread (temporal sharing) and all resources are potentially available to a single thread (physical sharing).

This fine-granularity sharing enables simultaneous multithreading to address the inflexibility and resource partitioning that plague fine-grain multithreading and single-chip multiprocessing. An SMT processor accommodates variations in parallelism because it essentially converts thread-level parallelism into instruction-level parallelism. When a program has only a single thread, i.e., it lacks TLP, all of the SMT processor's resources can be dedicated to that thread; when more TLP exists, this parallelism can compensate for a lack of per-thread ILP. An SMT processor can uniquely exploit whichever type of parallelism is available, thereby utilizing the functional units more effectively to achieve the goals of greater throughput and performance.

1.1 Potential costs of shared resources

While dynamic resource sharing enables SMT to achieve performance gains, it also has some potential drawbacks. With its simultaneous multi-thread instruction issue, simultaneous multithreading can place much greater stress on the shared on-chip hardware. This stress can be manifested in two primary ways: (1) increased interference and (2) contention.

The first form of stress occurs when the working sets of multiple threads or processes interfere in shared structures, such as the caches, TLB, and branch prediction tables. The primary task of these shared structures is to "cache" some portion of a thread's execution state, including memory state, address translations, or branch history. With multiple executing threads, however, conflicts or *interference* can degrade the effectiveness of these hardware resources. Most modern systems already see some of these effects, because they also execute several different threads at once, e.g., the operating system context switches between several applications in a multiprogrammed workload. But with its finer-granularity multithreading, an SMT processor can exacerbate interference in shared resources, thereby diminishing the degree of locality. The most prominent example of interference is an increased number of conflict misses, which may occur when the working sets of multiple threads compete for a shared cache. Additional cache misses can also occur when the working sets exceed the size of the cache, resulting in an increased number of capacity misses.

The second form of stress occurs when multiple threads cause excessive resource *contention*, rather than interference. An SMT processor may experience greater contention than a traditional processor, as threads compete for shared resources such as functional units and cache ports. Modern software may exacerbate this problem because applications have not been designed to run on an SMT processor. As a result, software may not exploit shared resources efficiently, or even worse, may cause excessive contention on an SMT. For example, several compiler optimizations may increase the number of dynamically executed instructions as a side effect of providing speculative execution or better cache reuse. On an SMT processor, this additional instruction overhead may degrade performance by increasing competition for shared functional units.

Given these two potential costs, multi-thread resource sharing on an SMT processor raises several questions.

- What shared structures suffer from interference?
- What are some of the causes of interference?
- What is the performance impact of contention?
- Does interference or contention ever overwhelm the benefits of dynamic resource sharing?
- How do software optimizations and policies affect interference?
- How can software manage threads to minimize or avoid interference and contention in shared resources?
- How can software encourage inter-thread sharing to take advantage of shared structures such as the cache and TLB?
- What techniques can be applied to ensure that shared resources are used efficiently?

This dissertation addresses these and many other related questions.

1.2 Scope and contributions of this dissertation

Prior research in simultaneous multithreading provides the architectural framework for this dissertation. We build on this prior work by exploring the implications of exploiting thread-level parallelism on a processor that provides fine-granularity resource sharing. The goal of this dissertation is to investigate software and hardware techniques for promoting constructive sharing and efficient utilization of processor resources on simultaneous multithreaded processors. These techniques enable an SMT system to take full advantage of the dynamic resource sharing provided by the hardware. An SMT processor can better exploit thread-level parallelism by adopting optimizations that encourage effective resource sharing, while avoiding destructive behavior that causes interference and contention.

This dissertation makes the following contributions:

First, we evaluate the performance benefits of dynamic resource sharing by comparing an SMT processor to both wide-issue superscalars and single-chip multiprocessors. Using several metrics to identify resource waste, we provide a detailed analysis that quantifies the costs of static partitioning on an MP, as well as the performance gain due to dynamic resource sharing on an SMT. Our experiments demonstrate that an SMT achieves roughly a 50% performance edge over a single-chip multiprocessor with comparable execution resources, because of its ability to adapt to variable levels of ILP and TLP. These experimental results attest to the importance of efficient and flexible resource sharing.

Second, we identify several avoidable causes of interference in shared structures on an SMT processor. For example, we show that inappropriate software policies can introduce excessive interference or even thrashing in an SMT processor. This typically occurs when the compiler, operating system, or application do not recognize the differences between an SMT processor and traditional architectures, such as superscalars and shared-memory multiprocessors.

Third, we present software techniques for minimizing interference in shared resources. By taking into consideration the shared memory subsystem and shared functional units on an SMT, software policies can minimize or avoid interference introduced by multithreading. In particular, we show that (1) cyclic loop iteration scheduling (as opposed to blocked scheduling) is more appropriate for an SMT parallelizing compiler, because of its ability to reduce the TLB footprint; (2) an SMT compiler should limit the use of software speculative execution, because the additional overhead of mis-speculated instructions can decrease useful instruction throughput; (3) loop tiling should promote inter-thread tile sharing to maximize cache reuse; (4) the operating system should use a bin hopping policy to minimize cache interference in database workloads, and (5) a database application can also reduce inter-thread cache conflicts by using application-level offsetting.

Fourth, we propose software-directed register deallocation, a set of techniques that improves the utilization of an SMT processor's shared register file. With these techniques, software communicates register last-use information to register renaming hardware so that physical registers can be deallocated more aggressively. We propose several architectural mechanisms and evaluate their use by the compiler and operating system. By improving physical register utilization and managing the shared physical register file more efficiently, software-directed register deallocation can achieve a 25% reduction in register file size without sacrificing performance.

1.3 Organization of the dissertation

This dissertation is organized as follows. Chapter 2 describes simultaneous multithreading. Chapter 3 discusses related work at an architectural level, focusing on processor designs that motivated and/or influenced simultaneous multithreading. Subsequent chapters will include their own discussions of related work, concentrating on research relevant to the particular chapter. Chapter 4 compares SMT with superscalars and on-chip multiprocessors to illustrate the benefits of dynamic resource sharing. Chapters 5 and 6 examine software techniques to minimize contention and interference in an SMT processor's shared resources: Chapter 5 investigates techniques for tuning compiler optimizations on SMT processors running scientific workloads comprised of both compiler- and programmer-parallelized applications. Chapter 6 describes operating system and application-level techniques for avoiding inter-thread cache conflicts in memory-intensive database workloads. It also analyzes the nature of cache interference in these applications. Chapter 7 discusses software-directed register deallocation, a technique that allows software to improve the utilization of the shared register file in an SMT processor. Chapter 8 concludes.

Chapter 2

Simultaneous multithreading

Before exploring the implications of resource sharing for simultaneous multithreaded processors, we briefly describe the SMT microarchitecture (first presented in [133]).

A simultaneous multithreaded processor provides several hardware contexts that enable the processor to execute along several “threads of control.” Each context maintains the state, including registers and a program counter, for an executing thread. These hardware contexts, however, are not limited to executing lightweight threads; the contexts can contain different processes as well as threads within the same address space. For the purposes of this dissertation, we collectively refer to these independent “threads of control” simply as threads.

Our SMT architecture has two important characteristics. First, simultaneous multithreading enables multiple threads to share processor resources dynamically. This is the focus of this dissertation. An equally important characteristic is the simplicity of our SMT microarchitecture. Simultaneous multithreading adds minimal hardware complexity to, and is a minimal extension of, modern out-of-order superscalar processors. By leveraging existing superscalar technology, we can minimize the impact of designing an SMT processor, and help facilitate the migration of this technology into mainstream, commodity processors. This chapter provides more background for the SMT architecture studied in this dissertation. Section 2.1 describes the out-of-order superscalar technology that forms the basis for SMT. In Section 2.2, we discuss the hardware extensions that enable simultaneous multithreading. Section 2.3 describes synchronization support in SMT processors. We summarize in Section 2.4.

2.1 Base processor architecture

In this dissertation, we investigate several architectures, including SMT and single-chip multiprocessors (Chapter 4). Each of these architectures is built upon a common processor core, which is a sophisticated, out-of-order superscalar processor with a

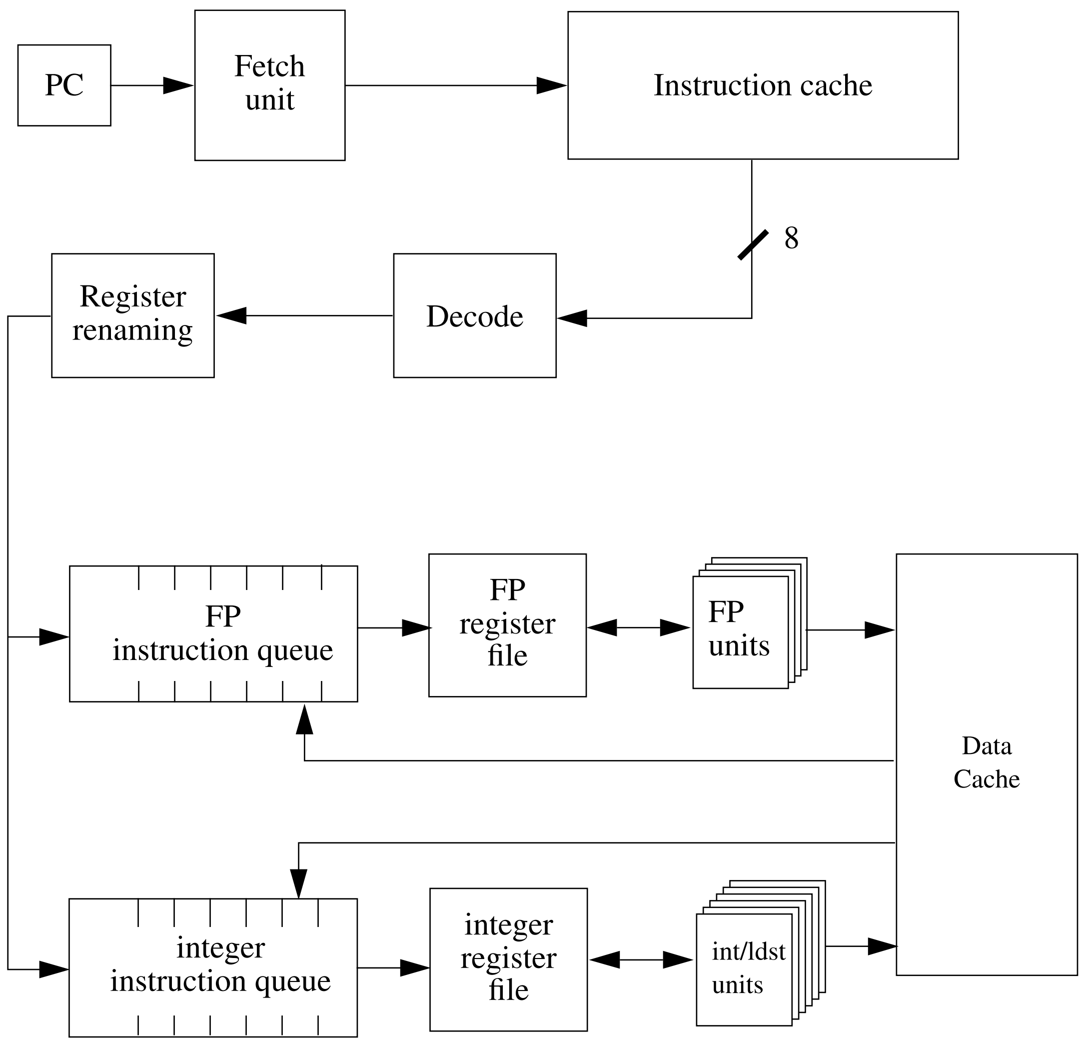


Figure 2.1: Organization of the dynamically-scheduled superscalar base processor used in this dissertation.

dynamic scheduling core similar to the MIPS R10000 [149].

The organization of this base processor is shown in Figure 2.1, while Figure 2.2 (a) illustrates its processor pipeline. On each cycle, the processor fetches a block of instructions from the instruction cache. After decoding these instructions, the register renaming logic removes false register dependences by mapping the architectural registers to a pool of physical renaming registers. Instructions are then fed to either the integer or floating point instruction queues. When their operands become available, instructions are issued from these queues to the corresponding functional units.

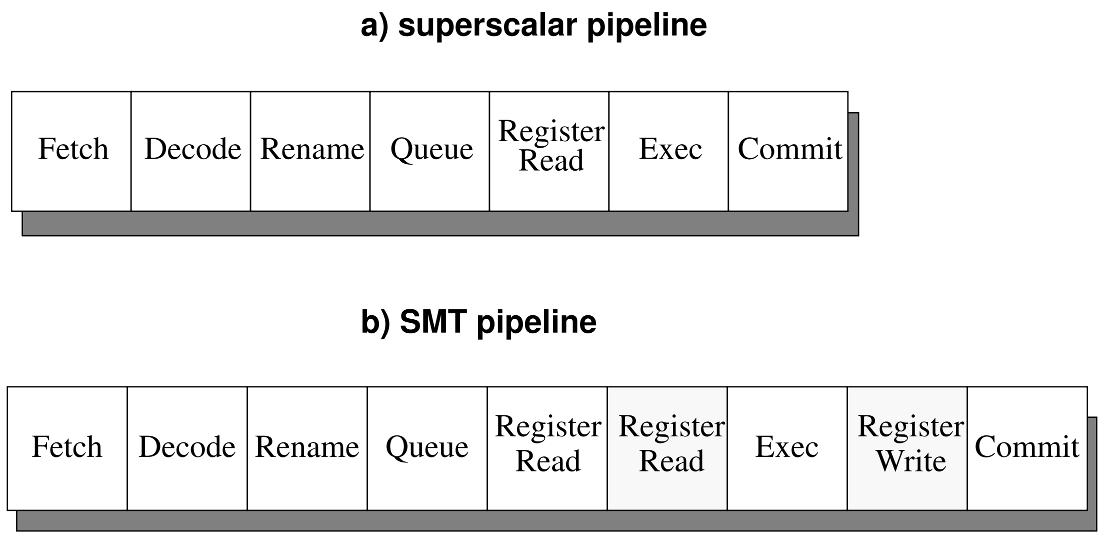


Figure 2.2: Comparison of the pipelines for (a) a conventional superscalar processor and (b) our SMT processor.

Table 2.1: Processor instruction latencies. These values are the minimum latencies from when the source operands are ready to when the result becomes ready for a dependent instruction.

Instruction class	Latency
integer multiply	8, 16
conditional move	2
compare	0
all other integer	1
FP divide	17, 30
all other FP	4
load (cache hit)	1

Instructions are retired in order.

The base processor's instruction set is based on the DEC Alpha instruction set architecture. The instruction latencies for the functional units are similar to those of the DEC Alpha 21164 [30] and are listed in Table 2.1.

2.2 Extensions to support SMT

Our SMT architecture is a straightforward extension of the base processor. We model an 8-wide SMT processor with eight hardware contexts, each of which consists of a program counter and register state.

To enable simultaneous multithreading on the eight hardware contexts, little of the microarchitecture need to be redesigned -- most components are an integral part of any conventional, dynamically-scheduled superscalar. The two major exceptions are the instruction fetch mechanism and a larger register file, highlighted in Figure 2.3.

2.2.1 Instruction fetch unit

Instruction fetching is still driven by a conventional system of branch prediction hardware (branch target buffer and branch prediction logic), but the fetch unit must manage eight program counters in our SMT architecture. Because of the complexities associated with an eight-ported instruction cache, the fetch mechanism only selects up to two threads (among threads not already incurring I-cache misses) in each cycle. Up to eight instructions are then fetched from the two threads (this is the 2.8 scheme from [133]).

To improve fetch efficiency, we augment this fetch mechanism with an intelligent fetch policy for prioritizing thread selection (determine which two threads get to fetch each cycle). Thread priorities are assigned using the ICOUNT feedback technique [133], which favors threads that are using processor resources most effectively. Under ICOUNT, highest priority is given to the threads that have the least number of instructions in the decode, renaming, and queue pipeline stages. This approach prevents a single thread from clogging the instruction queue, avoids thread starvation, and provides a more even distribution of instructions from all threads, thereby heightening inter-thread parallelism.

2.2.2 Register file and register renaming

Following instruction fetch and decode, register renaming is performed, as in the base processor. The instruction set allows each thread to address 32 architectural integer

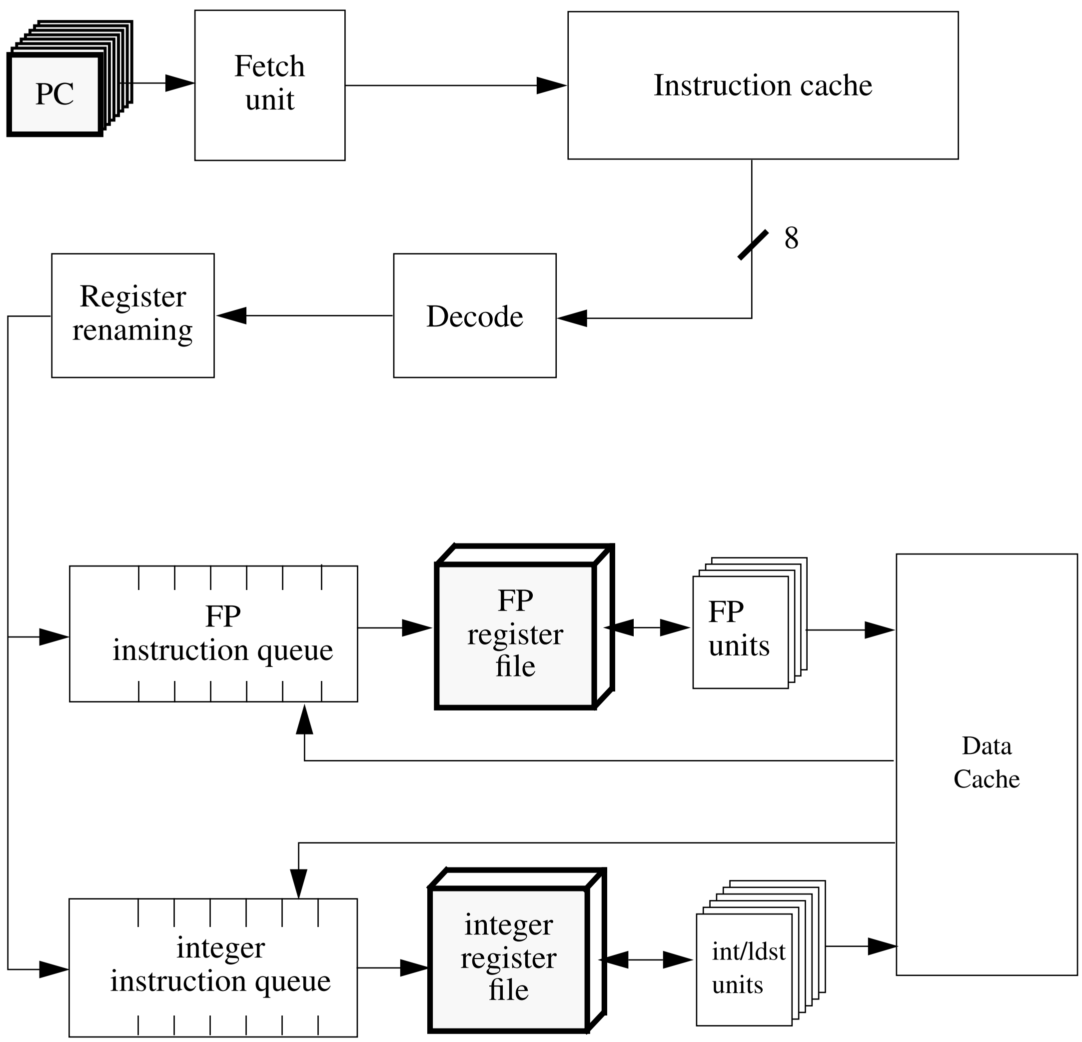


Figure 2.3: Organization of the SMT processor used in this dissertation.

(and FP) registers. The register renaming mechanism maps these architectural registers (one set per thread) onto the machine's physical registers. An eight-context SMT will require at least 8*32 = 256 physical registers, plus additional physical registers for register renaming. With a larger register file, longer access times will be required, so the SMT processor pipeline is extended by two cycles to avoid an increase in the cycle time. Figure 2.2 compares the pipeline for the SMT versus that of the base superscalar. On the SMT, register reads take two pipe stages and are pipelined. Writes to the register file behave in a similar manner, also using an extra pipeline stage. In practice, we found that the lengthened pipeline degraded performance by less than 2% when running a single thread [31, 133]. The additional pipe stage requires an extra level of bypass logic, but the number of stages has a smaller impact on the complexity and delay of this logic ($O(n)$) than the issue width ($O(n^2)$) [95]. In a previous study [133], we provide more details regarding the effects of the two-stage register read/write pipelines on the architecture and performance.

2.2.3 Other required modifications

In addition to the fetch unit and register file, a few other mechanisms must also be modified to support SMT. Per-context mechanisms are required for branch prediction recovery, interrupt/trap/exception handling, subroutine return stack prediction, and in-order per-thread instruction retirement. Thread identifiers are also required in the TLB and branch target buffer. Notably missing from this list is special per-thread hardware for scheduling instructions onto the functional units. Instruction scheduling is done as in a conventional, out-of-order superscalar: instructions are issued after their operands have been calculated or loaded from memory, without regard to thread; the renaming hardware eliminates inter-thread register name conflicts by mapping thread-specific architectural registers onto the processor's physical registers. The conventional instruction queues can then be used to dynamically schedule instructions from multiple threads. Instructions from all threads are dumped into the instruction queues, and an instruction from any thread can be issued from the queues once its operands become available.

In this design, few resources are statically partitioned among contexts; consequently, almost all hardware resources are available even when only one thread is executing. The architecture allows us to achieve the performance advantages of simultaneous multithreading, while keeping intact the design and single-thread peak performance of the dynamically-scheduled CPU core present in modern superscalar architectures.

2.3 SMT synchronization

In this dissertation, some of our experiments include the execution of parallel applications and therefore require synchronization mechanisms. On traditional single-threaded processors, synchronization is typically implemented using spinning locks. While spinning on a lock, a thread consumes resources on the local processor. For example, the instructions in a tight spin-wait loop must still be fetched and executed. On an SMT processor, this resource consumption can be costly, as it consumes resources that could be used by other executing threads. Consequently, blocking synchronization may be more appropriate [135].

We have therefore chosen to modify the processor in order to support hardware blocking synchronization primitives for SMT and enable cheap inter-thread synchronization. A synchronization functional unit, also called a lock-box, handles lock acquire and release instructions for all threads running on the CPU [135]. If a thread tries to acquire a lock that is already being held, then (1) the lock address is placed in an entry of the lock box and (2) the thread blocks, i.e., instruction fetch for this thread is stalled. When a thread releases a lock, the processor does a lookup of the lock address in the lock box. If there is a match, then the blocked thread is allowed to continue execution, by re-enabling instruction fetch. By using an in-processor synchronization unit, these lock acquire and release operations are much faster than traditional memory-based synchronization mechanisms. For more details about the lock-box, refer to the study by Tullsen et al. [135].

2.4 Chapter summary

In this chapter, we discussed an SMT microarchitecture that achieves substantial increases in processor throughput with requiring radical redesign of a modern processor’s microarchitecture. By leveraging modern out-of-order superscalar technology, the design complexity of an SMT processor can be greatly minimized. A few extensions are required in the microarchitecture primarily to maintain the state of the multiple threads (e.g., multiple program counters, register state, and thread identifiers in various locations). The dynamic scheduling core, however, remains largely intact.

Chapter 3

Related work

The previous chapter described an SMT architecture that builds on a large body of research in parallel architectures. This chapter reviews the prior work to provide context for the SMT studies in this dissertation. We concentrate on parallel architectures that exploit thread-level parallelism, focusing on various multithreaded architectures. We divide the prior work into three primary categories, based on the granularity of resource sharing: (1) coarse-grain multithreaded architectures, (2) fine-grain multithreaded architectures, (3) architectures that allow simultaneous, multi-thread instruction issue. We also discuss a fourth category of parallel architectures that advocate various alternative paradigms for exploiting thread-level parallelism. By comparing these architectures, we motivate the techniques proposed in this dissertation, which allow threads to exploit an SMT processor's uniquely flexible resource sharing.

3.1 Coarse-grain multithreading

In a coarse-grain multithreaded architecture, the processor only executes a single thread at a time, but has on-chip register state for multiple threads. The multiple register sets enable the processor to hide long-latency operations by quickly context switching between threads. The context switch time is typically on the order of 4-6 cycles, which is too large to hide shorter latencies, such as branch mispredictions or on-chip cache misses.

Two examples of coarse-grain multithreading are the APRIL processor [2] and a recently proposed PowerPC multithreaded CPU [118].

Several researchers have studied the performance of coarse-grain multithreaded architectures. Using a memory-intensive commercial database workload, Eickemeyer et al. [32] found that coarse-grain multithreading could increase throughput by 70% when using two or three threads. Weber and Gupta [142] evaluated the latency-hiding abilities of coarse-grain multithreading. They found that substantial processor utilization could be achieved, especially when context switch overhead is low, relative to remote access latencies. Thekkath and Eggers [122] examined the effectiveness of multiple contexts on conventional, coarse-grained multithreaded architectures for a set of parallel benchmarks. Although multithreading could be effective, the benefits were highly dependent on the data locality in the applications. In another study, Thekkath and Eggers [123] investigated the effects of sharing-based thread placement algorithms in a multiprocessor of coarse-grain multithreaded CPUs. Surprisingly, they found that performance was insensitive to sharing-based placement, because all threads had uniform sharing patterns and threads share data, but not at a fine granularity. As a result, load balancing was the most critical factor for thread placement.

Several researchers have proposed analytic models for evaluating multithreaded processors. Agarwal [1] measured the degradation in processor utilization that resulted from increases in cache interference and network contention. Saavedra-Barrera et al. [107] presented a model for multithreaded processor efficiency and explored the architectural design space for coarse-grain multithreaded processors. Lim and Bianchini [74] presented an analytic model of the performance benefits of multithreading on the Alewife multiprocessor. Nemawarkar et al. [88] present a single analytic model that incorporates the processor, memory, and network. They examine the interactions between thread run-lengths (the time between context switches) and memory and network latencies. Dubey, Krishna, and Flynn [28] present an analytic model for determining pipeline utilization as a function of the number of threads. Lastly, Dubey, Krishna, and Squillante [29] describe a comprehensive analytic model for evaluating a range of multithreaded architectures, including both coarse-grain and fine-grain multithreading.

3.2 Fine-grain multithreading

Fine-grain multithreaded architectures provide slightly more hardware than coarse-grain multithreaded CPUs, enabling the processor to switch between contexts on every cycle. Processor resources are thus shared at a finer-granularity. In any given cycle, though, only one thread can still issue instructions. This form of multithreading first appeared in the I/O unit of the CDC 6600 [125], where several peripheral processors timeshared a pipeline. Flynn and Podvin [37] and Kaminsky and Davidson [61] later proposed architectures that allowed multiple instruction streams to share a single processor pipeline in a time-multiplexed fashion.

The Tera MTA [5], the Denelcor HEP [112], and Horizon [68, 124] are other examples of fine-grain multithreading. These architectures are focused on achieving high throughput and rely on a large number of threads to hide long memory and network latencies. These machines do not provide data caches or bypass logic, thus sacrificing single-thread performance.

Laudon et al. [72] propose a fine-grain multithreaded architecture designed to address the shorter latencies and more limited parallelism available on workstations. Their interleaved architecture provides data caches and pipeline interlocks to improve single-thread performance. They demonstrate that this design can effectively tolerate short latencies using a small number of hardware contexts.

MASA [52] employs fine-grain multithreading to hide operation latencies and efficiently execute independent operations in parallel Lisp programs.

Farrens and Pleszkun [35] evaluate the performance of three types of multithreading for a deeply-pipelined processor, roughly based on the Cray-1S. They compare every-cycle (fine-grain), blocked (coarse-grain), and prioritized (a variant of coarse-grain) multithreading. All schemes had zero-overhead context switching. They found that the blocked scheme had the best performance on the Livermore Loops.

3.3 Simultaneous multithreading

Prior work in simultaneous multithreading has concentrated on the architectural design considerations and performance evaluation of SMT processors. Although the term "simultaneous multithreading" was coined by Tullsen et al. [133, 134], several other studies have proposed similar architectures that also allow multiple threads to issue instructions in the same cycle. While these architectures are conceptually similar at a high-level, they differ widely in their implementation and flexibility. We subdivide these architectures into two categories based on whether they allow dynamic partitioning of functional units and issue slots.

3.3.1 Dynamic partitioning

Tullsen et al. [134] presented an idealized SMT architectural model to evaluate the performance potential of simultaneous multi-thread instruction issue. Using a multiprogrammed workload of scientific applications, the study demonstrated that superscalar and fine-grain multithreaded processors lack sufficient instruction-level parallelism to fully utilize the functional units in a wide-issue processor. By allowing multiple threads to issue each cycle, instruction throughput increased greatly. The study also evaluated various configurations of SMT processors, including various cache designs and issue restrictions. Finally, the study favorably compared SMT to single-chip multiprocessors.

A later study [133] presented an implementable SMT microarchitecture, summarized in Chapter 2. This study evaluated several fetch and issue policies for improving the performance of an SMT processor.

These two studies also measured the degradations in cache hit rates and branch prediction accuracy that result of simultaneous multi-thread instruction issue. This dissertation builds upon the prior work by investigating the causes of inter-thread interference and by proposing several techniques for minimizing conflicts in shared structures on simultaneous multithreaded processors. These techniques can also be applied to the following architectures, which also permit threads to share resources at a fine granularity.

Hirata et al. [55] propose an architecture that dynamically packs instructions from different streams (threads). They achieve a 3.72 speedup over a single-threaded CPU when using four parallel threads for a ray-tracing application. Their simulations do not include the effects of caches or TLBs. The architecture includes multiple instruction queue and decode units. Each decode unit provides scoreboarding to check register dependences. When an instruction is ready, the decode unit arbitrates for an available functional unit. If there are no available functional units, then the ready instruction is placed in a standby station.

Yamamoto et al. [148] present an analytic model, which was also validated by a simulator, to estimate the performance of a multithreaded superscalar processor. Their model assumes a perfect memory system, perfect branch prediction, and no branch delay slots. They model the performance of several homogeneous workloads, in which all threads executed the same program. When executing four threads, they observe a 3-fold speedup over a single-threaded processor.

Yamamoto and Nemirovsky [147] extend this work with an architectural study of an out-of-order superscalar with multithreading. Each thread has its own 8-entry dispatch window and fetch unit. Ready instructions from these dispatch windows compete for available functional units to achieve simultaneous multi-thread instruction issue. Their simulator models an architecture with nine functional units and up to four threads. They model shared on-chip caches with a 5-cycle miss penalty, but do not model the next level of the memory hierarchy. In their study, they measure the execution of a multiprogrammed workload consisting of several SPEC and other UNIX benchmarks. They observe a 77% speedup when running two threads and a 107% speedup for four threads. They also examine the effect that multithreading has on the performance of an individual thread. On average, an individual thread's performance degraded by 13% and

47% when running with one and three other threads, respectively, because processor resources had to be shared between multiple threads.

Daddis and Torng [25] propose an architecture that combines out-of-order execution and multithreading. Instructions are fetched and then placed in an instruction stack (instruction window). They model an infinite number of functional units and assumed no data cache misses. Although their architecture could be extended to support more than two threads, their experiments are limited to a two-threaded processor. Using the Dhrystone and the Livermore Loops, they measure instruction throughput when varying both the fetch bandwidth and instruction stack size. With an 8-instruction fetch width and a 32-entry instruction stack, they observe speedups of up to 100% over a single-threaded processor.

Prasadh and Wu [101] and Keckler and Dally [63] propose architectures in which VLIW instructions from multiple threads are dynamically interleaved onto a processor. In these architectures, a single VLIW instruction consists of multiple operations that the compiler has determined to be data independent. These VLIW instructions may include no-ops, when the compiler cannot extract sufficient instruction-level parallelism. Prasadh and Wu describe an architecture that replaces these no-ops with instructions from other threads to improve processor utilization. Each thread has its own register set and instruction fetch unit. A dynamic interleaving unit uses a priority scheme to select operations from the multiple threads. Their simulations assume infinite instruction and data caches. With four threads, they achieve nearly a 2.5 speedup over a single-threaded processor, when executing a workload of 14 Livermore Loops and 4 other non-numeric benchmarks.

Keckler and Dally [63] propose processor coupling, which provides a hardware scheduling mechanism to interleave VLIW operations from multiple threads. A single processor-coupled node consists of several clusters of functional units and a shared register file. They use a functional simulator to evaluate this architecture and a statistical model to simulate the memory system, but ignore the instruction cache. On four numeric benchmarks, processor coupling achieves 60-75% speedups. They also examine the effects of varying the number and mix of functional units. Their study also discusses various mechanisms for inter-cluster communication and synchronization, as well as some chip area estimates.

Gulati and Bagherzadeh [47] use multithreading to improve the performance of a superscalar processor. As with our design, an out-of-order processor core provides the hardware to enable simultaneous, multi-thread instruction issue. However, the machine has 128 registers that the compiler statically partitions among the executing threads.

They achieve a 20%-55% speedup over a single-threaded superscalar for a set of scientific benchmarks. Because their processor is much narrower than the 8-wide processor used in our studies, they observe much lower speedups when compared to our studies. A narrower machine has fewer wasted or unutilized issue slots and therefore, there is less room for improvement.

Li and Chu [73] use an analytic model to study an SMT architecture in which each hardware context has its own instruction-cache. They use the model to evaluate the effects of varying several architectural parameters, including the numbers of contexts and functional units.

The XIMD [145] architecture extends a VLIW architecture to execute multiple threads simultaneously. The compiler controls the distribution of execution resources to threads and can partition the functional units between threads on a cycle-by-cycle basis. To handle inter-thread dependencies, the compiler can use explicit synchronization mechanisms. In order for the compiler to control the machine's execution resources, all threads must belong to the same application.

Espasa and Valero [33] extend a simultaneous multithreaded architecture to support vector instructions. The SMV architecture exploits data-level parallelism in vectorizable applications. They compare combinations of vector processing, out-of-order execution, and multithreading on several architectural configurations. Their simulations show that by combining all three techniques, SMV reduces the performance sensitivity to longer memory latencies.

3.3.2 Partitioned multiple-thread issue architectures

Several other architectures allow multiple threads to issue each cycle, but lack the flexible resource sharing provided by SMT. The architectures described by Govindarajan et al. [45], Gunther [48], and Beckmann and Polychronopoulos [9] partition issue bandwidth among threads in a fixed fashion. These architectures lack flexible resource sharing, which contributes to resource waste when only a single thread is running. Govindarajan et al. [45] allow multiple threads to be executed in a single processing element by using multiple execution pipes. On each cycle, a single instruction is selected from each pipe. These processing elements are joined with an interconnection network to access a shared memory system. Within a single processing element, resources are partitioned between execution pipes. Gunther [48] proposed the Concurro processor, in which processor are partitioned among context groups. Threads within a context group share resources, and only one instruction can be issued from each group on every cycle.

M-Machine Multicomputer

The M-Machine [36, 64] Multicomputer consists of four multi-ALU processing (MAP) nodes that enable the architecture to exploit both instruction-level and thread-level parallelism. To exploit instruction-level parallelism, each MAP node is an LIW processor that executes several operations each cycle. The compiler extracts ILP to generate a sequence of LIW instructions called an H-thread, which each run on a single MAP node. If the level of ILP is very high or if thread-level parallelism exists, the compiler can create additional H-threads to run on the other MAP nodes. Each H-thread runs on a separate MAP node, and this set of four H-threads is called a V-thread. Parallelism can also be exploited at an even coarser granularity, by taking advantage of the hardware support for fine-grain multithreading in each MAP node. A MAP node can switch between threads to hide operation latencies. But like other fine-grain multithreaded CPUs, a MAP node also suffers from horizontal waste when the compiler cannot extract sufficient ILP.

Despite their apparent architectural differences, an SMT processor and the M-Machine Multicomputer are rather similar. The M-Machine can be viewed as a coarse-grained SMT, in which an LIW instruction corresponds to a single instruction in an SMT processor, and a single MAP node could be viewed as a functional unit. At this granularity, multiple executing V-threads can dynamically and flexibly share MAP nodes. Both SMT and the M-Machine recognize the importance of designing an architecture that enables the flexible use of all forms of parallelism. To facilitate the use of fine-grain threading, both architectures also provide fast, blocking hardware synchronization primitives [64, 135].

The primary difference between the two architectures is that while our SMT architecture relies on dynamic scheduling hardware to extract instruction-level parallelism, the M-Machine relies on the compiler for fine-grain thread creation to exploit ILP across multiple MAP nodes. In contrast, the SMT compiler should be more concerned with managing the interactions of threads, as opposed to extracting parallelism (see Chapter 5).

3.4 Other parallel on-chip architectures

In addition multithreaded processors, various other parallel architectures have been proposed to exploit thread-level parallelism.

3.4.1 Single-chip multiprocessors

The Hydra research project advocates the use of single-chip multiprocessors to exploit thread-level parallelism in next-generation CPUs [53, 93]. A primary advantage of this technique is that relies on simpler processor cores that are easier to design and optimize. As a result, they may be able to achieve higher clock rates. The main drawbacks, however, are that these architectures sacrifice single-thread performance, are limited by resource partitioning, and suffer when thread-level parallelism does not exist.

3.4.2 Compiler-dependent thread-level parallel architectures

Several other architectures have been designed to exploit multiple levels of program parallelism with heavy support from the compiler. The MISC [136, 137], Multiscalar [39, 114], and superthreading [132, 131] architectures all require significant compiler support (or hand-coding) to extract maximum performance. In all three designs, execution resources are partitioned in a manner that prevents them from being dynamically shared by all threads.

Tyson and Farrens [136, 137] proposed a MIMD architecture to exploit ILP called MISC. Multiple processing elements are tightly-coupled using FIFO queues, so that fine-grain parallelism (ILP) can be distributed across the PEs.

The Multiscalar architecture [39, 114] exploits both task-level parallelism and instruction-level parallelism. Tasks (which can be as big as a collection of basic blocks) can be speculatively executed by hardware using dynamic branch prediction techniques. Processing units are organized in a circular queue, with a different task assigned to each unit. Hardware support is provided to squash tasks if the branches between tasks are incorrectly predicted.

The superthreading architecture [132, 131] provides thread-level control and data speculation, so that an application can be parallelized using tightly-coupled threads. Despite its name, the architecture is more like a tightly-coupled multiprocessor than a multithreaded or SMT processor. The machine is partitioned into multiple thread processing units, but the units are not completely independent. The processing units are connected by a unidirectional ring and a thread can only fork a thread on the successor processing unit. The architecture relies on a thread pipelining execution model to facilitate overlapping between threads and the enforcement of run-time inter-thread data dependences. A superthreaded processor provides some hardware support for run-time data dependence checking, but heavy compiler support is required both for parallelization and speculation.

The S-TAM architecture [138] exploits both instruction-level and thread-level parallelism by statically allocating threads to processors, and dynamically allocating each thread to a functional unit. A primary goal was to expose scheduling and allocation to the compiler. On their machine intrathread parallelism was avoided; parallelism was instead expressed only between threads.

3.4.3 Trace processors

Smith and Vajapeyam [113] and Rotenberg et al. [105] recently proposed trace processors, which provide hardware to construct and cache dynamic instruction sequences called traces. When the processor executes a series of basic blocks, the hardware can cache the entire sequence, so that future executions need not suffer penalties resulting from taken branches. Traces (instead of instructions) are the basic unit for (1) distributing computation to multiple processing elements and (2) applying control and data speculation. By providing a hierarchy of global and local values, execution resources can be distributed, thus minimizing complex, centralized logic. However, this architecture also introduces its own complexities for handling a trace cache, value prediction, and aggressive speculation.

Trace processors do not exploit thread-level parallelism *per se*, but the hardware essentially extracts “threads” for execution on the processing elements. Each trace is very similar to a thread, in that it has its own program counter and local register state. These traces, though, are much more speculative in nature than the threads on an SMT processor. Despite the differences, SMT and trace processors are not entirely orthogonal. Multithreading could be applied to a trace processor, or an SMT could be augmented with a trace cache and value prediction.

3.4.4 Dataflow architectures

Dataflow architectures [26, 46, 49, 58, 96, 106, 108] rely on dynamic scheduling and fine-grain synchronization to expose large amounts of parallelism. Although these architectures have the ability to hide long, unpredictable latencies, their performance has lagged behind conventional processors. Dataflow architectures require complex hardware and may not be as effective at exploiting instruction-level parallelism. Many dataflow concepts, however, have appeared in modern, out-of-order processors that rely on dynamic scheduling.

Other architectures, such as P-RISC [90] and *T [91], have also tried to combine dataflow techniques with traditional von Neumann architectures. The P-RISC architecture extends a RISC-like processor by adding fine-grain multithreading and instructions that provide light-weight, dataflow capabilities.

Papadopoulos and Traub [97] revisit the Monsoon dataflow architecture from the perspective of multithreading. They retain the synchronization efficiencies of the original architecture, while trying to improve the performance of sequential execution.

3.5 Chapter summary

This chapter discussed several parallel architectures that rely on thread-level parallelism to increase performance. In many of these architectures, threads share processor resources at a relatively coarse-granularity, and this sharing may cause interference in the cache or other structures. In a simultaneous multithreaded processor, sharing occurs at a much finer-granularity, and has the potential to aggravate the degree of interference. This dissertation extends the prior work by exploring this issue in depth.

In the rest of this dissertation, we investigate the implications of fine-grain resource sharing in SMT processors and examine techniques for managing threads so that shared resources can be used more efficiently. We begin this investigation in the next chapter, by first evaluating the performance benefits of fine-grain, dynamic resource sharing.

Chapter 4

Converting thread-level parallelism to instruction-level parallelism

This chapter explores the benefits of dynamic resource sharing on simultaneous multithreaded processors. We examine the use of instruction-level parallelism (ILP) and thread-level parallelism (TLP), in the context of simultaneous multithreading and single-chip multiprocessing. In particular, we focus on the interaction between resource partitioning and both ILP and TLP.

Instruction-level and thread-level parallelism are heavily exploited by contemporary computer systems. Although they correspond to different granularities of parallelism, ILP and TLP are fundamentally identical; they both identify independent instructions that can execute in parallel and therefore can utilize parallel hardware. Wide-issue superscalar processors exploit ILP by executing multiple instructions from a single program in a single cycle. Single-chip multiprocessors (MP) exploit TLP by executing instructions from different threads in parallel on different processors. Unfortunately, neither parallel-processing style is capable of adapting to dynamically-changing levels of ILP and TLP, because the hardware enforces the distinction between the two types of parallelism. A single-chip multiprocessor must statically partition its resources among the multiple CPUs (see Figure 1.1); if insufficient TLP is available, some of the processors will remain idle. A superscalar executes only a single thread; if insufficient ILP exists, much of that processor's multiple-issue hardware will be wasted.

In contrast, simultaneous multithreading is very attractive for parallel processing because of its ability to use thread-level parallelism and instruction-level parallelism interchangeably. By permitting multiple threads to share the processor's functional units simultaneously, an SMT processor accommodates variations in both ILP and TLP. When a program has only a single thread, i.e., it lacks TLP, all of the SMT processor's resources can be dedicated to that thread; when more TLP exists, this parallelism can compensate for a lack of per-thread ILP. An SMT processor can uniquely exploit whichever type of parallelism is available, thereby utilizing the functional units more effectively to achieve the goals of greater throughput and significant program speedups.

Our investigation of parallel program performance on an SMT is motivated by the results of our previous work [133, 134]. In [134], we used a multiprogrammed workload to assess the potential of SMT on a high-level architectural model and favorably compared total instruction throughput on an SMT to several alternatives: a superscalar processor, a fine-grained multithreaded processor, and a single-chip, shared-memory multiprocessor. In [133], we presented a microarchitectural design that demonstrated that this potential can be realized in an implementable SMT processor. In those two studies, the multiprogrammed workload provided plenty of TLP, because each thread corresponded to an entirely different application. In parallel programs, however, threads execute code from the same application and may share code and data. In some cases this sharing can be beneficial, but in other circumstances, these threads may exacerbate resource contention, as they compete for the same shared structures.

These applications are therefore especially appropriate for exploring resource utilization in future parallel architectures, where greatly-increased chip densities will permit several alternative on-chip, parallel designs. In particular, we compare SMT and small-scale on-chip multiprocessors (MP) in their ability to exploit both ILP and TLP.

This chapter makes several contributions in this respect. First, we identify the hardware bottlenecks that prevent multiprocessors from effectively exploiting ILP in parallel applications. Then, we show that dynamic resource sharing enables an SMT processor to (1) avoid these inefficiencies, because its resources are not statically partitioned, and (2) exploit more threads on a single processor. This is especially advantageous when per-thread ILP is limited. The ease of designing in more thread contexts on an SMT (relative to adding more processors on an MP) allows simultaneous multithreading to expose more thread-level parallelism, further increasing functional unit utilization and attaining greater speedup than a single-chip multiprocessor with comparable execution resources. Finally, these experiments quantify the performance benefits offered by dynamic resource sharing and provide motivation for the remainder of this dissertation.

The rest of this chapter is organized as follows. Section 4.1 describes SMT and MP configurations that we compare. Section 4.2 discusses the methodology used for our experimentation. In Section 4.3, we examine the shortcomings of small-scale multiprocessors and also demonstrate how SMT addresses these flaws. Section 4.4 discusses some implications that SMT has for architects, compiler writers, and operating systems developers. In Section 4.5, we discuss related work, including a comparison with our previous results. Finally, we summarize in Section 4.6.

4.1 Simultaneous multithreading and single-chip multiprocessors

An SMT processor and a small-scale, on-chip multiprocessor have many similarities: e.g., both have large numbers of registers and functional units, on-chip caches, and the ability to issue multiple instructions each cycle. With the continued increases in chip densities, we expect that both SMT and single-chip multiprocessing will be feasible for commodity processors around the year 2000. In this study, we therefore focus on this particular point in the design space. This section describes the SMT and on-chip multiprocessor configurations that we compare.

4.1.1 SMT hardware configuration

The SMT architecture was described in Chapter 2. For the purposes of the comparisons in this chapter, however, we make one modification to the base SMT architecture. We use a less aggressive fetch scheme (the 2.4 scheme from [133] instead of 2.8), which fetches up to four instructions in a single cycle from each of two threads. With a total fetch bandwidth of eight instructions per cycle, this scheme is slightly less aggressive than our base SMT, but allows for a more fair comparison by matching the fetch bandwidth to that of the single-chip MPs, described below. Other details of the SMT configuration are listed in Table 4.1.

4.1.2 Single-chip multiprocessor hardware configurations

We evaluate two single-chip multiprocessor configurations by extending the base superscalar processor described in Section 2.1. For these configurations, we keep on-chip resources approximately similar to the SMT configuration, and in some cases, we give a hardware advantage to the MP. For all MP configurations, the base processor uses the out-of-order processor described earlier and the base pipeline from Figure 2.2 (a). Each MP processor only supports one context; therefore, its register file will be smaller and access will be faster than the SMT. Hence, the shorter pipeline is more appropriate.

We look at both two- and four-processor multiprocessors, partitioning the scheduling unit resources of the multiprocessor CPUs (the functional units, instruction queues, and renaming registers) differently for each case. In the two-processor MP (MP2), each processor receives half of the on-chip execution resources described above, so that the total resources relative to an SMT are comparable (Table 4.1). For a four-processor MP (MP4), each processor contains approximately one-fourth of the chip resources, although

Table 4.1: Per-processor configurations. For the enhanced MP2 and MP4 configurations, the resources that have been increased from the base configurations are listed in boldface. In our processors, not all integer functional units can handle load and store instructions. The integer (load/store) column indicates the total number of integer units and the number of these units that can be used for memory references.

	Per-processor execution resources				
	Number of functional units				
Configuration	**Integer (load/ store)**	**FP**	**Instruction queue size (integer/FP)**	**Number of renaming registers (integer/FP)**	**Number of instructions fetched per cycle**
SMT	6 (4)	4	32/32	100/100	8
MP2	3 (2)	2	16/16	50/50	4
MP4	2 (1)	1	16/16	25/25	2
MP2fu	**6 (4)**	**4**	16/16	50/50	4
MP2q	3 (2)	2	**32/32**	50/50	4
MP2r	3 (2)	2	16/16	**146/146**	4
MP2a	**6 (4)**	**4**	**32/32**	**146/146**	4
MP4fu	**3 (2)**	**2**	16/16	25/25	2
MP4q	2 (1)	1	**32/32**	25/25	2
MP4r	2 (1)	1	16/16	**57/57**	2
MP4a	**3 (2)**	**2**	**32/32**	**57/57**	2

we rounded up when necessary to give the MP the benefit of the doubt. (For example, MP4 has a total of 8 integer functional units, in contrast to the 6 available to MP2 and SMT). The issue width for each processor in these two MP models is indicated by the total number of functional units. Note that even within the MP design space, these two alternatives (MP2 and MP4) represent an interesting tradeoff between TLP and ILP. The two-processor machine can exploit more ILP, because each processor has more functional units than its MP4 counterpart, while MP4 has additional processors to take advantage of more TLP.

In this study, we also evaluate several other MP configurations, as shown in Table 4.1. These configurations have an increased amount of hardware and are designed to reduce bottlenecks in resource usage in order to improve aggregate performance.

MP2fu (MP4fu), MP2q (MP4q), and MP2r (MP4r) address bottlenecks of functional units, instruction queues, and renaming registers, respectively. MP2a increases all three of these resources, so that the total execution resources of each processor are equivalent to a single SMT processor. (MP4a is similarly augmented in all three resource classes, so that the entire MP4a multiprocessor also has twice as many resources as our SMT.)

Note that in the base comparisons, we equalize the number of renaming registers between the MPs and SMT (100 each). For some of the enhanced configurations, however, we give the MPs an additional benefit by equalizing the number of *physical* registers. Our SMT processor has 356 physical registers (32 registers/context * 8 contexts + 100 renaming registers), but only 100 are available, even if only two threads are active. For the MP2r, MP2a, MP4r, and MP4a configurations, we increase the number of renaming registers so that they also have 356 physical registers. For example, in the MP2r and MP2a configurations, each MP processor has 146 renaming registers + 32 architectural registers, or a total of 2*(32+146) = 356.

4.1.3 Synchronization mechanisms and memory hierarchy

SMT has three key advantages over multiprocessing: (1) flexible use of ILP and TLP, (2) fast synchronization, and (3) the potential benefits of a shared L1 cache. This study focuses on SMT's ability to exploit the first advantage by determining the costs of partitioning execution resources; we therefore allow the multiprocessor to use SMT's synchronization mechanisms and cache hierarchy to avoid tainting our results with effects of the latter two.

We implement a set of synchronization primitives for thread creation and termination, as well as hardware blocking locks. Because the threads in an SMT processor share the same scheduling core, inexpensive hardware blocking locks can be implemented in a synchronization functional unit, as described in Section 2.3. This cheap synchronization is not available to multiprocessors, because the distinct processors cannot share functional units. In our workload, most inter-thread synchronization is in the form of barrier synchronization or simple locks, and we found that synchronization time is not critical to performance. Therefore, we allow the MPs to use the same cheap synchronization techniques, so that our comparisons are not colored by synchronization effects.

The entire cache hierarchy, including the L1 caches, is shared by all threads in an SMT processor. Multiprocessors typically do not use a shared L1 cache to exploit data sharing between parallel threads. Each processor in an MP usually has its own private cache (as in the commercial multiprocessors described in [59, 109, 111, 119]), and therefore incurs some coherence overhead when data sharing occurs. Because we allow the MP to use the SMT's shared L1 cache, this coherence overhead is eliminated. Although multiple threads may have working sets that interfere in a shared cache, a previous study showed that inter-thread cache interference was not a problem for the workload in this study [76].

4.1.4 Comparing SMT and MP

In comparing the total hardware dedicated to our multiprocessor and SMT configurations, we have not taken into account the chip area required for buses or the cycle-time effects of a wider-issue machine. In our study, the intent is not to claim that SMT has an absolute "x percent" performance advantage over MP, but instead to demonstrate that SMT can overcome some fundamental limitations of multiprocessors, namely, their inability to exploit changing levels of ILP and TLP. We believe that in the target design space we are studying, the intrinsic flaws resulting from resource partitioning in MPs will limit their effectiveness relative to SMT, even taking into consideration cycle time. (We will discuss this further in Section 4.4).

4.2 Methodology

4.2.1 Why parallel applications?

While our previous work [133, 134] demonstrated SMT's effectiveness for multiprogrammed workloads, those studies did not examine parallel applications. Parallel and multiprogrammed workloads are similar in that they both provide thread-level parallelism, but they compete for shared hardware resources differently.

Because a multiprogrammed workload does not share memory references across threads, it places more stress on the caches. Furthermore, its threads have different instruction execution patterns, potentially causing interference in branch prediction hardware. On the other hand, multiprogrammed workloads are less likely to compete for identical functional units.

In parallel programs, however, threads execute code from the same application, synchronize, and share data and instructions. These programs place different demands on an SMT than a multiprogrammed workload; for example, because parallel threads often execute the same code at the same time (the Single Program Multiple Data model), they may exacerbate resource bottlenecks.

While parallel applications may have the benefit of sharing the caches and branch prediction hardware, they are an interesting and different test of SMT for three reasons.

First, unlike the multiprogrammed workload, all threads in a parallel application execute the same code, and therefore, have similar execution resource requirements, memory reference patterns, and levels of ILP. Because all threads tend to have the same resource needs at the same time, there is potentially more contention for these resources compared to a multiprogrammed workload. For example, a particular loop may have a large degree of instruction-level parallelism, so each thread will require a large number of renaming registers and functional units. Because all threads have the same resource needs, they may exacerbate or create bottlenecks in these resources. Parallel applications are therefore particularly appropriate for this study, which focuses on these execution resources.

Second, parallel applications illustrate the promise of SMT as an architecture for improving the performance of single applications. By using threads to parallelize programs, SMT can improve processor utilization, but more importantly, it can achieve program speedups. Finally, parallel applications are a natural workload for traditional parallel architectures, and therefore serve as a fair basis for comparing multiprocessors to SMT. For the sake of comparison, in Section 4.5, we will also briefly compare our parallel results with the multiprogrammed results from [133].

4.2.2 Workload

Our workload of parallel programs includes both explicitly- and implicitly-parallel applications (Table 4.2). Five of our benchmarks (FFT, LU, radix, water-nsquared, and water-spatial) are explicitly-parallel programs from the SPLASH-2 suite [146], which are built on the Argonne National Laboratories parallel macro library [10]. Tomcatv and hydro2d from SPEC92 [27], as well as shallow and linpack, are implicitly-parallel programs for which we use the SUIF compiler [51] to extract loop-level parallelism. SUIF generates transformed C output files that contain calls to a parallel runtime library to create threads and execute loop iterations in parallel. In each application, all threads share the same address space, but each thread has its own private data and stack, which are stored in a distinct location in the address space.

For all programs in our workload, the Multiflow trace scheduling compiler [79] does the back-end compilation and generates DEC Alpha object files. The compiler generates highly-optimized code using aggressive static scheduling, loop unrolling, and other ILP-exposing optimizations, so that single-thread performance is maximized. We also ported versions of the ANL and SUIF runtime libraries to our SMT architecture. These libraries were then linked with the Alpha object files to create executables.

Many multiprocessor studies look only at the parallel portions of a program for

Table 4.2: Benchmark suite.

Program	Language	Description
FFT	C	Complex, 1-dimensional fast fourier transform
hydro2d	Fortran	Solves hydrodynamical Navier- Stokes equations to solve galactical jets
linpackd	Fortran	Linear systems solver
LU	C	LU factorization
radix	C	Integer radix sort
shallow	Fortran	Shallow water benchmark
tomcatv	Fortran	Vectorized mesh generation program
water-nsquared	C	Molecular dynamics N-body problem, partitioned by molecule
water-spatial	C	Molecular dynamics N-body problem, using 3-D spatial data structure

measuring speedups; we look at the execution of the entire program, including the sequential portions, for two reasons. First, a parallel program typically has a section of sequential code. In order to obtain a complete performance picture, it should be included (Amdahl's Law). Second, performance on the sequential sections is an indication of an architecture's effectiveness on applications that cannot be efficiently parallelized. Processor architectures need not sacrifice single-thread performance for good parallel performance. Because of its resource partitioning, a multiprocessor typically cannot get good single-thread performance; SMT can, by taking better advantage of ILP, even in sequential sections.

4.2.3 Simulation framework

Our simulator measures the performance of these benchmarks on the multiprocessor and SMT configurations described in Section 4.1. For each application on each of the hardware configurations, we look at how our results vary when we change the number of threads used (1 or 2 threads for MP2; 1, 2, or 4 for MP4, and up to 8 for SMT). To distinguish between the MP configurations and the number of threads running on them, we use the designation MPx.Ty, where x refers to the particular hardware configuration (as named in Table 4.1), and y is the total number of threads that are running on the multiprocessor. In our MP experiments, we limit each MP processor to a single thread. A user-level threads package could be used to execute more than 1 thread, but without hardware support for multithreading, context switch overhead would overwhelm the performance benefits of software multithreading.

For both the SMT and MP architectures, the simulator takes unmodified Alpha executables and uses emulation-based, instruction-level simulation to model the processor pipelines, the TLBs, and the entire memory hierarchy. The memory hierarchy in our processor consists of three levels of cache, with sizes, latencies, and bandwidth characteristics as shown in Table 4.3. We model the cache behavior, as well as the contention at the L1 banks, L2 banks, L1-L2 bus, and L3 bank. Recall that we use the same memory hierarchy for both the SMT and MP architectures, in order to focus on the effects of partitioning execution resources.

For branch prediction, we use a 256-entry, 4-way set associative branch target buffer and a 2K x 2-bit pattern history table [12]. These structures are shared by all running threads (even if less than 8 are executing), allowing more flexible and therefore higher interthread utilization. Equally important, these structures are fully available even if only a single thread is executing. Of course, the competition for the shared resources among the threads may increase the frequency of cache misses and branch mispredictions. We discuss and quantify these effects in Section 4.5.1.

Table 4.3: Memory hierarchy details. Instruction and data TLBs are both 64-entry, fully associative buffers. TLB misses require two full memory accesses.

Parameter	**L1 I-cache**	**L1 D-cache**	**L2 cache**	**L3 cache**
Size	32 KB	32 KB	256 KB	8 MB
Associativity	direct-mapped	direct-mapped	4-way	direct-mapped
Line size (bytes)	64	64	64	64
Banks	8	8	8	1
Transfer time/bank	1 cycle	1 cycle	1 cycle	4 cycles
Accesses/cycle	2	4	1	1/4
Cache fill time (cycles)	2	2	2	8
Latency to next level	6	6	12	62

4.3 Exploiting parallelism

4.3.1 SMT and multiprocessing performance results

Performance on parallel applications is maximized when both instruction-level and thread-level parallelism can be effectively used by the architecture. In order to understand the benefits of both types of parallelism, we first compare the average speedup (Figure 4.1) and throughput (Table 4.4) for simultaneous multithreading and the two multiprocessors, MP2 and MP4, as more threads are used in the programs. With 1 and 2 threads, MP2 outperforms MP4 (by more than 40%), because its configuration of hardware resources allows each processor to exploit more ILP. MP4, on the other hand, has two additional processors to take advantage of TLP. When each MP configuration is given its maximum number of threads (2 for MP2 and 4 for MP4), MP4.T4 has a 1.76 speedup, while MP2.T2 only reaches 1.63.

Because of static partitioning, both multiprocessor configurations fall short of the performance possible with simultaneous multithreading. When all three architectures use the same number of threads, SMT gets the best speedups, because the dynamic resource sharing permits better use of ILP. When more threads are used, simultaneous multithreading improves speedups even further by taking advantage of both ILP *and* TLP. With 4 threads, SMT has a 2.17 speedup over the base configuration, and with all 8 contexts, its average speedup reaches 2.68.

The key insight is that SMT has the unique ability to ignore the distinction between TLP and ILP. Because resources are not statically partitioned, SMT avoids the MP's hardware constraints on program parallelism.

The following three subsections analyze the results from Table 4.4 in detail to understand why inefficiency exists in multiprocessors.

4.3.2 Measuring MP inefficiency

We measured the amount of resource inefficiency due to static partitioning by counting the cycles in which the following two conditions hold: (1) a processor runs out of a particular resource, and (2) that same resource is unused on another processor. If all processors run out of the same resource in a given cycle, then the total number of resources, not the partitioning, is the limiting factor.

The following three classes of metrics allow us to analyze the performance of our parallel applications on multiprocessor configurations.

Issue limits: The number of functional units in each processor determines how many instructions of a particular type can be issued in a single cycle. We identify the number

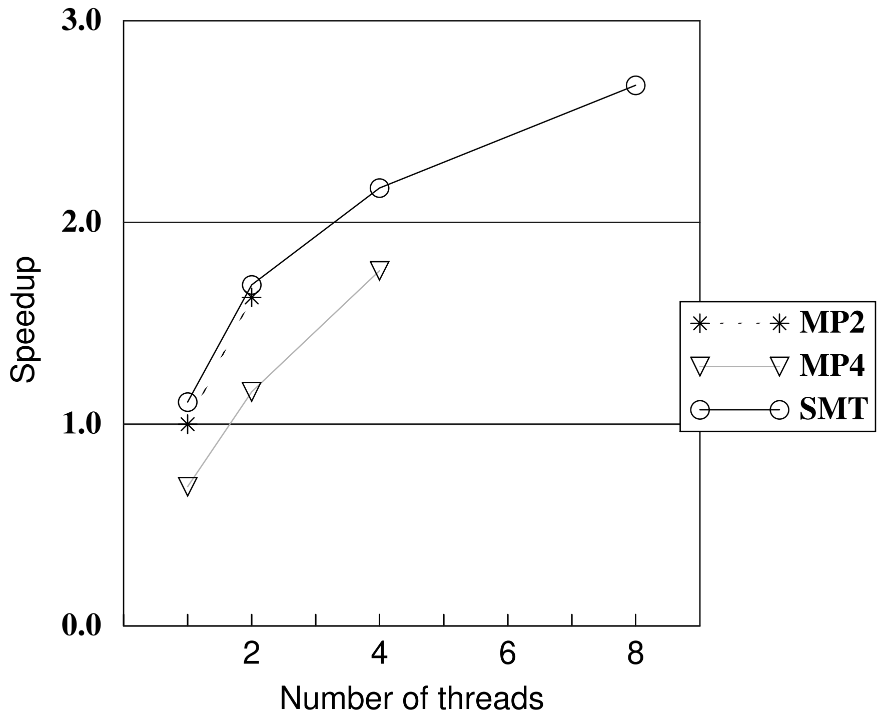


Figure 4.1: Speedups normalized to MP2.T1.

Table 4.4: Throughput comparison of MP2, MP4, and SMT, measured in instructions per cycle.

Configuration	Number of threads			
	1	**2**	**4**	**8**
MP2	2.08	3.32	---	---
MP4	1.38	2.25	3.27	---
SMT	2.40	3.49	4.24	4.88

of cycles in which both (1) an instruction cannot be issued because there are insufficient functional units on a processor, and (2) an appropriate functional unit is free on another processor. We subcategorize this metric based on the particular functional unit type:

integer issue, load/store issue, and *FP issue*.

Too few renaming registers: When one processor uses up its entire pool of renaming registers, we increment this metric if another processor still has available registers. Metrics for integer and FP registers are specified as *integer reg* and *FP reg*.

Instruction queue full: The instruction queue full condition refers to the situation when one processor incurs a fetch stall because its instruction queue is full, while another processor still has free slots in its queue. We keep counts for both queues, *integer IQ* and *FP IQ*.

In our results, we present the frequency of these events as a percentage of total execution cycles. The metrics are independent, i.e., more than one of them can occur in each cycle; therefore we also include a metric that identifies the percentage of cycles in which *any* inefficiency occurs to account for overlaps between the metrics. While a reduction in these inefficiency metrics does not directly correlate with a performance improvement, the metrics are useful for identifying system bottlenecks introduced by partitioning resources that are preventing better program performance.

4.3.3 Exploiting ILP

We examine in detail the performance of the multiprocessor scheduling unit (instruction issue, functional units, instruction queues, and renaming registers) by graphing the frequency of the metrics for MP2.T2 and MP4.T4 in Figures 4.2 and 4.3, respectively. The high frequencies illustrate that (1) many of the programs have per-thread ILP that exceeds the amount that can be supported by a single MP2 or MP4 processor, and (2) there is a large degree of inefficiency for all programs but one.

For most benchmarks, instruction issue metrics highlight the biggest inefficiencies. For MP2.T2 (MP4.T4), the integer and load/store units are used inefficiently in 23.5% (18.0%) and 17.3% (38.7%) of total execution cycles, respectively. The third issue metric, FP issue, is a much larger problem for MP4.T4 than MP2.T2 (25.1% vs. 7.7%), because MP4 only has one FP functional unit and most of the applications are floating-point intensive.

For shallow and tomcatv, however, the FP renaming registers are also a large contributor to inefficiency. These two benchmarks have two of the highest average memory latencies, which extends register lifetimes. This heightened register pressure increases the likelihood that an individual processor will run out of FP renaming registers. In contrast, the integer renaming registers and the integer IQ are not a frequent source of waste in our programs, again, because the applications tend to be more floating-point intensive.

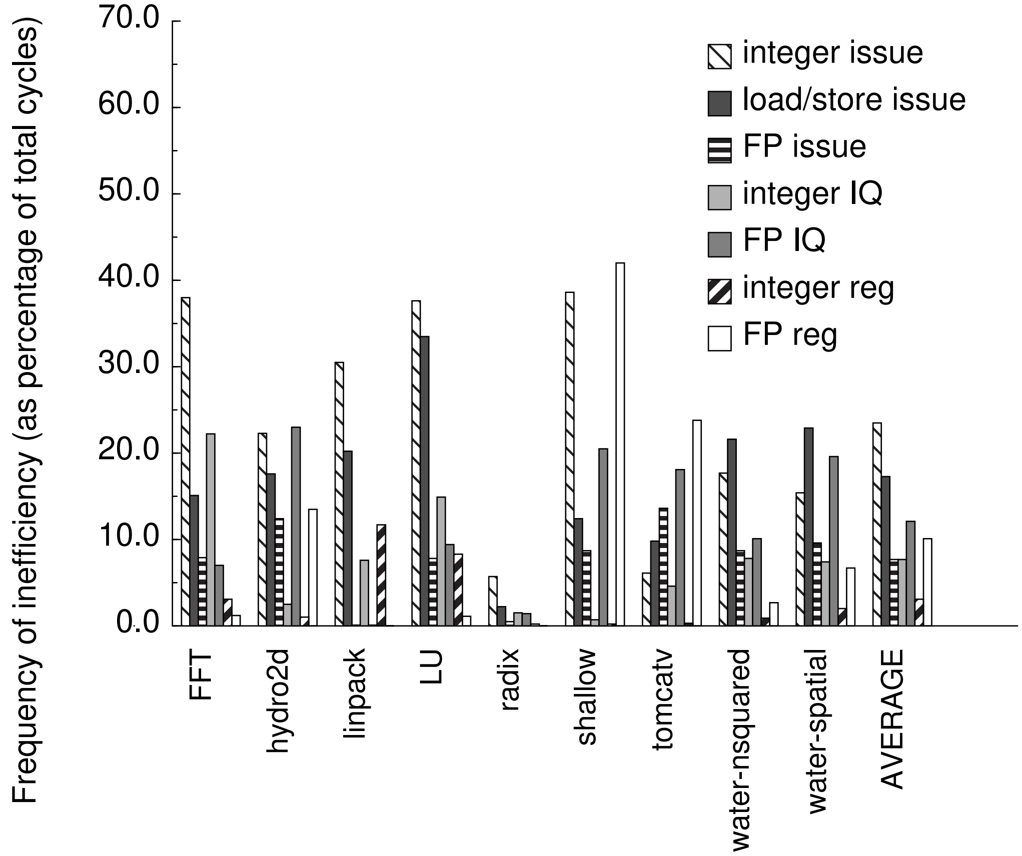


Figure 4.2: Frequencies of partitioning inefficiencies for MP2.T2.

Each of the seven inefficiency metrics indicates the percentage of total execution cycles in which (1) one of the processors runs out of a resource, and (2) that same resource is available on the other processor. In a single cycle, more than one of the inefficiencies may occur, so the sum of all 7 metrics may be greater than 100%.

For radix, the inefficiency metrics are much smaller, with no metric greater than 5.7% in MP2.T2. Inefficiencies are infrequent, because ILP (an average of 0.46 instructions per cycle per thread) is rarely high enough to require more resources than are available on a single processor. During part of the program, instructions are almost exclusively floating point operations. Because of their long latencies and the dependences between them, these instructions quickly fill the FP IQs in both processors, causing them to stall.

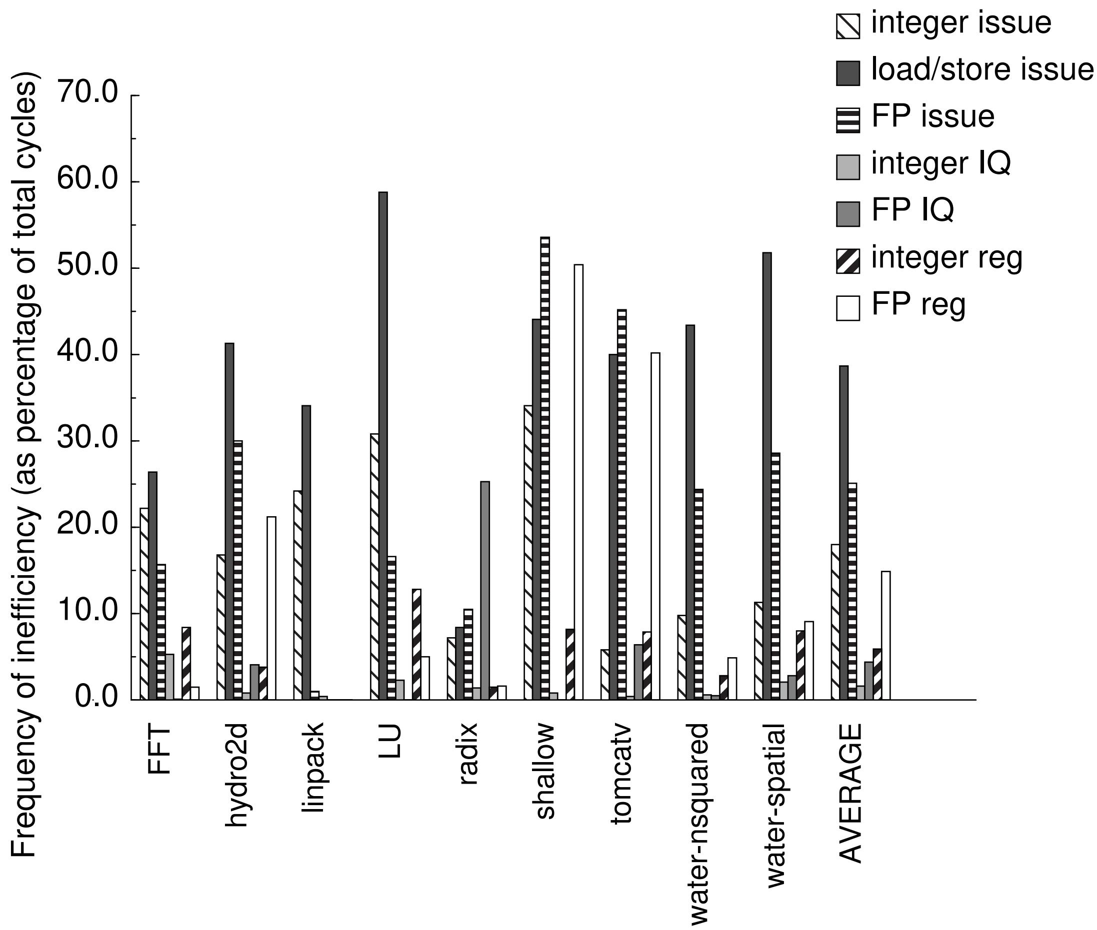


Figure 4.3: Frequencies of partitioning inefficiencies for MP4.T4.

Addressing resource inefficiencies

In this section, we examine the performance of two sets of enhanced MP configurations, which are designed to address the resource inefficiencies indicated by Figures 4.2 and 4.3. The first set includes configurations that address bottlenecks in each of the resources. As shown in Table 4.1, the MP2fu (MP4fu), MP2q (MP4q), and MP2r (MP4r) configurations have twice as many resources dedicated to functional units, queues, and renaming registers, respectively, relative to the base MP2 (MP4) configuration. These configurations determine whether there are any single resource performance bottlenecks in an MP system. Any reasonable MP implementation would be

Table 4.5: Average frequencies for inefficiency metrics (MP2 configurations). Each configuration focuses on a particular subset of the metrics, printed in boldface. Notice that the boldface percentages are small, showing that each configuration successfully addresses the targeted inefficiencies. The inefficiency metrics can overlap, i.e., more than one bottleneck can occur in a single cycle; consequently, in the last row, we also show the percentage of cycles in which at least one of the inefficiencies occur.

	Configuration				
Metric	**MP2.T2**	**MP2fu.T2**	**MP2q.T2**	**MP2r.T2**	**MP2a.T2**
integer issue	23.5%	**0.4%**	27.9%	23.6%	**1.3%**
load/store issue	17.3%	**1.1%**	18.0%	18.1%	**1.8%**
FP issue	7.7%	**0.9%**	8.9%	7.9%	**1.4%**
integer IQ	7.7%	5.7%	**2.2%**	9.5%	**2.8%**
FP IQ	12.1%	11.3%	**1.1%**	15.6%	**4.1%**
integer reg	3.1%	3.1%	5.4%	**0.0%**	**0.0%**
FP reg	10.1%	10.3%	17.0%	**0.0%**	**0.0%**
any inefficiency	58.9%	31.9%	55.9%	56.0%	12.0%

Table 4.6: Average frequencies for inefficiency metrics (MP4 configurations).

	Configuration				
Metric	**MP4.T4**	**MP4fu.T4**	**MP4q.T4**	**MP4r.T4**	**MP4a.T4**
integer issue	18.0%	**1.5%**	18.0%	21.6%	**2.5%**
load/store issue	38.7%	**2.7%**	38.2%	42.2%	**3.8%**
FP issue	25.1%	**2.3%**	24.5%	27.6%	**3.2%**
integer IQ	1.6%	1.3%	**0.0%**	4.2%	**0.7%**
FP IQ	4.4%	7.5%	**0.0%**	10.4%	**1.4%**
integer reg	5.9%	7.1%	1.2%	**0.2%**	**1.3%**
FP reg	14.9%	23.1%	6.5%	**3.9%**	**6.3%**
any inefficiency	67.9%	26.5%	67.6%	71.4%	16.6%

designed in a balanced manner to avoid a single resource bottleneck. Tables 4.5 and 4.6 show the average frequencies for each of the seven inefficiency metrics for our enhanced configurations. Each configuration targets a subset of the metrics and virtually eliminates

Table 4.7: Speedups for MP2 and MP4 configurations relative to 1 thread MP2.

Configuration	Speedups relative to 1 thread MP2	Configuration	Speedups relative to 1 thread MP2
MP2.T2	1.63	MP4.T4	1.76
MP2fu.T2	1.66	MP4fu.T4	1.78
MP2q.T2	1.66	MP4q.T4	1.75
MP2r.T2	1.68	MP4r.T4	1.85
MP2a.T2	1.80	MP4a.T4	1.92

the inefficiencies in that subset. Speedups, however, are small: Table 4.7 shows that these enhanced MP2 (MP4) configurations get speedups of 3% (4%) or less relative to MP2.T2 (MP4.T4). Performance is limited, because removing a particular bottleneck merely shifts it to another hardware resource (Tables 4.5 and 4.6).

Of all the resources, renaming registers have the greatest impact on program performance, for two related reasons. First, when all renaming registers are in use, the processor must stop fetching, which keeps an entire block of instructions from entering the processor pipeline. In contrast, a lack of available functional units will typically only delay a particular instruction until the next cycle. Second, a lack of registers may actually prevent fetching for several cycles, because the lifetimes of renaming registers are very long (see Chapter 7) and because registers are not deallocated until the end of the pipeline. In contrast, instruction queue entries and functional units can become available much more quickly as they are held for a shorter period of time and are released in the middle of the pipeline.

Although some resources may be more critical than others, the variety of inefficiencies means that effective use of ILP (i.e., better performance) cannot be attained by increasing a single set of resources alone. Therefore, in the second set of MP enhancements (MP2a and MP4a), we increase all execution resources to address the entire range of inefficiencies. Tables 4.5 and 4.6 show that the MP2a and MP4a configurations are indeed able to reduce inefficiencies in all metrics, thereby achieving increased speedups (Table 4.7). With its enhanced configuration, the performance of MP2a.T2 surpasses SMT.T2 (1.80 speedup vs. 1.69). Naturally, this is not a surprising result, given that MP2a has twice the execution resources of our single SMT processor and the SMT processor is limited to executing only two threads.

However, when all architectures are allowed to exploit all of their resources (i.e., the

SMT uses all 8 threads), simultaneous multithreading still maintains a significant performance edge, achieving a speedup of 2.68 versus the 1.80 and 1.92 speedups for MP2a and MP4a, respectively (Table 4.7 and Figure 4.1). SMT is a more cost-effective solution for improving performance because dynamic resource sharing permits much more efficient resource utilization.

4.3.4 Exploiting TLP

While individual processors in a superscalar MP can exploit ILP in a single thread, the architecture as a whole is specifically designed to obtain speedups by using thread-level parallelism. The amount of TLP that can be exploited by an MP architecture is limited to the number of processors. Consequently, MP4 is better suited (than the two-processor MP) for this type of parallelism. The MP4 configuration trades off the ability to exploit large amounts of ILP for the opportunity to exploit more TLP.

In Figure 4.4, we compare MP2 and MP4 speedups as we increase the number of threads. On average, MP4.T4 outperforms MP2.T2, but for individual programs, the relative performance of the two varies, depending on the amount of per-thread ILP in the benchmark. (We approximate the level of ILP by using the single-thread throughput data shown in Table 4.8.) In the programs with higher per-thread ILP (LU, linpack, FFT, and water-nsquared), MP2 has the performance edge over MP4, because there is sufficient ILP to compensate for having fewer processors. When programs lack ILP (the other five applications), however, MP4 uses the additional processors to exploit TLP, compensating for individual processors that have limited execution resources. The best examples are hydro2d and the ILP-starved radix, where MP4.T4 gets speedups of 1.95 and 3.46, while MP2.T2 only gets speedups of 1.79 and 1.95, respectively.

TLP and ILP are identical in that they expose independent instructions that can be executed in parallel, and can therefore take advantage of parallel hardware. In multiprocessors, however, static resource partitioning forces a tradeoff; TLP can only be exploited by adding more processors, while ILP can only be exploited by adding more resources on each processor. Multiprocessors lack the flexibility to adapt to fluctuations in both forms of parallelism, and their performance suffers when resource partitioning fails to match the ILP and TLP characteristics of a program.

4.3.5 Effectively using parallelism on an SMT processor

Rather than adding more execution resources (like the MP2a and MP4a configurations) to improve performance, SMT boosts performance and improves utilization of existing resources by using parallelism more effectively. Unlike

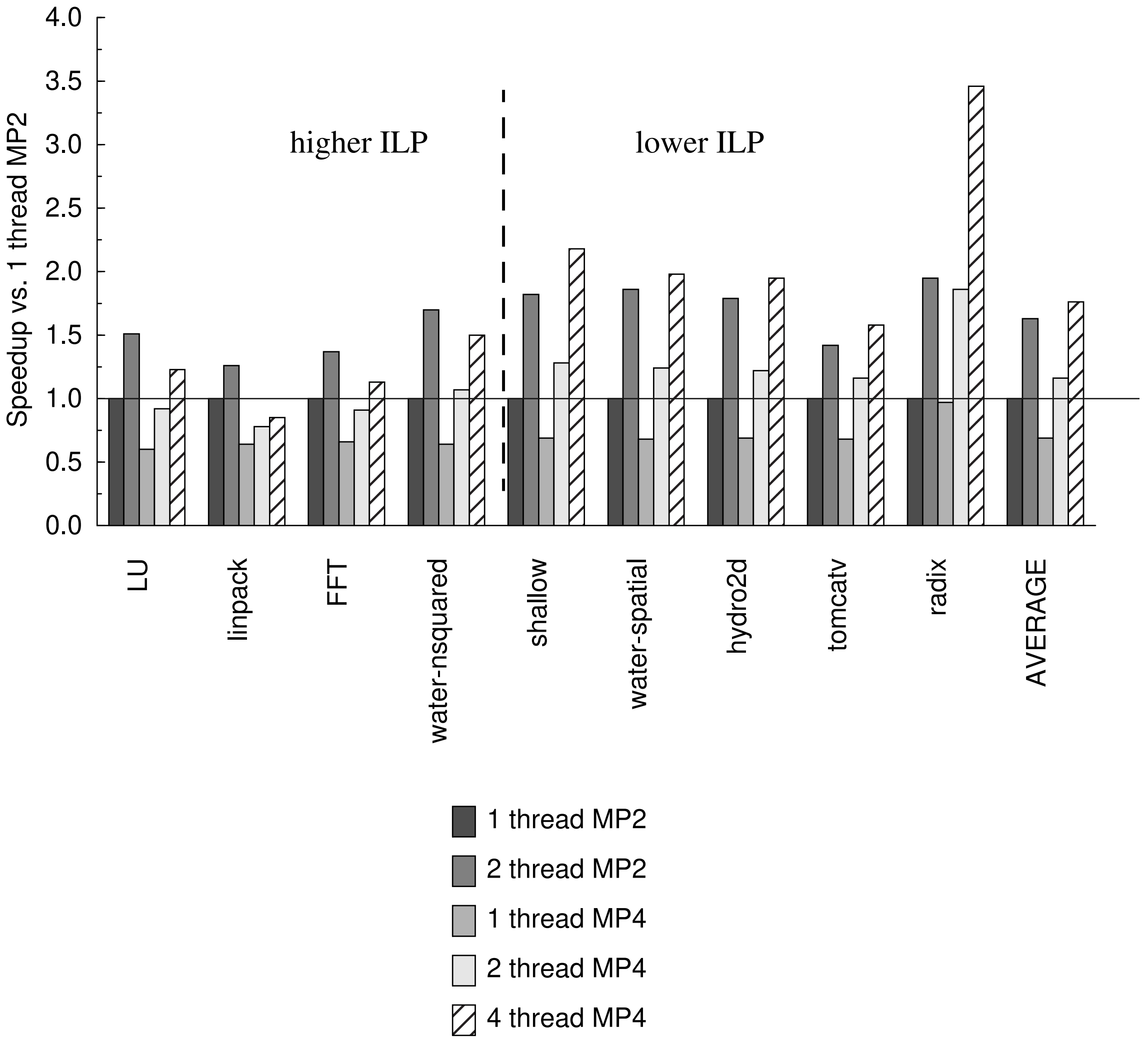


Figure 4.4: MP2 and MP4 speedups vs. one-thread MP2 baseline.

Programs are listed in descending order based on the amount of ILP in the program. MP2.T2 outperforms MP4.T4 in the programs to the left of the dashed line. MP4.T4 has the edge for those on the right.

multiprocessors which suffer from rigid partitioning, simultaneous multithreading permits dynamic resource sharing, so that resources can be flexibly partitioned on a per-cycle basis to match the ILP and TLP needs of the program. When a lot of ILP is available, the processor can exploit this parallelism because a single thread has access to all processor resources. When little ILP is available, SMT can compensate by executing several threads together. Table 4.9 shows that simultaneous multithreading exploits the

Table 4.8: Benchmark throughput (instructions per cycle) for SMT.T1. The SMT.T1 configuration gives an indication of the amount of instruction-level parallelism in each program, using the decomposed application running one thread.

Benchmark	IPC
LU	4.01
linpack	3.00
FFT	2.65
water-nsquared	2.60
shallow	2.27
water-spatial	2.25
hydro2d	2.21
tomcatv	2.10
radix	0.47

Table 4.9: SMT speedups relative to 1 thread MP2. Benchmarks are listed in descending order, based on the amount of single-thread ILP.

Benchmark	SMT.T1	SMT.T2	SMT.T4	SMT.T8
LU	1.15	1.76	1.82	1.89
linpack	1.16	1.39	1.34	1.15
FFT	1.11	1.47	1.61	1.64
water-nsquared	1.07	1.71	2.18	2.42
shallow	1.18	1.89	2.03	2.80
water-spatial	1.04	1.84	2.50	2.71
hydro2d	1.13	1.81	2.38	2.72
tomcatv	1.10	1.38	2.00	2.44
radix	1.01	1.95	3.65	6.36
Average	1.11	1.69	2.17	2.68

two types of parallelism to improve speedups significantly. As more threads are used, speedups increase (up to 2.68 on average with 8 threads), exceeding the performance gains attained by the enhanced MP configurations.

The degree of SMT's improvement varies across the benchmarks, depending on the

Table 4.10: SMT throughput in instructions per cycle.

Benchmark	SMT.T1	SMT.T2	SMT.T4	SMT.T8
LU	4.01	5.26	5.43	5.70
linpack	3.00	3.78	4.01	4.04
FFT	2.65	3.53	3.88	3.92
water-nsquared	2.60	4.17	5.34	5.94
shallow	2.27	3.64	3.90	5.38
water-spatial	2.25	3.98	5.41	5.87
hydro2d	2.21	3.54	4.68	5.37
tomcatv	2.10	2.63	3.83	4.71
radix	0.47	0.91	1.71	3.02
Average	2.40	3.49	4.24	4.88

Table 4.11: SMT throughput (measured in instructions per cycle) in the computation phase of SPLASH-2 benchmarks.

Benchmark	SMT.T1	SMT.T2	SMT.T4	SMT.T8
LU	3.50	6.09	6.41	6.83
FFT	2.70	4.63	5.77	5.95
water-nsquared	2.60	4.21	5.42	6.07
water-spatial	2.24	4.01	5.50	5.99
radix	3.02	3.96	4.52	4.73
Average	2.81	4.58	5.52	5.91

amount of per-thread ILP. The five programs with the least ILP (radix, tomcatv, hydro2d, water-spatial, and shallow) get the five largest speedups for SMT.T8, because TLP compensates for low ILP; programs that already have a large amount of ILP (LU and FFT) benefit less from using additional threads, because resources are already busy executing useful instructions. In linpack, performance tails off after two threads, because the granularity of parallelism in the program is very small. The gain from parallelization is outweighed by the overhead of parallelization (not only thread creation, but also the work required to set up the loops in each thread).

For all programs, as we use more threads, the combined use of ILP and TLP increases processor utilization. On average, throughput reaches 4.88 instructions per cycle, while several benchmarks attain an IPC of greater than 5.7 with 8 threads, as shown in Table 4.10. These results, however, fall short of peak processor throughput (the 8 instructions per cycle limit imposed by instruction fetch and decode bandwidth), in part because they encompass the execution of the entire program, including the sequential parts of the code. In the parallel computation portions of the code, throughput is closer to the maximum. Table 4.11 shows that with 8 threads in the computation phase of the SPLASH-2 benchmarks, average throughput reaches 5.91 instructions per cycle and peaks at 6.83 IPC for LU.[1]

Simultaneous multithreading thus offers an alternative architecture for parallel programs that uses processor resources more cost-effectively than multiprocessors. SMT adapts to fluctuating levels of TLP and ILP across the entire workload, or within an individual program, to effectively increase processor utilization, and therefore performance.

4.4 Placing these results in context

When interpreting the results of the previous sections, we can make several important observations that are relevant for next-generation architects and compiler and operating systems writers.

Implications for architects

As chip densities continue to increase, architects must decide what is the best use of the additional chip area. In evaluating the merits of SMT and MPs as two potential design alternatives, architects should also recognize two additional factors. First, potential cycle time impacts should be taken into consideration when interpreting the results of this study, which uses total execution cycles to compare SMT and MPs. Our SMT architecture (as described in Chapter 2) is designed in a manner that limits the cycle time impact of SMT vs. a modern out-of-order execution superscalar of similar width, i.e., an extra cycle is taken to access the larger register file. However, the MP2 or MP4 processors could still have a faster cycle time than the 8-wide SMT. Without specific implementations of each of the processors, it is difficult to make detailed quantitative comparisons of the cycle times for 2, 4, and 8-wide issue machines. Our

---

1. In the SPLASH-2 benchmarks, the parallel computation phase is very clearly defined by the application programmers. We did not measure the performance for the parallel phases of the implicitly parallel applications (the others in our benchmark suite), because the delineation is less clear.

experiments, however, show that when both SMT and MPs are at full capacity with a parallel workload (i.e., maximum number of threads -- 8 for SMT, 2 for MP2, and 4 for MP4), SMT outperforms MP2 (MP4) by more than 62% (54%); in order for MPs to overcome SMT's performance advantage, the improvement in MP cycle times would have to be comparable. Although other comparisons can also be made, e.g., SMT.T4 vs. MP.T4, they are less indicative of SMT's true potential, because they restrict SMT's advantage -- its ability to benefit from extra thread-level parallelism.

Second, workload and program parallelism is highly variable, but in most conventional parallel processing architectures, hardware resources are statically allocated at design time. Given a fixed amount of hardware, multiprocessor designers must determine how much should be used to exploit ILP and how much should be dedicated for TLP. If the design does not match the ILP and TLP characteristics of the application, then performance suffers. Given the variability of parallelism both within an application as well as across a workload, it becomes very difficult to hit this "sweet spot" for best performance. A simultaneous multithreading architecture has the ability to adapt to a wider range of applications, by effectively using both ILP and TLP, and therefore potentially simplifies the task of chip area allocation.

Finally, the building block for thread-level parallelism is smaller in an SMT than in a single-chip multiprocessor. if a chip designer wishes to exploit more thread-level parallelism, an SMT processor requires an additional context and register set. In a single-chip MP, an entire CPU is required to exploit more TLP.

Implications for compiler writers

Multiprocessors also complicate the task of compilers and programmers, because ILP and TLP must be extracted in a manner that best matches the allocation of ILP and TLP hardware in the particular MP. The dynamic nature of parallelism in a program may make this difficult or even impossible. In addition to load balancing, the compiler or programmer must also ensure that the levels of parallelism (in addition to the total amount of work to be done) are distributed in a manner that suits the particular machine.

Simultaneous multithreading offers compilers and programmers more flexibility to extract whatever form of parallelism exists, because the underlying architecture can use ILP and TLP equally effectively. Aggressive superscalar or VLIW compilers already use many optimizations to extract large amounts of ILP. An SMT processor gives these compilers yet another mechanism for exploiting parallelism that can be used to fill wide-issue processors. Several researchers have proposed thread-level speculation techniques to improve the performance of sequential applications; these techniques are appropriate for SMT as well [94, 117].

SMT offers an opportunity to expose new parallelization techniques, because of the potential for faster synchronization. In comparison to multiprocessors, even those with shared L2 caches, SMT provides an opportunity for cheaper synchronization and communication, either through a shared L1 cache or through functional units or registers. For example, SMT could enable the parallelization of loops with tight doacross synchronization because of the reduction in synchronization overhead [135]. Applications with high communication-to-computation ratios, like ear from the SPEC92 suite, can also benefit from cheap synchronization via the shared L1 cache [87].

Implications for operating systems developers

Simultaneous multithreading also opens a new set of operating systems issues, in particular thread scheduling and thread priority. With parallel applications, improving processor utilization with additional threads translates to improved program speedups. With multiprogrammed workloads, however, there may be some threads that are more important than others. Priority mechanisms are inherent to most operating systems, and an interesting area of research will be to investigate various schemes to maximize both processor throughput and the performance of high-priority threads. Taken one step further, a multiprogrammed workload might consist of several single-threaded applications, as well as a few parallel programs. The thread scheduler (whether it's the operating system or a runtime system) must manage and schedule the threads and applications in a manner to best utilize both the execution resources and the hardware contexts.

4.5 Related work

This chapter compared SMT with an alternative next-generation architecture, the single-chip multiprocessor. In this section, we discuss several related research topics. We begin by comparing this study to other studies that have examined both SMT and single-chip MPs.

4.5.1 Previous SMT work

As indicated in the previous section, SMT may exhibit different behavior for parallel applications when compared to a multiprogrammed workload, because of different degrees of inter-thread interference. Table 4.12 compares the SMT throughput results from this study (parallel workload) with the multiprogrammed results from one of our previous studies [133]. Note that because the applications themselves are different, and

Table 4.12: Throughput for parallel and multiprogrammed workloads. This table compares the throughput (in instructions per cycle) obtained for the multiprogrammed and parallel workloads, as the number of threads is increased. The last column compares the throughput achieved in the parallel computation phases of the parallel applications.

Number of threads	Throughput: Instructions per cycle		
	multiprogrammed	**parallel**	**parallel computation only**
1	2.2	2.4	2.8
2	4.2	3.5	4.6
4	4.7	4.2	5.5
8	5.4	4.9	5.9

Table 4.13: Branch and jump misprediction rates. The table compares branch and jump misprediction rates for parallel and multiprogrammed workloads, as the number of threads is increased.

Number of threads	Branch misprediction		Jump misprediction	
	parallel	**multiprogrammed**	**parallel**	**multiprogrammed**
1	2.0%	5.0%	0.0%	2.2%
2	2.5%	N/A	0.1%	N/A
4	2.8%	7.4%	0.1%	6.4%
8	2.5%	9.1%	0.0%	12.9%

the simulation methodology is slightly different (mainly a 2MB L3 cache vs. an 8MB L3), we can only make rough comparisons and highlight some differences.

Our first observation is that, on average, there is more single-thread ILP in the parallel workload (2.4 IPC) versus the multiprogrammed workload (2.2 IPC). This is not surprising, because the loop-based nature of most of the parallel applications lends itself to larger degrees of ILP. As more threads are used, however, the parallel applications benefit less than the multiprogrammed workload. This is primarily because the parallel applications also include sequential sections of program execution.

Second, when all threads are being used, the multiprogrammed workload sustains greater throughput than the parallel applications. However, when using only the parallel computation sections of the SPLASH-2 applications (last column reproduced from

Table 4.11), parallel throughput is greater than the multiprogrammed results, for all numbers of threads.

Parallel applications experience less inter-thread interference in the branch hardware, because the distinct threads tend to exhibit similar branching behavior. Table 4.13 compares the parallel applications of this study with the multiprogrammed workload in our previous study [133], in terms of branch and jump misprediction rates. The data indicates that minimal branch prediction interference occurs in parallel applications, relative to a multiprogrammed workload.

Parallel applications also suffer from less inter-thread interference in the memory system, compared to multiprogrammed applications. For the sake of comparison, Table 4.14 presents cache interference statistics from [133]. Although interference occurs in the parallel applications, it is much less significant than in multiprogrammed workloads. These results demonstrate that simultaneous multithreading is particularly well-suited for parallel applications. In Chapter 6, we will examine the issue of cache interference in more depth.

Table 4.14: Comparison of memory system interference for parallel and multiprogrammed workloads. The L2 and L3 cache miss rates are local miss rates. MPCI is the number of misses per 1000 completed instructions.

		parallel			**multiprogrammed**		
Cache	**metric**	**1**	**4**	**8**	**1**	**4**	**8**
L1 I-cache	miss rate	0.0	0.0	0.0	2.5	7.8	14.1
	MPCI	0.0	0.0	0.0	6	17	29
L1 D-cache	miss rate	4.6	8.9	9.3	3.1	6.5	11.3
	MPCI	14	26	27	12	25	43
L2 cache	miss rate	21.0	13.5	16.0	17.6	15.0	12.5
	MPCI	4	4	5	3	5	9
L3 cache	miss rate	22.3	16.0	20.4	55.1	33.6	45.4
	MPCI	1	1	1	1	3	4

Tullsen et al. [134] compared the performance of SMT and various multiprocessor configurations and found that SMT outperforms an MP with comparable hardware for a multiprogrammed workload. In contrast to that study, this chapter compares the architectures using a parallel workload and an implementable (rather than idealized)

SMT architecture, and also identifies the sources of resource waste in multiprocessor configurations. Furthermore, this study also uses speedup, rather than processor utilization, as the primary performance metric. In the parallel applications used in this study, all threads work together toward completing the entire program, so speedup becomes the critical metric.

4.5.2 Single-chip multiprocessors

The Hydra research project exploits thread-level parallelism through the use of multiple CPUs on a single-chip [87, 93]. This approach has the advantage of utilizing simpler processor cores that are easier to design and optimize. As a result, they may be able to achieve higher clock rates. As we have seen in this chapter, however, these architectures sacrifice single-thread performance and are limited by resource partitioning.

To evaluate the multiprocessing approach, Hammond et al. [53] compare the performance of a single-chip multiprocessor to an SMT processor and superscalar. They find that for a single-threaded application, the single-chip multiprocessor lagged behind the other two architectures. When executing a set of parallel workloads (both multiprogrammed and parallel applications), however, the multiprocessor achieves the best performance. There are several differences between their study and ours. First, they find that the performance of simultaneous multithreading is inhibited by two limitations imposed by a unified data cache: an extra cycle for cache access and bank contention. The additional cache access time can often be hidden with multithreading or more aggressive compiler scheduling. (We use a more aggressive compiler in our experiments that exposed more instruction-level parallelism.) The effects of bank contention can also be addressed by either software or hardware. The compiler can restructure application data access patterns to reduce bank contention. Wave-pipelined caches can provide more cache ports to address bank contention, as in the DEC Alpha 21264 [50]. We have found experimentally that a four-bank, wave-pipelined cache minimizes the effect of bank contention for these workloads. Second, their SMT processor did not use the 2.8 ICOUNT fetch scheme (described in Section 2.2.1 and [133]), which can increase SMT throughput by up to 34% over other instruction fetch policies [133]. Third, they use separate memory systems for the SMT and single-chip MPs, where the MP caches have shorter access times. In this chapter, we use a common memory system for both the SMT and single-chip MP architectures to focus on execution resource effects. Fourth, our study measures the overall execution time of the applications, which included some sequential portions, while their parallel workloads have very little, if any, sequential execution time. As discussed in this chapter and reaffirmed by their experiments, an

SMT processor is much more effective than a single-chip MP in these sequential or single-threaded regions. Fifth, their parallel applications are not compiled in an SMT-specific manner. As we will see in subsequent sections, this can affect application performance.

4.5.3 Related SMT work

As discussed in Chapter 3, several other proposed architectures permit multiple threads to issue instructions in the same cycle. While some of these designs permit flexible resource sharing, others partition various pieces of execution hardware among the multiple threads. As a result, they suffer from many of the same resource partitioning problems as a single-chip multiprocessor. (Note that these architectures were discussed in more detail in Chapter 3.)

We first discuss a set of architectures that permit flexible sharing of the processor's functional units, i.e. multiple threads can issue multiple instructions each cycle.

Hirata et al. [55], Yamamoto and Nemirovsky [147], and Prasadh and Wu [101] propose architectures that dynamically packs instructions from different threads. In all three designs, however, the early stages of the pipeline are partitioned. Hirata's architecture includes a separate instruction queue and fetch/decode unit for each thread. When an instruction is ready, the decode unit must arbitrate for an available functional unit. If there are no available functional units, then the ready instruction is placed in a standby station. Yamamoto and Nemirovsky propose a similar architecture, where each thread has its own 8-entry dispatch window and fetch unit. Ready instructions compete for available functional units. Prasadh and Wu also require per-thread register sets and instruction fetch units. A dynamic interleaving unit uses a priority scheme to select operations from the multiple threads.

Although these three architectures permit flexible sharing of processor functional units, they require replicated fetch/decode units and partition the dispatch window/instruction queue. In this chapter, we demonstrated that a partitioned instruction queue (in a single-chip MP) can lead to under-utilization of this resource. Our SMT architecture avoids this problem by sharing the instruction queue between all threads. The instruction queue is fed by a single fetch unit, so that fewer instruction cache ports are required than in the other SMT proposals.

Daddis and Torng [25] combine out-of-order execution with multithreading, but only experimented with a two-threaded processor. Gulati and Bagherzadeh [47] extend an out-of-order superscalar processor by adding hardware for multithreading. Although the architecture is not physically partitioned, the compiler statically partitions the register set

(128 registers) among the threads. Even though the hardware provides register renaming, this static partitioning can limit the effectiveness of the compiler, by restricting ILP and increasing the amount of register spill code.

Partitioned instruction-issue architectures

Several other architectures allow multiple threads to issue each cycle, but partition the issue hardware and functional units. These architectures lack flexible resource sharing, which contributes to resource waste when only a single thread is running. Govindarajan et al. [45] provide multiple execution pipes so that several threads can be executed in a single processing element. During a single cycle, however, only a single instruction is selected from each pipe. Within a single processing element, resources are partitioned between execution pipes. Gunther's Concurro processor provides slightly more flexibility by partitioning threads into context groups [48]. Threads within a context group share resources, and only one instruction can be issued from each group on every cycle.

4.6 Chapter summary

This chapter investigates the importance of dynamic resource sharing in future high-end processors. First, we identify the performance costs of resource partitioning for various multiprocessor configurations. By partitioning execution resources between processors, multiprocessors enforce the distinction between instruction-level and thread-level parallelism. In this study, we examined two MP design choices with similar hardware cost in terms of execution resources: one design with more resources per processor (MP2), and one with twice as many processors, but fewer resources on each (MP4). Our results showed that both alternatives frequently suffered from an inefficient use of their resources, and that improvements could only be obtained with costly upgrades in processor resources. The MP designs were unable to adapt to varying levels of ILP and TLP, and consequently, their performance depended heavily on the parallelism characteristics of the applications. For programs with more ILP, MP2 outperformed MP4; for programs with less ILP, MP4 was superior because it exploited more thread-level parallelism. To maximize performance on an MP, compilers and parallel programmers are therefore faced with the difficult task of partitioning program parallelism (ILP and TLP) in a manner that matches the physical partitioning of processor resources.

Second, we illustrate that, in contrast, simultaneous multithreading allows compilers and programmers to focus on extracting whatever parallelism exists, because it treats instruction-level and thread-level parallelism equally. ILP and TLP are fundamentally identical; they both represent independent instructions that can be used to increase processor utilization and improve performance. By dynamically sharing resources between threads, an SMT processor can use both forms of parallelism to exploit wide-issue hardware. This flexibility allows simultaneous multithreading to use existing resources more efficiently, improving average throughput to 4.88 instructions per cycle. Furthermore, by avoiding static resource partitioning, an SMT outperforms MP2 and MP4 by 64% and 52%, respectively.

Third, our results demonstrate that SMT can achieve large program speedups on parallel applications. The latency-hiding characteristics of simultaneous multithreading allow it to achieve a 2.68 average speedup over a single MP2 processor, while MP2 and MP4 speedups are limited to 1.63 and 1.76, respectively. The bottom line is that simultaneous multithreading makes better utilization of shared on-chip resources to run parallel applications effectively.

Chapter 5

Tuning compiler optimizations for SMT

In the previous chapter, we demonstrated that dynamic sharing can enable greater application performance by increasing resource utilization. When resources are shared between threads, however, these independent instruction streams may introduce interference or contention in shared structures. SMT software should be aware of this sharing in order to provide optimization policies that manage resources most effectively.

This chapter reexamines several compiler optimizations in this light. An SMT compiler needs to recognize that simultaneous multithreading presents a different model for hiding operation latencies and for sharing code and data. Operation latencies are hidden by instructions from all executing threads, not just by those in the thread with the long-latency operation. In addition, multi-thread instruction issue increases instruction-level parallelism (ILP) to levels much higher than can be sustained with a single thread. Both factors suggest reconsidering uniprocessor optimizations that hide latencies and expose ILP at the expense of increased dynamic instruction counts: on an SMT the latency-hiding benefits may not be needed, and the extra instructions may consume shared resources that could be better utilized by instructions in concurrent threads.

The compiler must also recognize that multiple threads can cheaply share common data and incur no penalty from false sharing, because they reside within a single SMT processor. In fact, threads benefit from cross-thread spatial locality. This calls into question compiler-driven parallelization techniques, originally developed for distributed-memory multiprocessors, that partition data to physically distributed threads to avoid communication and coherence costs. On an SMT, it may be beneficial to parallelize programs so that they process the same or contiguous data.

The goal of this chapter is to understand how simultaneous multithreading affects the use of several compiler optimizations that depend on these models for hiding operation latency and data/code sharing. In particular, we examine one parallel technique (loop-iteration scheduling for compiler-parallelized applications) and two optimizations that hide memory latencies and expose instruction-level parallelism (software speculative execution and loop tiling). Our results prescribe a different usage of all three optimizations when compiling for an SMT processor. By following the guidelines proposed in this chapter, compilers can generate code that manages shared resources in an SMT-conscious manner, thus improving the performance of an SMT processor.

The remainder of this chapter is organized as follows. Section 5.1 discusses in more detail two architectural assumptions that are affected by simultaneous multithreading and their ramifications on compiler-directed loop iteration scheduling, software speculative execution, and loop tiling. Section 5.2 presents our experimental methodology. Sections 5.3 through 5.5 examine each of the compiler optimizations, providing experimental results and analysis. Section 5.6 briefly discusses other compiler issues raised by SMT. Related work appears in Section 5.7, and we summarize in Section 5.8.

5.1 Rethinking compiler optimizations

In modern systems, the compiler and processor architecture are tightly intertwined. As a result, many back-end compiler optimizations are driven by specific assumptions about the underlying architecture and implementation of the target machine. For example, compilers schedule long-latency operations early to minimize critical paths, order instructions based on the processor's issue slot restrictions to maximize functional unit utilization, and allocate frequently used variables to registers to benefit from their fast access times. As another example, parallel programs are compiled to minimize inter-thread sharing when targeting shared-memory multiprocessors, in order to decrease high-cost, inter-processor communication. When new processing paradigms change these architectural assumptions, however, we must reevaluate machine-dependent compiler optimizations in order to maximize performance on the new machines.

Simultaneous multithreading allows threads to dynamically share processor resources much more efficiently than existing systems. This fine-granularity sharing prompts us to revisit several compiler optimizations to ensure that shared resources are managed in an SMT-conscious manner. In this section we discuss two factors affected by SMT's unique design, data sharing among threads and the availability of instruction issue slots, in light of three compiler optimizations they affect.

5.1.1 Inter-thread data sharing

Conventional parallelization techniques target multiprocessors, in which threads are physically distributed on different processors. To minimize cache coherence and inter-processor communication overhead, data and loop distribution techniques partition and distribute data to match the physical topology of the multiprocessor. Parallelizing compilers attempt to decompose applications to minimize synchronization and communication between loops. Typically, this is achieved by allocating a disjoint set of data for each processor, so that they can work independently [16, 20, 144].

In contrast, on an SMT, multiple threads execute on the same processor, affecting performance in two ways. First, both real and false inter-thread data sharing entail local memory accesses and incur no coherence overhead, because of SMT's shared L1 cache. Consequently, sharing, and even false sharing, is beneficial. Second, by sharing data among threads, the memory footprint of a parallel application can be reduced, resulting in better cache and TLB behavior. Both factors suggest a loop iteration scheduling policy that clusters, rather then separates, data for multiple threads.

5.1.2 Latency-hiding capabilities and the availability of instruction issue slots

On most workloads, wide-issue processors typically cannot sustain high instruction throughput, because of low instruction-level parallelism in their single, executing thread. Compiler optimizations, such as software speculative execution and loop tiling (or blocking), try to increase ILP (by hiding or reducing instruction latencies, respectively), but often with the side effect of increasing the dynamic instruction count. Despite the additional instructions, the optimizations are often profitable, because the instruction overhead can be accommodated in otherwise idle functional units.

Because it can issue instructions from multiple threads, an SMT processor has fewer empty issue slots; in fact, sustained instruction throughput can be rather high, roughly 2 times greater than on a conventional superscalar [31]. Furthermore, SMT does a better job of hiding latencies than single-threaded processors, because it uses instructions from one thread to mask delays in another. In this environment, some of the aforementioned optimizations may be less useful, or even detrimental, because the functional units are shared at a fine granularity. One thread's overhead instructions may compete for or even steal processor resources from another thread with more useful instructions. This suggests that we revisit the suitability of these optimizations for SMT processors.

5.2 Methodology

Before examining the compiler optimizations, we describe the methodology used in the experiments. We use applications from the SPEC 92 [27], SPEC 95 [121] and SPLASH-2 [146] benchmark suites (Table 5.1). The SPLASH-2 applications are explicitly parallel programs that are built on the ANL parallel macro library, while the SPECint95 applications are sequential. The SPEC92 and SPECfp95 applications are also sequential, but we were able to parallelize them, as described in the next section.

Table 5.1: Benchmarks. The last three columns identify the studies in which the applications are used. LD = loop distribution or loop iteration scheduling, SSE = software speculative execution, and T = tiling.

Application		Data set	Instructions simulated	LD	SSE	T
SPECfp95	applu	33x33x33 array, 2 iterations	272 M	X	X	
	hydro2d	2 iterations	474 M	X	X	
	mgrid	64x64x64 grid, 1 iteration	3.2 B	X	X	
	su2cor	16x16x16x16, vector len. 4K, 2 iterations	5.4 B	X	X	
	swim	512x512 grid, 10 iterations	419 M	X	X	
	tomcatv	513x513 array, 5 iterations	189 M	X	X	
SPLASH-2	fft	64K data points	32 M		X	
	LU	512x512 matrix	431 M		X	
	radix	256K keys, radix 1K, 512K max key value	6 M		X	
	water-nsquared	512 molecules, 3 timesteps	870 M		X	
	water-spatial	512 molecules, 3 timesteps	784 M		X	
SPECint95	compress	train input set	64 M		X	
	go	train input set, 2stone9	700 M		X	
	li	train input set	258 M		X	
	m88ksim	test input set, dhrystone	164 M		X	
	perl	train input set, scrabble	56 M		X	
SPEC92	mxm from NASA7	matrix multiply of 256x128 and 128x64 arrays	29 M			X
	gmt from NASA7	500x500 Gaussian elimination	354 M			X
	adi integration	1Kx1K stencil computation for solving partial differential equations	16 M			X

5.2.1 Compiler framework

For this study, we use the same basic compilation framework as described in Section 4.2. We then examine three specific compiler optimizations, as described below.

Loop iteration scheduling

The SUIF parallelizing compiler implements a blocked loop distribution policy commonly used for multiprocessor execution. For our experiments, we use applications compiled with the latest version of SUIF [11], but we do not have access to the source code. Therefore, we implement our alternative algorithm (described in Section 5.3) by hand.

Loop tiling

SUIF also performs loop tiling. The compiler finds tileable loops, determines appropriate multiprocessor-oriented tile sizes for particular data sets and caches, and then generates tiled code. For the tile size sensitivity experiments in Section 5.5, we experiment with other tile sizes with manual coding.

Software speculative execution

Software speculative execution is enabled/disabled by modifying the Multiflow compiler's machine description file, which specifies which instructions can be moved speculatively by the trace scheduler. We experiment with both statically-generated and profile-driven traces; for the latter, profiling information is generated by instrumenting the applications and then executing them with a training input data set that differs from the set used during simulation.

5.2.2 Simulation framework

The compiler optimization experiments in this chapter rely on the simulation infrastructure described in Section 4.2.3, but with slightly more aggressive branch prediction and memory system characteristics. We are targeting the microprocessor design space a few years into the future, and have extrapolated technology trends accordingly. The details of our SMT processor core are listed in Table 5.2.

For branch prediction, we use a 256-entry, 4-way set-associative branch target buffer and a McFarling-style hybrid predictor with an 8K entry selector that chooses between a global history predictor (13 history bits) and a local predictor (a 2K-entry local history table that indexes into a 4K-entry, 2-bit local prediction table) [84].

The details of the memory hierarchy are shown in Table 5.3. We also model separate

Table 5.2: CPU parameters used in our simulator.

Active list	128 entries/context
Instruction queue	32 integer entries, 32 FP entries
Functional units	6 integer (including 4 ld/st units), 4 FP
Architectural registers	32*8 integer / 32*8 FP
Renaming registers	100 integer / 100 FP
Instruction retirement	up to 12 instructions per cycle

Table 5.3: Memory hierarchy parameters. When there is a choice of values, the first (the more aggressive) represents a forecast for an SMT implementation roughly three years in the future and is used in all experiments. The second set is more typical of today's memory subsystems and is used to emulate larger data set sizes [110]; it is used in the tiling studies only.

	L1 I-cache	**L1 D-cache**	**L2 cache**
Cache size (bytes)	128K / 32K	128K / 32K	16 M / 2 M
Line size (bytes)	64	64	64
Associativity	2-way	2-way	direct-mapped
Banks	8	4	1
Ports/bank	1	2	1
Transfer time/bank	1 cycle	1 cycle	4 cycles
Accesses/cycle	2	4	1/4
Cache fill time (cycles)	2	2	8
Latency to next level	10 / 18	10 / 18	68
Maximum number of in-flight misses	16	16	16

128-entry data and instruction TLBs. In addition, for the loop distribution studies, we experiment with 48-entry TLBs to illustrate how the performance of loop distribution policies is sensitive to TLB size. The larger TLB represents a probable configuration for a (future) general-purpose SMT; the smaller is more appropriate for a less aggressive design, such as an SMT multimedia co-processor, where page sizes are typically in the range of 2-8MB. For both TLB sizes, misses require two full memory accesses, incurring a 160 cycle penalty.

Because of the length of the simulations, we limit our detailed simulation results to the parallel computation portion of the applications. For the initialization phases of the applications, we use a fast simulation mode that only simulates the caches, so that they were warm when the main computation phases were reached. We then turn on the detailed simulation mode.

5.3 Loop iteration scheduling

To reduce communication and coherence overhead in distributed-memory multiprocessors, parallelizing compilers employ a blocked loop parallelization policy to distribute iterations across processors. A blocked distribution assigns each thread (processor) continuous array data and iterations that manipulate them (Figure 5.1). Figure 5.2 presents SMT speedups for applications parallelized using a blocked distribution with two TLB sizes. Good speedups are obtained for many applications (as the number of threads is increased), but in the smaller TLB the performance of several programs (hydro2d, swim, and tomcatv) degrades with 6 or 8 threads. The 8-thread case is particularly important, because most applications will be parallelized to exploit all 8 hardware contexts in an SMT. Analysis of the simulation bottleneck metrics indicated that the slowdown was the result of thrashing in the data TLB, as indicated by the TLB miss rates of Table 5.4.

The TLB thrashing is a direct result of blocked partitioning, which increases the total

Table 5.4: TLB miss rates for a blocked distribution and a 48-entry data TLB. The bold entries correspond to decreased performance (see Figure 5.2) when the number of threads was increased.

Application	Number of threads				
	1	**2**	**4**	**6**	**8**
applu	0.7%	0.9%	1.0%	0.9%	1.0%
hydro2d	0.1%	0.1%	0.1%	0.7%	**6.3%**
mgrid	0.0%	0.0%	0.0%	0.0%	0.1%
su2cor	0.1%	5.2%	7.7%	6.2%	5.5%
swim	0.2%	0.2%	0.2%	**27.6%**	**49.4%**
tomcatv	0.1%	0.1%	0.1%	2.0%	**10.7%**

a) original loop

```
for (j = 1; j < n; j++)
 for (i = 0; i < m; i++)
  u1[j][i] = u2[j][i] + t1/8.0 * Z[j][i]
```

b) blocked parallelization

```
for (j = max(n * myid / numthreads + 1, 1);
     j < min(n * (myid+1) / numthreads + 1, n);
     j++)
 for (i = 0; i < m; i++)
  u1[j][i] = u2[j][i] + t1/8.0 * Z[j][i]
```

c) cyclic parallelization

```
for (j = myid + 1; j < n; j += numthreads)
 for (i = 0; i < m; i++)
  u1[j][i] = u2[j][i] + t1/8.0 * Z[j][i]
```

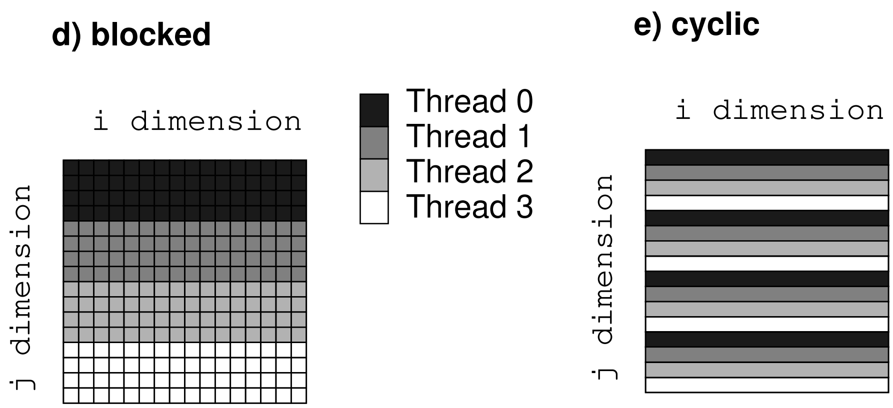


Figure 5.1: A blocked and cyclic loop distribution example.

The code for an example loop nest is shown in a). When using a blocked distribution, the code is structured as in b). The cyclic version is shown in c). d) and e) illustrate which portions of the array are accessed by each thread for the two policies. (For clarity, we assume 4 threads). Assume that each row of the array is 2KB (512 double precision elements). With blocked distribution (d), each thread accesses a different 8KB page in memory. With cyclic (e), however, the loop is decomposed in a manner that allows all four threads to access a single 8KB page at the same time, thus reducing the TLB footprint.

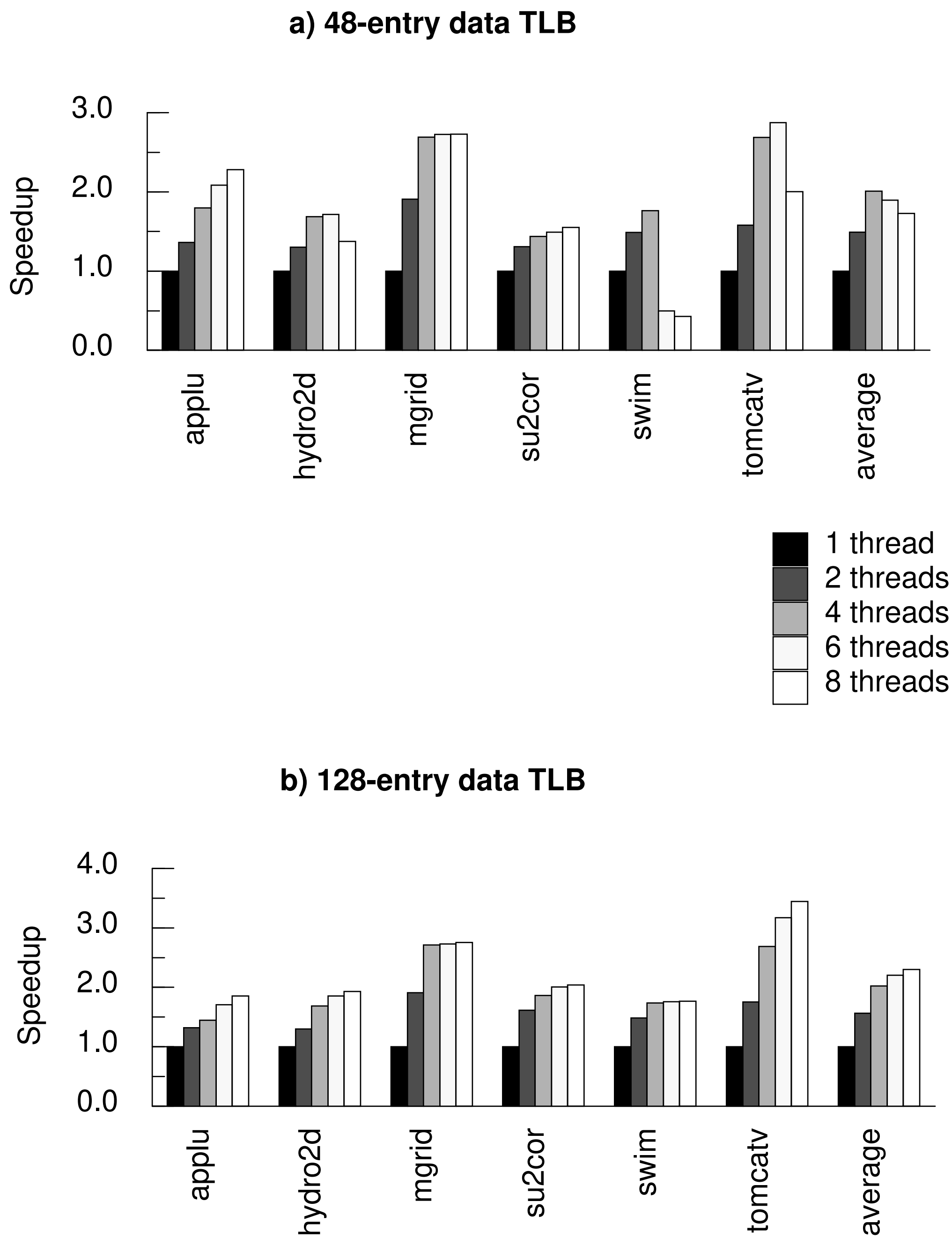


Figure 5.2: Speedups over one thread for blocked parallelization.

working set of an application because threads work on disjoint data sets. In the most severe cases, each of the 8 threads requires many TLB entries, because loops stride through several large arrays at once. Since the primary data sets are usually larger than a typical 8KB page size, at least one TLB entry is required for each array.

The swim benchmark from SPECfp95 illustrates an extreme example. In the main loop of the `calc3` routine, 9 large arrays are accessed on each iteration of the loop. When the loop is parallelized using a blocked distribution, the data TLB footprint is 9 arrays * 8 threads = 72 TLB entries, excluding the entries required for other data. With any TLB size less than 72, significant interference will occur and the parallelization is not profitable.

The lesson here is that the TLB is a shared resource that needs to be managed efficiently in an SMT. At least three approaches can be considered: (1) using fewer than 8 threads when parallelizing, (2) increasing the data TLB size, or (3) parallelizing loops differently.

The first alternative unnecessarily limits the use of the thread hardware contexts, and neither exploits SMT nor the parallel applications to their fullest potential. The second choice incurs a cost in access time and hardware, although with increasing chip densities, future processors may be able to accommodate.[1] Even with larger TLBs, however, it is desirable to reduce the TLB footprint on an SMT. A true SMT workload would be multiprogrammed: for example, multiple parallel applications could execute together, comprising more threads than hardware contexts. The thread scheduler could schedule all 8 threads for the first parallel application, then context switch to run the second, and later switch back to the first. In this type of environment it would be performance-wise to minimize the data TLB footprint required by each application. (As an example, the TLB footprint of a multiprogrammed workload consisting of swim and hydro2d would be greater than 128 entries.)

The third and most desirable solution relies on the compiler to reduce the data TLB footprint. Rather than distributing loop iterations in a blocked organization, it could use a cyclic distribution to cluster the accesses of multiple threads onto fewer pages. (With cyclic partitioning, swim would consume 9 rather than 72 TLB entries). Cyclic partitioning also requires less instruction overhead in calculating array partition bounds, a non-negligible, although much less important factor. (Compare the blocked and cyclic loop distribution code and data in Figure 5.1.)

---

[1]. We found that 64 entries did not solve the problem. However, a 128-entry data TLB avoids TLB thrashing, and as Figure 5.2b indicates, achieves speedups, at least for the SPECfp95 data sets.

Figure 5.3 illustrates the speedups attained by a cyclic distribution over blocked, and Table 5.5 contains the corresponding changes in data TLB miss rates. With the 128-entry data TLB, absolute miss rates are low enough (usually under 0.2%, except for applu and su2cor, which reached 0.9%) that most changes produced little or no overall performance benefit for cyclic. The two exceptions were mgrid and su2cor. Mgrid saw a large performance improvement, because of a reduction in dynamic instruction count. As Figures 5.1b and c illustrate, cyclic parallelization requires fewer computations and no long-latency divide. In contrast, su2cor saw degradation, because cyclic scheduling increased loop unrolling instruction overhead.

Table 5.5: Improvement (decrease) in TLB miss rates of cyclic distribution over blocked.

	48-entry TLB					**128-entry TLB**				
	Number of threads					**Number of threads**				
Application	**1**	**2**	**4**	**6**	**8**	**1**	**2**	**4**	**6**	**8**
applu	0%	50%	58%	53%	15%	0%	91%	98%	85%	69%
hydro2d	0%	0%	14%	91%	99%	0%	0%	0%	0%	14%
mgrid	0%	0%	0%	0%	50%	0%	0%	0%	0%	0%
su2cor	14%	99%	99%	99%	97%	0%	0%	98%	91%	94%
swim	0%	0%	0%	99%	99%	0%	0%	0%	0%	0%
tomcatv	0%	-60%	-60%	96%	99%	0%	-60%	-60%	-60%	-60%

When a small TLB footprint is more critical, i.e., with the 48-entry TLB, all applications did better with a cyclic distribution. In most cases the significant decrease in data TLB misses, coupled with the long 160 cycle TLB miss penalty, was the major factor. Cyclic increased TLB conflicts in tomcatv at 2 and 4 threads, but, because the number of misses was so low, overall program performance did not suffer. At 6 and 8 threads, tomcatv's blocked data TLB miss rate jumped to 2% and 11%, causing a corresponding hike in speedup for cyclic. Su2cor also saw a performance improvement, because the improved TLB hit rate offset the degradations due to increased instruction overhead.

In summary, these results suggest using a cyclic loop distribution for SMT, rather than the traditional blocked distribution. For parallel applications with large data footprints, cyclic distribution increased program speedups. (We saw speedups as high as 4.1, even with the smallish SPECfp95 reference data sets.) For applications with smaller

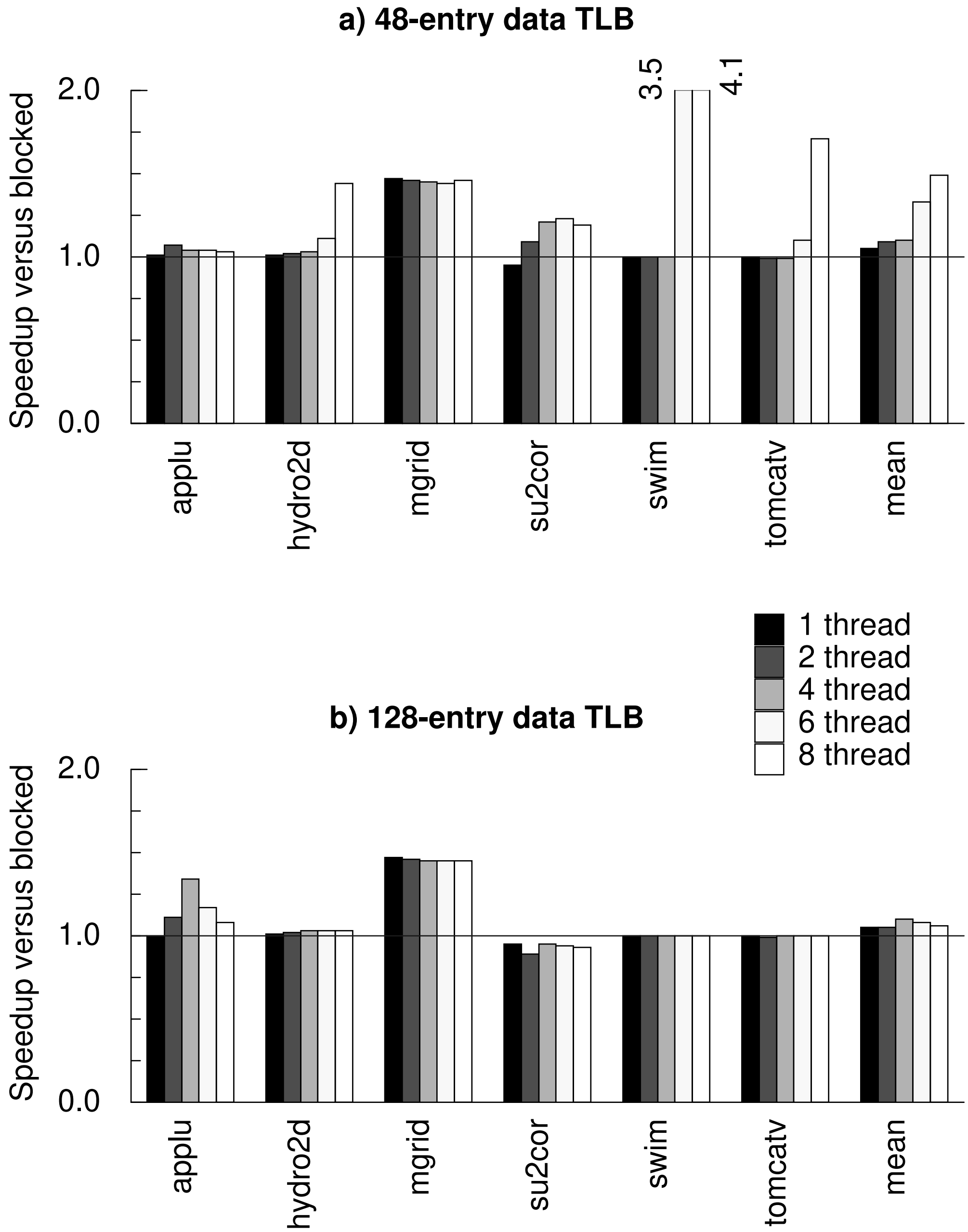


Figure 5.3: Speedup attained by cyclic over blocked parallelization.

For each application, the execution time for blocked is normalized to 1.0 for all numbers of threads. Thus, each bar compares the speedup for cyclic over blocked with the same number of threads.

data footprints, cyclic broke even. Only in one application, where there was an odd interaction with the loop unrolling factor, did cyclic worsen performance.

As we move to larger SMT systems, for example a multiprocessor of SMT CPUs, a cyclic distribution would still be appropriate. A hybrid parallelization policy might be desirable, though, with a cyclic distribution within each CPU, but with a blocked distribution across processors to minimize inter-processor communication.

5.4 Software speculative execution

Today's optimizing compilers rely on aggressive code scheduling to hide instruction latencies. In global scheduling techniques, such as trace scheduling [79] or hyperblock scheduling [81], instructions from a predicted branch path may be moved above a conditional branch, so that their execution becomes speculative. If at runtime, the other branch path is taken, then the speculative instructions are useless and potentially waste processor resources.

On in-order superscalars or VLIW machines, software speculation is necessary, because the hardware provides no scheduling assistance. On an SMT processor (whose execution core is an out-of-order superscalar), not only are instructions dynamically scheduled and speculatively executed by the hardware, but multithreading is also used to hide latencies. Furthermore, as the number of SMT threads is increased, instruction throughput also increases because the functional units are shared among threads. Therefore, the latency-hiding benefits of software speculative execution may be needed less, or even be unnecessary, and the additional instruction overhead introduced by incorrect speculations may degrade performance by increasing contention for functional units.

Our experiments were designed to evaluate the appropriateness of software speculative execution for an SMT processor. The results highlight two factors that determine its effectiveness for SMT: static branch prediction accuracy and instruction throughput.

Correctly-speculated instructions have no instruction overhead; incorrectly-speculated instructions, however, add to the dynamic instruction count. Therefore, speculative execution is more beneficial for applications that have high speculation accuracy, e.g., loop-based programs with either profile-driven or state-of-the-art static branch prediction.

Table 5.6 compares the dynamic instruction counts between speculative and non-speculative versions of our applications. Small increases in the dynamic instruction

Table 5.6: Percentage increase in dynamic instruction count due to profile-driven software speculative execution. Data are shown for eight threads. (One thread numbers were identical or very close). Applications in bold have high speculative instruction overhead and high IPC without speculation; those in italics have only the former.

SPECfp95	% increase	SPECint95	% increase	SPLASH-2	% increase
applu	2.1%	compress	2.9%	**fft**	**13.7%**
hydro2d	1.9%	*go*	*12.6%*	**LU**	**12.5%**
mgrid	0.5%	**li**	**7.3%**	radix	0.0%
su2cor	0.1%	*m88ksim*	*4.0%*	water-nsquared	3.0%
swim	0.1%	*perl*	*4.9%*	water-spatial	3.0%
tomcatv	1.2%			applu	2.1%

count indicate that the compiler has been able to accurately predict which paths will be executed.[2] Consequently, speculation may incur no penalties. Higher increases in dynamic instruction count, on the other hand, mean wrong-path speculations, and a probable loss in SMT performance.

In this study, we used profile-driven speculation to assist the compiler's static branch prediction, thus providing a best-case comparison versus no speculation. Without profiles, more mispredictions would have occurred and more overhead instructions would have been generated. Consequently, software speculation would have had worse performance than we report, making its absence appear even more beneficial for SMT.

While static branch prediction (and therefore instruction overhead) influence the effectiveness of speculation, it is not the only factor. The level of instruction throughput in programs without speculation is also important, because it determines how easily speculative overhead can be absorbed. With sufficient instruction issue bandwidth (low IPC), incorrect speculations may cause no harm; with higher per-thread ILP or more threads, software speculation should be less profitable, because incorrectly-speculated instructions are more likely to compete with useful instructions for processor resources (in particular, fetch bandwidth and functional unit issue). Table 5.7 contains the instruction throughput for each of the applications. For some programs IPC is higher with software speculation, indicating some degree of absorption of the speculation overhead. In others, it is lower, because of additional hardware resource conflicts, most

[2] All the SPECfp95 programs, radix from SPLASH-2, and compress from SPECint95, are loop-based; all have small increases in dynamic instruction count with speculation.

Table 5.7: Throughput (instructions per cycle) with and without profile-driven software speculation for 8 threads. Programs in **bold** have high IPC without speculation, plus high speculation overhead; those in *italics* have only the former.

	Benchmark	**speculation**	**no speculation**
SPECint95	compress	4.1	4.0
	go	2.4	2.3
	li	4.5	**4.6**
	m88ksim	4.2	4.1
	perl	3.8	3.7
SPECfp95	*applu*	4.9	*4.7*
	hydro2d	5.6	*5.4*
	mgrid	7.2	*7.1*
	su2cor	6.1	*6.0*
	swim	6.5	*6.2*
	tomcatv	6.2	*5.9*
SPLASH-2	**fft**	6.0	**6.4**
	LU	6.7	**6.8**
	radix	5.4	*5.4*
	water-nsquared	6.4	*6.1*
	water-spatial	6.4	*6.5*

notably L1 cache misses. Note that the number of threads has an indirect effect on the benefits of speculation, because additional threads provide more parallelism. When many threads are running, speculative instructions are more likely to compete with useful instructions for processor resources; the advantage of turning off speculation generally becomes even larger.

Speculative instruction overhead (related to static branch prediction accuracy) and instruction throughput together explain the speedups (or lack thereof) illustrated in Figure 5.4. When both factors were high (the non-loop-based fft, li, and LU), speedups without software speculation were greatest, ranging up to 22%. If one factor was low or only moderate, speedups were minimal or nonexistent (the SPECfp95 applications, radix and water-nsquared had only high IPC; go, m88ksim and perl had only speculation

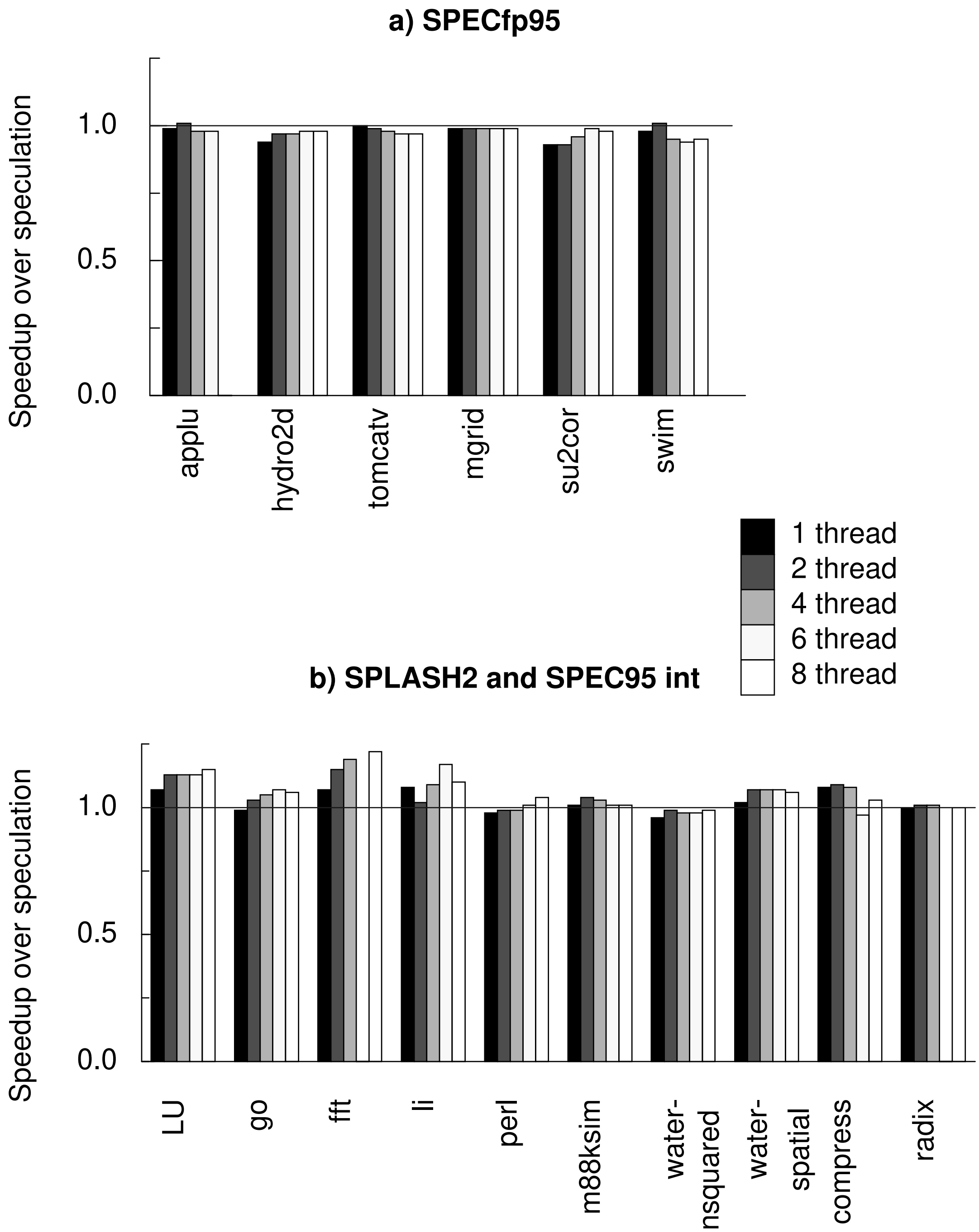


Figure 5.4: Speedups of applications executing without software speculation over with speculation (speculative execution cycles / no speculation execution cycles).

Bars that are greater than 1.0 indicate that no speculation is better.

overhead).[3] Without either factor, software speculation helped performance, and for the same reasons it benefits other architectures -- it hid latencies and executed the speculative instructions in otherwise idle functional units.

In addition to these two factors, the number of threads can also have a small effect. As more threads are used, the advantage of turning off speculation generally becomes even larger. Additional threads provide more parallelism, and therefore, speculative instructions are more likely to compete with useful instructions for processor resources.

Keep in mind that we used run-time support (profiling) to improve the compiler's static branch prediction accuracy. Had we speculated without run-time support (profiling), the relative benefit of no speculation (versus speculation) would have been higher. For example, at 8 threads, water-nsquared breaks even with profile-driven speculation. However, relying on Multiflow's static branch prediction gives no speculation a slight edge, with a speedup of 1.1. Nevertheless, the general conclusions still hold: both good branch prediction and low multi-thread IPC are needed for software speculation to benefit applications executing on an SMT.

The bottom line is that, while loop-based applications should be compiled with software speculative execution, non-loop applications should be compiled without it. Doing so either improves SMT program performance or maintains its current level -- performance is never hurt.

5.5 Loop tiling

In order to improve cache behavior, loops can be tiled to take advantage of data reuse. In this section, we examine two tiling issues: tile size selection and the distribution of tiles to threads.

If the tile size is chosen appropriately, the reduction in average memory access time more than compensates for the tiling overhead instructions [15, 21, 71]. (The code in Figures 5.5b and c illustrates the source of this overhead). On an SMT, however, tiling may be less beneficial. First, SMT's enhanced latency-hiding capabilities may render tiling unnecessary. Second, the additional tiling instructions may increase execution time, given SMT's higher (multithreaded) throughput and shared functional units. (These are the same factors that influence whether to software speculate.)

To address these issues, we examined tileable loop nests with different memory

---

[3]. Even though it has few floating point computations, water-spatial had a high IPC without speculation (6.5). Therefore the speculative instructions bottlenecked the integer units, and execution without speculation was more profitable.

a) original loop

```
for (j = 1; j < N; j++
   for (k = 1; k < L; k++)
      for (i = 1; i < M; i++)
         c[j][k] += a[i][k]*b[j][i];
```

b) blocked tiling

```
ub = min(N*(myid+1)/numthreads+1, N);
lb = max(N*myid/numthreads+1, 1);
for (jT=lb; jT <= ub; jT+=jTsize)
   for (kT=1; kT <= L; kT+=kTsize)
      for (iT=1; iT < M; iT+=iTsize)
         for (j=jT; j <= min(N,jT+jTsize-1); j++)
            for (k= max(1, kT);
                   k <= min(L,kT+kTsize-1); k++)
               for (i=iT; i <= min(M,iT+iTsize-1); i++)
                  c[j][k] += a[i][k]*b[j][i];
```

c) cyclic tiling

```
ub = min(N * (myid + 1)/numthreads + 1, N);
lb = max(N * myid / numthreads + 1, 1);
for (jT = lb; jT <= ub; jT += jTsize)
   for (kT = 1; kT <= L; kT += kTsize)
      for (iT = 1; iT < M; iT += iTsize)
         for (j = jT+myid; j <= min(N,jT+jTsize-1);
                j += numthreads)
            for (k= max(1, kT);
                   k <= min(L,kT+kTsize-1); k++)
               for (i=iT; i <= min(M,iT + iTsize-1); i++)
                  c[j][k] += a[i][k]*b[j][i];
```

Figure 5.5: Code for blocked and cyclic versions of a tiled loop nest.

The boldface code highlights additional instruction overhead required for tiling.

access characteristics, executing on an SMT processor. The benefits of tiling vary when the size of the cache is changed. Smaller caches require smaller tiles, which naturally introduce more instruction overhead. On the other hand, smaller tiles also produce lower average memory latencies -- i.e., fewer conflict misses -- so the latency reducing benefit of tiling is better. We therefore varied tile sizes to measure the performance impact of a range of tiling overhead. We also simulated two memory hierarchies to gauge the interaction between cache size, memory latency and tile size. The larger memory configuration represents a probable SMT memory subsystem for machines in production approximately 3 years in the future. The other configuration is smaller, modeling today's memory hierarchies, and is designed to provide a more appropriate ratio between data set and cache size, modeling loops with larger, i.e., more realistic, data sets than those in our benchmarks. For these experiments, each thread was given a separate tile (the tiling norm).

Figure 5.6 presents the performance (total execution cycles, average memory access time, and dynamic instruction count for a range of tile sizes and the larger memory configuration) of an 8-thread SMT execution of each application and compares it to a single-thread run (approximating execution on a superscalar [31]). The results indicate that tiling is profitable on an SMT, just as it is on conventional processors. Mxm may seem to be an exception, since tiling brings no improvement, but it is an exception that shows there is no harm in applying the optimization. Programs executing on an SMT appear to be insensitive to tile size; at almost all tile sizes examined, SMT was able to hide memory latencies (as indicated by the flat AMAT curves), while still absorbing tiling overhead. Therefore SMT is less dependent on static algorithms to determine optimal tile sizes for particular caches and working sets. In contrast, conventional processors are more likely to have a tile size sweet spot. Even with out-of-order execution, these processors, as well as single-die multiprocessor architectures, lack sufficient latency-hiding abilities; consequently, they require more exact tile size calculations from the compiler.

Tile size is also not a performance determinant with the less aggressive memory subsystem, indicating that tiling on SMT is robust across memory hierarchies (or, alternatively, a range of data set sizes). Execution time is, of course, higher, because performance is more dependent on AMAT parameters, rather than tiling overhead. Only adi became slightly less tolerant of tile size changes. At the largest tile size measured (32x32), its AMAT increased sharply, because of inter-thread interference in the small cache. For this loop nest, either tiles should be sized so that all fit in the cache, or an alternative tiling technique (described below) should be used.

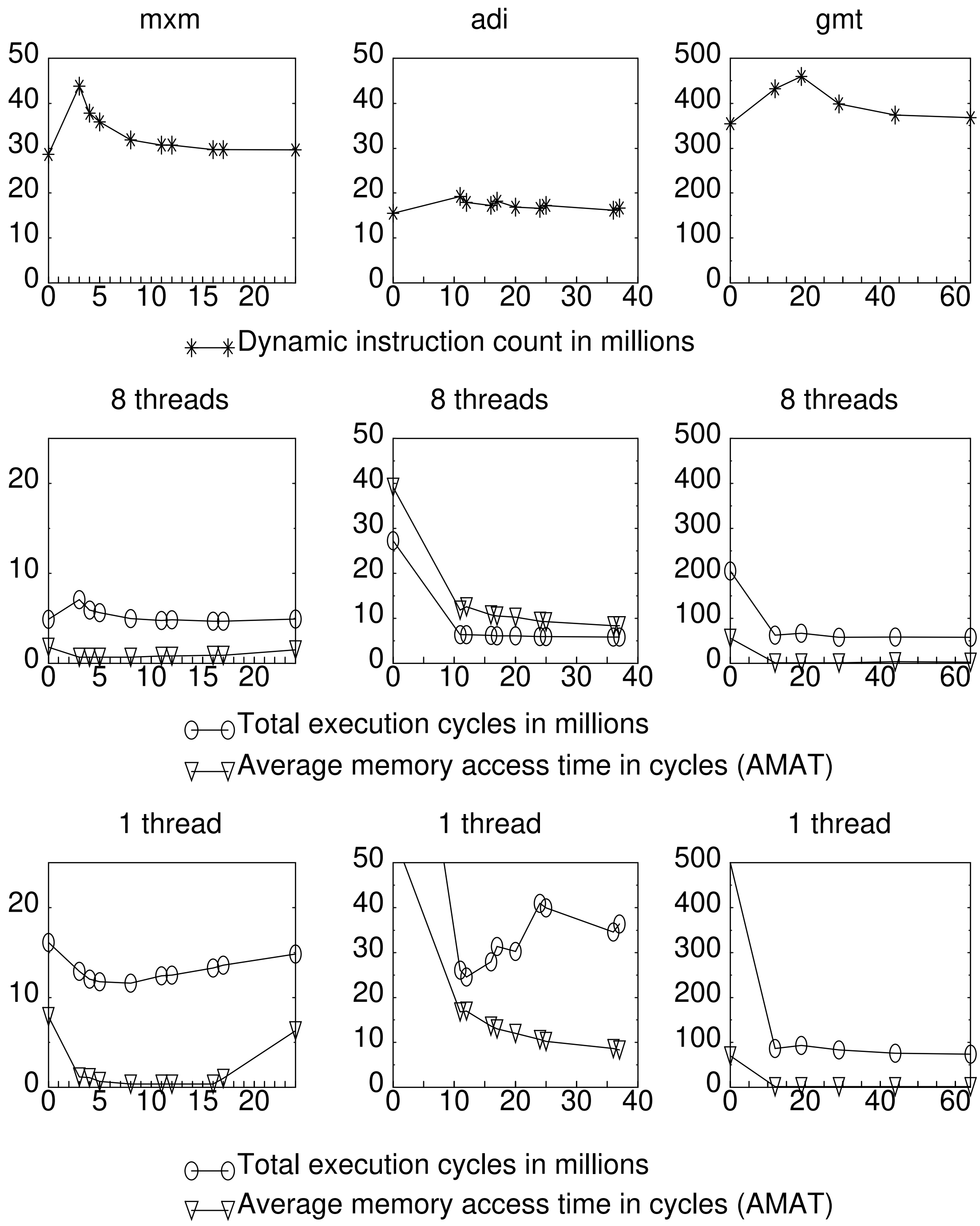


Figure 5.6: Tiling results with the larger memory subsystem and separate tiles/thread.

All the horizontal axes are tile size. A tile size of 0 means no tiling; sizes greater than 0 are one dimension of the tile, measured in array elements. The vertical axes are metrics for evaluating tiling: dynamic instruction count, total execution cycles and AMAT.

The second loop tiling issue is the distribution of tiles to threads. When parallelizing loops for multiprocessors, a different tile is allocated to each processor (thread) to maximize reuse and reduce inter-processor communication. On an SMT, however, tiling in this manner could be detrimental. Private, per-thread tiles discourage inter-thread tile sharing and increase the total-thread tile footprint on the single-processor SMT (the same factors that make blocked loop iteration scheduling inappropriate for SMT).

Rather than giving each thread its own tile (called blocked tiling), a single tile can be shared by all threads, and loop iterations can be distributed cyclically across the threads (cyclic tiling). (See Figure 5.5 for a code illustration of blocked and cyclic tiling, and Figure 5.7 for the effect on the per-thread data layout).

Because the tile is shared, cyclic tiling can be optimized by increasing the tile size to reduce tiling overhead (Figure 5.7c). With larger tiles, cyclic tiling can drop application execution times on SMT processors with smaller caches closer to that of SMTs with more aggressive memory hierarchies. (Or, put another way, the performance of programs with large data sets can approach those with smaller.) For example, Figure 5.8e illustrates that with larger tile sizes (greater than 8 array elements per dimension) cyclic tiling reduced mxm's AMAT enough to decrease average execution time on the smaller cache hierarchy by 51% (compare to blocked in Figure 5.8d) and to within 35% of blocked tiling on a memory subsystem several times the size (Figure 5.8c). Only at the very smallest tile size did an increase in tiling overhead overwhelm SMT's ability to hide memory latency.

Cyclic tiling is still appropriate for a multiprocessor of SMTs, where a hierarchical [17] or hybrid tiling approach might be most effective. Cyclic tiling could be used to maximize locality in each processor, with blocked tiling distributing tiles across processors to minimize inter-processor communication.

5.6 Other compiler optimizations

In addition to the optimizations studied here, compiler-directed prefetching, predicated execution and software pipelining should also be re-evaluated in the context of an SMT processor.

On a conventional processor, compiler-directed prefetching [13, 85] can be useful for tolerating memory latencies, as long as prefetch overhead (due to prefetch instructions, additional memory bandwidth, and/or cache interference) is minimal. On an SMT, this overhead is more detrimental: it interferes not only with the thread doing the prefetching, but also competes with other threads.

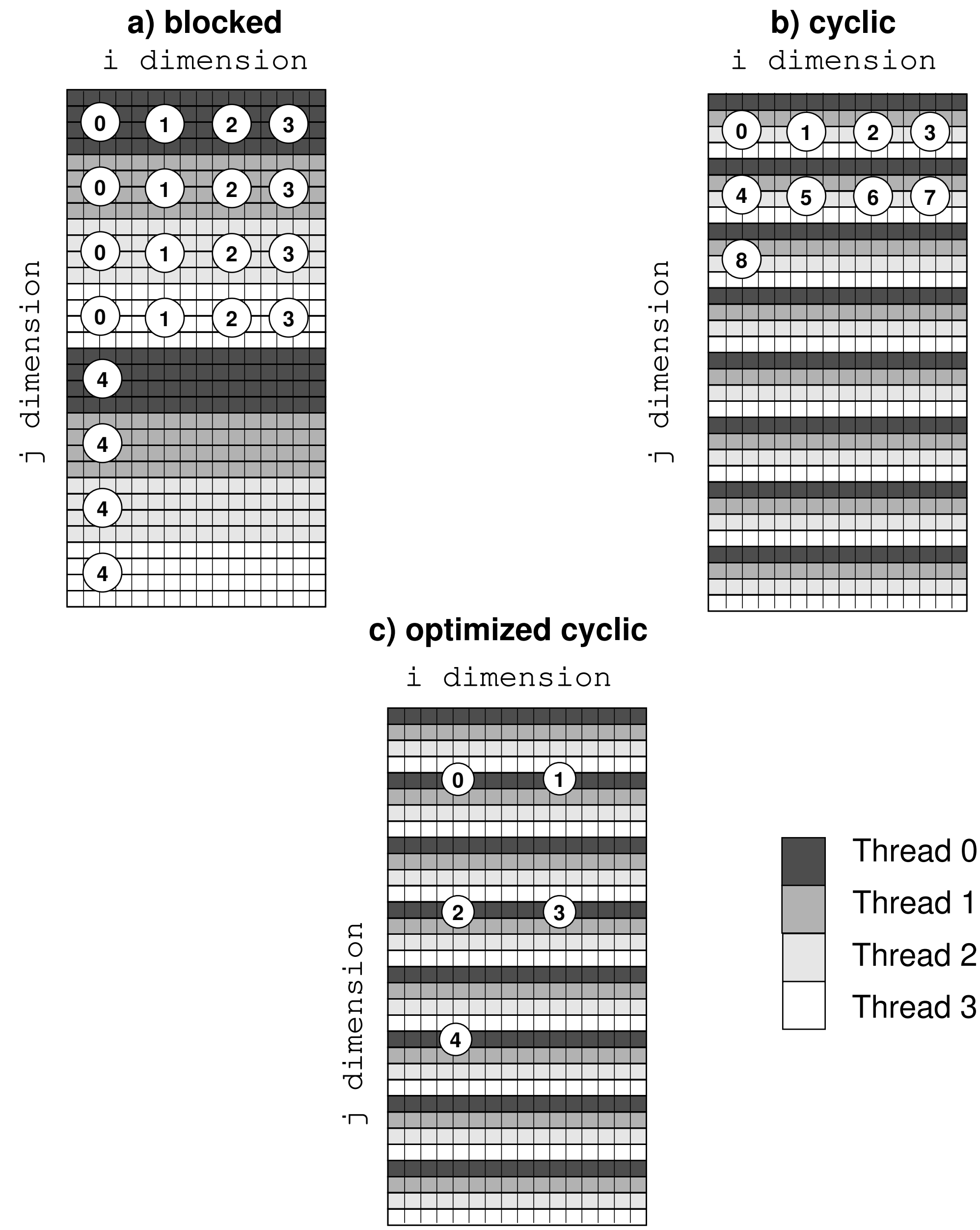


Figure 5.7: A comparison of blocked and cyclic tiling techniques for multiple threads.

The blocked tiling is shown in a). Each tile is a 4x4 array of elements. The numbers represent the order in which tiles are accessed by each thread. For cyclic tiling, each tile is still a 4x4 array, but now the tile is shared by all threads. In this example, each thread gets one row of the tile, as shown in b). With cyclic tiling, each thread works on a smaller chunk of data at a time, so the tiling overhead is greater. In c), the tile size is increased to 8x8 to reduce the overhead. Within each tile, each thread is responsible for 16 of the elements, as in the original blocked example.

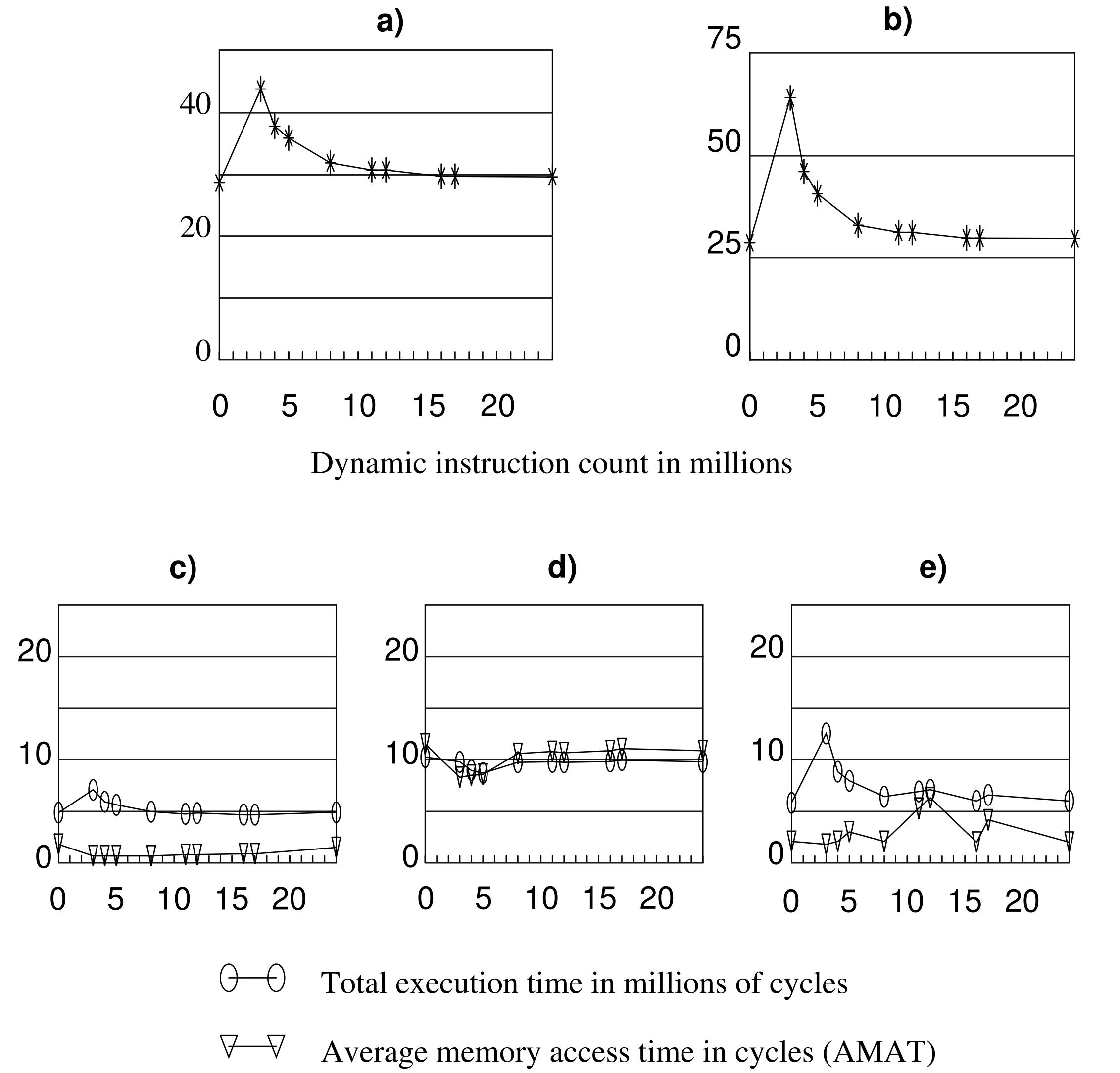


Figure 5.8: Tiling performance of 8-thread mxm.

Tile sizes are along the x-axis. Dynamic instruction counts are shown for a) blocked tiling and b) cyclic tiling. Total execution time results are shown for c) blocked tiling and the larger memory subsystem, d) blocked tiling with the smaller memory subsystem, and e) cyclic tiling, also with the smaller memory subsystem.

Predicated execution [56, 81, 103] is an architectural model in which instruction execution can be guarded by boolean predicates that determine whether an instruction should be executed or nullified. Compilers can then use if-conversion [4] to transform control dependences into data dependences, thereby exposing more ILP. Like software speculative execution, aggressive predication can incur additional instruction overhead by executing instructions that are either nullified or produce results that are never used.

Software pipelining [3, 18, 69, 102] improves instruction scheduling by overlapping the execution of multiple loop iterations. Rather than pipelining loops, SMT can execute them in parallel in separate hardware contexts. Doing so alleviates the increased register pressure normally associated with software pipelining. Multithreading could also be combined with software pipelining if necessary.

A final issue related to an SMT compiler is the question of compiling for throughput or for a single executing thread. Most of the optimizations discussed in this chapter were originally designed to increase single-thread ILP. While intra-thread parallelism is still important on an SMT processor, simultaneous multithreading relies on multiple threads to provide useful parallelism, and throughput often becomes a more important performance metric. For example, from the perspective of a single running thread, these optimizations, as traditionally applied, may be desirable to reduce the thread's running time. But from a global perspective, greater throughput (and therefore more useful work) can be achieved by limiting the amount of speculative work. In this chapter, our goal has been to achieve greater machine throughput. Research in techniques for balancing throughput and priority-thread performance is an interesting area of future work.

5.7 Compiler related work

The three compiler optimizations discussed in this paper have been widely investigated in non-SMT architectures.

5.7.1 Loop iteration scheduling

Loop iteration scheduling for shared-memory multiprocessors has been evaluated by Wolf and Lam [144], Carr et al. [16], Anderson et al. [6], and Cierniak and Li [20], among others. These studies focus on scheduling to minimize communication and synchronization overhead. In each study, memory access locality was improved by restructuring loops and data layout for each processor. In particular, Anderson et al. discuss the blocked and cyclic mapping schemes, and present a heuristic for choosing between them. None of these studies, though, investigate the implications of loop iteration scheduling for multithreaded processors.

5.7.2 Code scheduling optimizations

Global scheduling optimizations, like trace scheduling [79], superblocks [57] and hyperblocks [81], allow code motion (including speculative motion) across basic blocks, thereby exposing more ILP for statically-scheduled VLIWs and wide-issue superscalars. In their study on ILP limits, Lam and Wilson [70] found that speculation provides greater speedups on loop-based numeric applications than on non-numeric codes, but their study did not include the effects of wrong-path instructions. Our experiments found similar relative behavior in the context of an SMT processor.

Much of the work in global code scheduling algorithms has relied on priorities and heuristics to guide speculation and aggressive code motion. The results of our study could be integrated into these algorithms to tune them for SMT processors. For example, we found that the cost of speculative execution was highly dependent on the static branch prediction or branch bias. If the compiler could determine (statically or with profile information) that a particular branch was almost always taken, it would still be profitable to speculatively execute that instruction, even on an SMT.

5.7.3 Optimizations for improving locality

Previous work in code transformation for improved locality has proposed various frameworks and algorithms for selecting and applying a range of loop transformations [15, 16, 41, 65, 143, 144]. These studies illustrate the effectiveness of tiling and also propose other loop transformations for enabling better tiling. Lam et al. [71], Coleman and McKinley [21], and Carr et al. [15] show that application performance is sensitive to the tile size, and present techniques for selecting tile sizes based on problem-size and cache parameters, rather than targeting a fixed-size or fixed-cache occupancy. These studies focus on in-order, single-threaded processors, which lack the latency tolerance of an SMT processor.

5.8 Chapter summary

On an SMT processor, improper optimization policies can lead to interference or contention in shared structures like the TLB or functional units. The compiler can avoid these inefficiencies, however, by recognizing how simultaneous multithreading differs from previous parallel architectures. In this chapter we have identified three characteristics that an optimizing SMT compiler should take into consideration. First, SMT threads share memory system resources of a single processor on a fine-grained basis, even within a single cycle. Optimizations for an SMT should therefore seek to benefit from this fine-grained sharing, rather than avoiding it, as is done on conventional shared-memory multiprocessors. Second, inter-thread sharing of functional units contributes to much higher utilization and instruction throughput. Optimizations that expose ILP may not be needed, because SMT hides intra-thread latencies by using instructions from other active threads. Third, inter-thread sharing also increases the likelihood of contention for functional units. Threads must compete for available function units; optimizations that increase the dynamic instruction count may therefore degrade performance.

An effective compilation strategy for simultaneous multithreaded processors must recognize these unique characteristics in order to properly manage shared resources. Our results show specific cases where an SMT processor can benefit from changing the compiler optimization strategy. In particular, we showed that (1) cyclic iteration scheduling (as opposed to blocked scheduling) is more appropriate for an SMT, because of its ability to reduce the TLB footprint; (2) software speculative execution can be bad for an SMT, because it decreases useful instruction throughput; (3) loop tiling algorithms can be less concerned with determining the exact tile size, because SMT performance is less sensitive to tile size; and (4) loop tiling to increase, rather than reduce, inter-thread tile sharing, is more appropriate for an SMT, because it increases the benefit of sharing memory system resources.

Chapter 6

Managing cache interference in database workloads

In Chapter 5, we demonstrated the importance of managing shared resources in a simultaneous multithreaded processor through the use of an SMT-aware compiler. In this chapter, we continue our investigation of shared resources by focusing on the effects of thread-level parallelism on the performance of a shared cache. In particular, we look at cache interference and other memory system issues for memory-intensive commercial database workloads on simultaneous multithreaded processors.

SMT performance on commercial databases is of interest for three related reasons. First, a database workload is intrinsically multithreaded, providing a natural source of threads for an SMT processor. Second, many database workloads are memory-intensive and lead to extremely low processor utilization. For example, our studies show that a transaction processing workload achieves only 0.79 IPC on an 8-wide, out-of-order superscalar with 128KB L1 caches — less than 1/4 the throughput of the SPEC suite. As a result, there is great potential for increased utilization through simultaneous multithreaded instruction issue. Third, but somewhat troubling, SMT's fine-grained sharing of the caches among multiple threads may seriously diminish memory system performance, because database workloads can stress the cache even on a single-threaded superscalar. Therefore, while SMT seems a promising candidate to address the low instruction throughput on database systems, the memory system behavior of databases presents a potentially serious challenge to the multithreaded design approach. An SMT system may need to carefully manage inter-thread interactions in the memory system in order to control cache interference.

That is the focus of this chapter. We begin with an analysis of the memory-system behavior of our database workloads, identifying the different memory access patterns of a DBMS's internal memory regions (instruction segment, private data, database buffer cache, and shared metadata) and the implications those patterns have for SMT memory system design. With an improved understanding of database memory behavior, we then investigate cache interference in database workloads on SMT processors. First, we identify the causes of cache interference and quantify the effect of cache conflicts in both the instruction and data caches on SMT processors. Second, we evaluate software alternatives for minimizing destructive inter-thread cache interference. These techniques manage memory addresses in an SMT-conscious manner, enabling an SMT processor to exploit thread-level parallelism to hide memory latencies and achieve increases in throughput.

The organization of this chapter is as follows. Section 6.1 describes the methodology used in our simulation-based study. Section 6.2 characterizes the memory-system behavior of database systems running on-line transaction processing and decision support system benchmarks. Section 6.3 quantifies the effects of cache interference and identifies the causes of these conflicts. Section 6.3 also investigates techniques for minimizing cache interference. Section 6.4 compares the performance of the OLTP and DSS workloads on SMT and a wide-issue superscalar, explaining the architectural basis for SMT's higher instruction throughput and demonstrating the performance gains from properly managing the shared caches. Section 6.5 discusses related work and Section 6.6 summarizes.

6.1 Methodology

To investigate database memory system behavior on SMT processors, we instrumented and measured the Oracle version 7.3.2 database system executing under Digital UNIX on DEC Alpha processors. This section describes the methodology used for our experiments. We begin by presenting details of the hardware model implemented by our trace-driven processor simulator. We then describe the workload used to generate traces and our model for the general execution environment of database workloads.

6.1.1 SMT processor model

For the database studies, our simulated SMT processor (including branch prediction and the memory hierarchy) is similar to the one modeled in Chapter 5. We have also augmented the processor model to accurately simulate timer interrupts and precise exceptions. This functionality is essential for modeling operating system scheduling activity, which we describe below. TLB misses also invoke the exception mechanism. Before handling a TLB miss, the processor must ensure that the excepting instruction is non-speculative. Therefore, the TLB miss is not handled until the excepting instruction reaches the head of its active list (as in the MIPS R1000). A 20 cycle TLB miss penalty is then incurred to model the TLB miss handler.

6.1.2 Simulating database workloads

Compared to typical benchmarks, such as SPEC and SPLASH, commercial workloads have substantially more complex execution behavior. Accurate simulation of these applications must capture this complexity, especially I/O latencies and the interaction of the database with the operating system. We therefore examined the behavior of the Oracle DBMS and the underlying Digital UNIX operating system to validate and strengthen our simulation methodology. Though DBMS source code was not available, we used both the Digital Continuous Profiling Infrastructure (DCPI) [7] and separate experiments running natively on Digital AlphaServers to understand database system behavior and extract appropriate parameters for our simulations. The remainder of this section describes the experimental methodology, including the workloads, trace generation, operating system activity (including modelling of I/O), and synchronization.

The database workload

On-line transaction processing (OLTP) and decision support systems (DSS) dominate the workloads handled by database servers; our studies use two workloads, one representative of each of these domains. Our OLTP workload is based on the TPC-B benchmark [128]. Although TPC-C has supplanted TPC-B as TPC's current OLTP benchmark, the two workloads have similar processor and memory system characteristics [8]. We chose TPC-B because it is easier to set up and run.

The OLTP workload models transaction processing for a bank, where each transaction corresponds to a bank account deposit. Each transaction is small, but updates several database tables (e.g., the teller, branch, and account). OLTP workloads are intrinsically parallel, and therefore database systems typically employ multiple server processes to process client transactions and hide I/O latencies.

In decision support systems, queries execute against a large database to answer critical business questions. The database consists of several inter-related tables, such as parts, nations, customers, orders, and lineitems. Our DSS workload is based on query 6 of the TPC-D benchmark [129], which models the database activity for a business that manages, sells, or distributes products worldwide. The query scans the largest table (*lineitem*) to quantify the amount of revenue increase that would have resulted from eliminating certain discounts in a given percentage range in a given year. This query is representative of DSS workloads; other TPC-D queries tend to have similar memory system behavior [8].

Trace generation

Commercial database applications require considerable tuning to achieve optimal performance. Because the execution time of different workload components (user, kernel, I/O, etc.) may vary depending on this level of optimization and customization, we extensively tuned Oracle v.7.3.2 and Digital UNIX to maximize database performance when running natively on a 4-processor Digital AlphaServer 4100. Using the best-performing configuration, we instrumented the database application with ATOM [116] and generated a separate instruction trace file for each server process. We then fed these traces to our cycle-level SMT simulator, whose parameters were described above. In each experiment, our workload consisted of 16 processes (threads), unless otherwise noted. For the OLTP workload, each process contains 315 transactions (a total of 5040) on a 900MB database. For a single OLTP experiment, we simulate roughly 900M instructions. For our DSS workload, scaling is more complex, because the run time (and therefore, simulation time) grows linearly with the size of the database. Fortunately, the DSS query exhibits very consistent behavior throughout its execution, so we could generate representative traces using sampling techniques [8]. With the sampled traces, each of our DSS experiments simulate roughly 500M instructions from queries on a 500MB database.

Operating system activity

Although ATOM generates only user-level traces, we took several measures to ensure that we carefully modelled operating system effects. While some previous studies have found that operating system kernel activity can dominate execution time for OLTP workloads [40, 83, 104], we found that a well-tuned database workload spends most of its time in user-level code. Using DCPI, we determined that for OLTP, roughly 70% of execution time was spent in user-level code, with the rest in the kernel and the idle loop. For DSS, kernel and idle time were negligible. These measurements therefore verified that our traces account for the dominant database activity.

In addition, we monitored the behavior of Digital UNIX to ensure that our simulation framework models the behavior of the operating system scheduler and underlying I/O subsystem to account for I/O latencies. We use a simple thread scheduler when there are more processes (threads) than hardware contexts. Although the scheduler can preempt threads at the end of a 500K-cycle scheduling quantum, most of the scheduling decisions are guided by hints from the server processes via four UNIX system calls: *fread*, *fwrite*, *pid_block*, and *pid_unblock*. We therefore annotate the traces to indicate where the server processes call these routines.

The OLTP workload uses *fread* and *fwrite* calls for pipe communication between the client (the application) and the server process. Writes are non-blocking, while reads have an average latency of 14,500 cycles on the AlphaServer. Our simulator models this *fread* latency and treats both *fread* and *fwrite* as hints to the scheduler to yield the processor. The other important system call, *pid_block*, is primarily used during the commit phase of each transaction. During transaction commit, the logwriter process must write to the log file. The *pid_block* call is another scheduler hint that yields the CPU to allow the logwriter to run more promptly.

For our DSS workload, system calls are infrequent, but the server processes periodically invoke *freads* to bring in new 128KB database blocks for processing.

Our simulation experiments also include the impact of the I/O subsystem. For the OLTP workload, we use a 1M cycle latency (e.g., 1ms for a 1 GHz processor) for the logwriter's small (about 8KB) file writes. This latency models a fast I/O subsystem with non-volatile RAM to improve the performance of short writes. For DSS, we model database reads (about 128KB) with 5M cycle latencies. All of our experiments use 16 processes, but in systems with longer I/O latencies, more processes will be required to hide I/O.

Synchronization

Oracle's primary synchronization primitive uses the Alpha's load-locked/store-conditional instructions, and higher level synchronization primitives (e.g., test-and-test-and-set locks) are built upon this mechanism. However, on an SMT processor, this conventional spinning synchronization can have adverse effects on threads running in other contexts, because the spinning instructions consume processor resources that could be used more effectively by the other threads. We therefore use hardware blocking locks (Section 2.3), which are a more efficient synchronization mechanism for SMT processors. To incorporate blocking synchronization in the simulations, we replaced the DBMS's synchronization scheme with blocking locks in the traces.

6.2 Database workload characterization

This section characterizes the memory-system behavior of our commercial OLTP and DSS workloads, providing a basis for the detailed SMT architectural simulations presented in Section 6.3.

Previous architectural studies have shown that commercial workloads can exhibit strikingly high cache miss rates [24, 40, 83, 104, 127]. In the past, these miss rates were less significant, because I/O latency was the limiting factor for database performance. However, with the latest generation of commercial database engines employing numerous processes, disk arrays, increased I/O concurrency, and huge memories, many of the I/O limitations have been addressed [42]. Memory system performance is now the crucial problem: the high miss rates of database workloads, coupled with long memory latencies, make the design of future CPUs for database execution a significant challenge.

A database's poor memory system performance causes a substantial instruction throughput bottleneck. For example, our processor simulations (described in the next section) show that the OLTP workload achieves only 0.79 instructions per cycle on an 8-wide, single-threaded, superscalar with 128KB L1 caches (compared to 3.4 IPC for simulations of the SPEC and SPLASH benchmarks in Chapter 5 on the same processor). The OLTP workload achieves only 0.26 IPC with 32KB L1 caches! Because of its latency-hiding capability, simultaneous multithreading has the potential to substantially improve the single-threaded superscalar's low IPC. On the other hand, SMT could exacerbate conflicts in the already-overloaded caches beyond its ability to hide the latencies. An evaluation of this issue requires an analysis of the thread working sets, their access patterns, the amount of inter-thread sharing, and the memory-access patterns that can lead to high miss rates. We provide that analysis in this section.

Our studies of memory-system behavior focus on the performance of the database server processes that dominate execution time for commercial workloads. In Oracle's dedicated mode, a separate server process is associated with each client process. Each server process accesses memory in one of 3 segments:

- The *instruction text* segment contains the database code and is shared among all database processes.
- The *Program Global Area* (PGA) contains per-process data, such as private stacks, local variables, and private session variables.
- The *Shared Global Area* (SGA) contains the database buffer cache, the data dictionary (indices and other metadata), the shared SQL area (which allows multiple users to share a single copy of an SQL statement), redo logs (for tracking data updates and guiding crash recovery), and other shared resources. The SGA is the largest region and is shared by all server processes. For the purposes of this study, we consider the database buffer cache to be a fourth region (which we'll call the *SGA buffer cache*), separate from the rest of the SGA (called *SGA-other*), because its memory access pattern is quite distinct.

To better understand memory behavior, we compare and analyze the memory access patterns of these regions on both OLTP and DSS workloads.

6.2.1 OLTP workload characterization

As described in the previous section, we traced our OLTP workload and then used these traces to analyze cache behavior for a traditional, single-threaded uniprocessor. Table 6.1 presents the results. Overall, this data confirms the aggregate cache behavior of transaction processing workloads found by others; namely, that they suffer from higher miss rates than scientific codes (at least as exhibited by SPEC and SPLASH benchmarks), with instruction misses a particular problem [24, 40, 83, 104]. For example, columns 2 and 3 of Table 6.1 show that on-chip caches are relatively ineffective both at current cache sizes (32KB) and at larger sizes (128KB) expected in next-generation processors. In addition, instruction cache behavior is worse than data cache behavior, having miss rates of 23.3% and 13.7% for 32K and 128K caches,

Table 6.1: Memory behavior characterization for OLTP (16 processes, 315 transactions each) on a single-threaded uniprocessor. Results are shown for both 32KB and 128KB caches. All caches are 2-way associative.

Program Segments	OLTP					
	L1 cache miss rate		Memory footprint	Avg. # of refs per 64-byte block	Avg. # accesses to a block until a cache conflict	
	32KB	128KB			32KB	128KB
Instruction	23.3%	13.7%	556KB	52K	3	4
PGA	8.4%	7.4%	1.3MB	14K	8	11
SGA buffer cache	7.5%	6.8%	9.3MB	66	9	12
SGA-other	17.5%	12.9%	26.5MB	169	3	5
All data segments	10.1%	8.4%	37.1MB	630	7	9

respectively.

To provide more detail, Table 6.1 also shows a breakdown of cache-access information by memory region. Here we see that the high miss rates are partly attributable to OLTP's large memory footprints, which range from 556KB in the instruction segment up to 26.5MB in SGA-other. The footprints for all four regions easily exceed on-chip cache sizes; for the two SGA areas, even large off-chip caches are insufficient.

Surprisingly, the high miss rates are not a consequence of a lack of instruction and data reuse. Column 5 shows that, on average, blocks are referenced very frequently, particularly in the PGA and instruction regions. Cache reuse correlates strongly with the increase in the memory footprint size as transactions are processed. For example, we found that as more of the database is accessed, the memory footprint of the SGA buffer cache continues to grow and exceeds that of the SGA-other, whose size levels off over time; reuse in the buffer cache is therefore relatively low. In contrast, the PGA and instruction segment footprints remain fairly stable over time, and reuse is considerably larger in those regions.

High reuse only reduces miss rates, however, if multiple accesses to cache blocks occur over a short enough period of time that the blocks are still cache-resident. Results in columns 6 and 7 show that the frequency of block replacement strongly and inversely correlates with miss rates, for all segments. Replacement is particularly frequent in the instruction segment, where cache blocks are accessed on average only 3 or 4 times before they are potentially replaced[1], either by a block from this thread or another thread. So, despite a relatively small memory footprint and high reuse, the instruction segment's miss rate is high.

In summary, all three of these factors, large memory footprints, frequency of memory reuse, and the interval length between cache conflicts, make on-chip caching for OLTP relatively ineffective.

The "critical" working set

Within a segment, cache reuse is not uniformly distributed across blocks, and for some segments is highly skewed, a fact hidden by the averaged data in Table 6.1. To visualize this, Figure 6.1 characterizes reuse in the four memory regions. To obtain data points for these graphs, we divided the memory space into 64-byte (cache-line sized) blocks and calculated how many times each was accessed. The black line (the higher of the two lines) plots a cumulative histogram of the percentage of *blocks* that are accessed n times or less; for example, the top circle in Figure 6.1b says that for the PGA, 80% of the blocks are accessed 20,000 times or less. The gray line (bottom) is a cumulative histogram that plots the percentage of total *references* that occurred to blocks accessed n times or less; the lower circle in Figure 6.1b shows that those blocks accessed 20,000 times or less account for only 25% of total references. Alternatively, these two points indicate that 20% of the blocks are accessed more than 20,000 times and account for 75% of all the references. In other words, for the PGA, a minority of the memory blocks are responsible for most of the memory references. (The curves in Figure 6.1b are all cumulative distributions and thus reach 100%; we have omitted part of the right side of the graphs for most cases because the curves have long tails.)

All four regions exhibit skewed reference distributions, but to different extents.

[1]. Columns 6 and 7 measure inherent cache mapping conflicts using a direct-mapped, instead of two-way associative cache. Even though this may overestimate the number of replacements (compared to two-way), the relative behavior for the different data segments is still accurate.

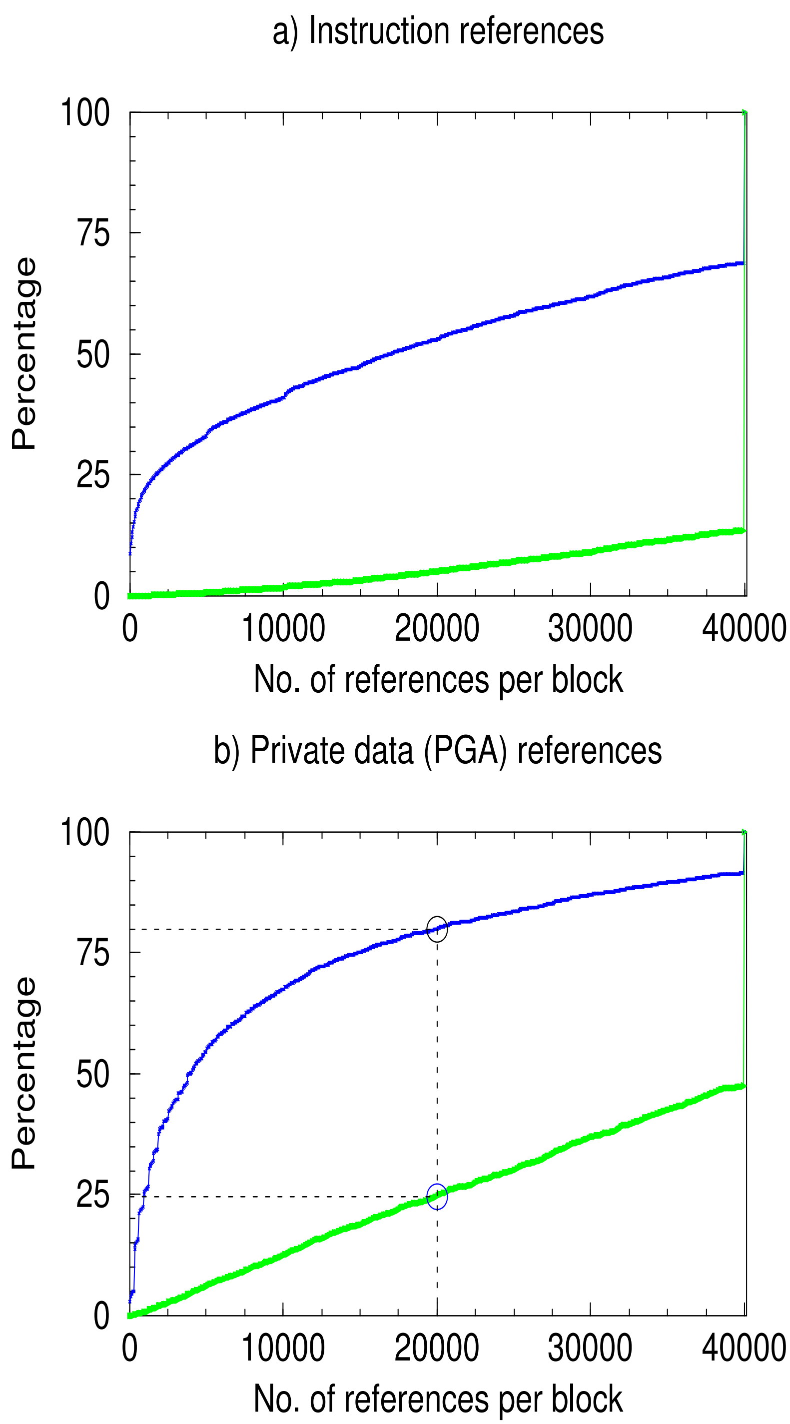
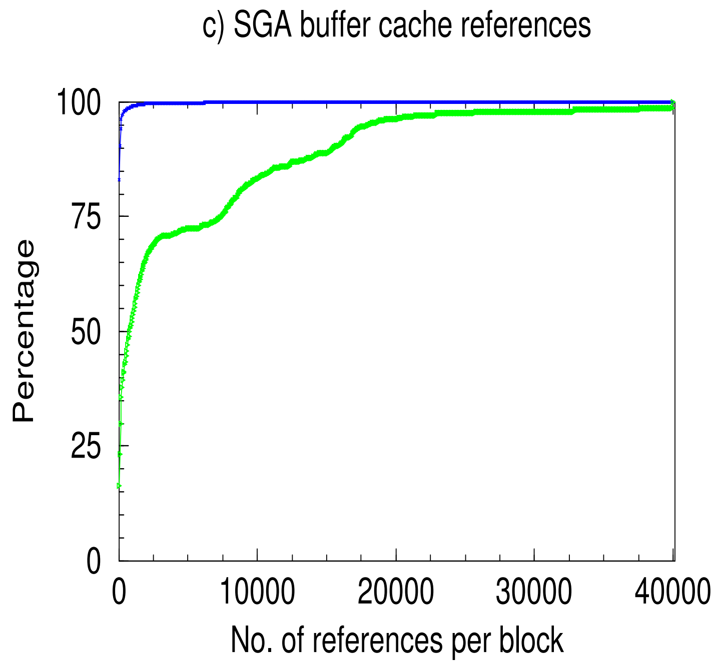


Figure 6.1: Locality profiles for OLTP (instruction and PGA regions).

In each graph, the top curve plots the cumulative percentage of 64-byte blocks accessed n times or less; the lower curve plots the cumulative percentage of references made to blocks accessed n times or less.



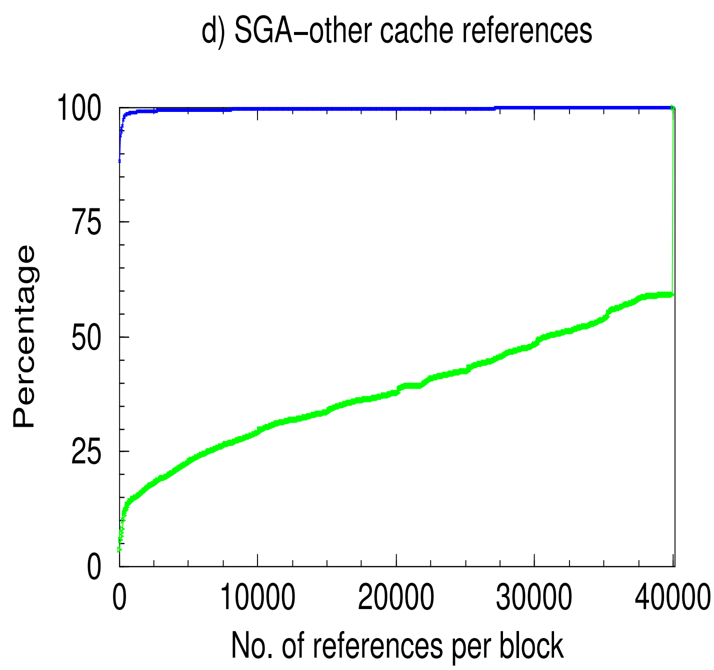


Figure 6.1 (continued): Locality profiles for OLTP (SGA buffer cache and SGA-other regions).

Comparing them at the highest reuse data point plotted in Figure 6.1b, i.e., more than 40K accesses per block, 31% of the blocks in the instruction segment account for 87% of the instruction references (Figure 6.1a), 8.5% of the blocks in the PGA account for 53% of the references (Figure 6.1b), and a remarkable 0.1% of the blocks in SGA-other account for 41% of the references (Figure 6.1d). The SGA buffer cache's reference distribution is also skewed (9% of the blocks comprise 77% of the references); however, this point occurs at only 100 accesses. Consequently, most blocks in the SGA buffer cache (91%) have very little reuse and the more frequently used blocks comprise a small percentage of total references.

Reference behavior that is skewed to this extent strongly implies that the "critical" working set of each segment, i.e., the portion of the segment that absorbs the majority of the memory references, is much smaller than the segment's memory footprint. As an example, the SGA-other blocks mentioned above are three orders of magnitude smaller (26KB) than this segment's memory footprint (26.5MB). The implication for simultaneous multithreading is that, for the segments that exhibit skewed reference behavior and make most of their references to a small number of blocks (instruction, PGA, and SGA-other segments), there will be some performance-critical portion of their working sets that fit comfortably into SMT's context-shared caches.

6.2.2 DSS workload characterization

As with OLTP, we used traces of the DSS workload to drive a simulator for a single-threaded uniprocessor. Our results, shown in Table 6.2, indicate that the DSS workload should cause fewer conflicts in the context-shared SMT caches than OLTP, because its miss ratios are lower, reuse is more clustered, and the segments' critical working sets are smaller. The instruction and (overall) data cache miss rates, as well as those of 2 of the 3 data segments (columns 2 and 3 of Table 6.2), are negligible, and cache reuse per block (columns 6 and 7) is sometimes even an order of magnitude higher. Because of more extreme reference skewing and/or smaller memory footprints, the cache-critical working sets for all segments except the SGA buffer cache are easily cacheable on an SMT. In the instruction region, 98% of the references are made to only 6KB of instruction text (Figure 6.2); and 253 blocks (16KB) account for 75% of PGA references. SGA-other is even more skewed, with more than 97% of the references touching only 51 blocks or

Table 6.2: Memory behavior characterization for DSS (16 processes) on a single-threaded uniprocessor.

Program Segments	DSS					
	L1 cache miss rate		Memory footprint (sample)	Avg. # of refs per 64-byte block	Avg. # accesses to a block until a cache conflict	
	32KB	128KB			32KB	128KB
Instruction	0.5%	0.0%	43.3KB	216K	11	43
PGA	0.8%	0.7%	2.2MB	3.8K	38	102
SGA buffer cache	9.3%	8.0%	2.7MB	383	7	10
SGA-other	0.4%	0.2%	878KB	3.4K	43	117
All data segments	1.5%	1.2%	5.8MB	2.8K	29	59

3KB.

The SGA buffer cache has a much higher miss rate than the other segments (8%), because the query scans through the large *lineitem* table and little reuse occurs. The buffer cache is so uniformly accessed that its critical working set and memory footprint are almost synonymous; 99% of the blocks are touched fewer than 800 times, as shown by the locality histogram in Figure 6.2c.

The scalability of DSS's locality profile is an important issue as DSS databases continue to grow in size. The reuse profiles demonstrate that the locality and good cache behavior in this workload scales to much larger databases. With larger databases (and therefore, longer-running queries), the instruction and PGA references dominate, but their working sets should remain small and easily cacheable. Although the footprints of both SGA segments grow with larger databases, DSS has good spatial locality independent of the size of the cache, and therefore references to these regions have minimal effects on locality.

6.2.3 Summary of workload characterization

This section analyzed the memory-system behavior of the OLTP and DSS workloads

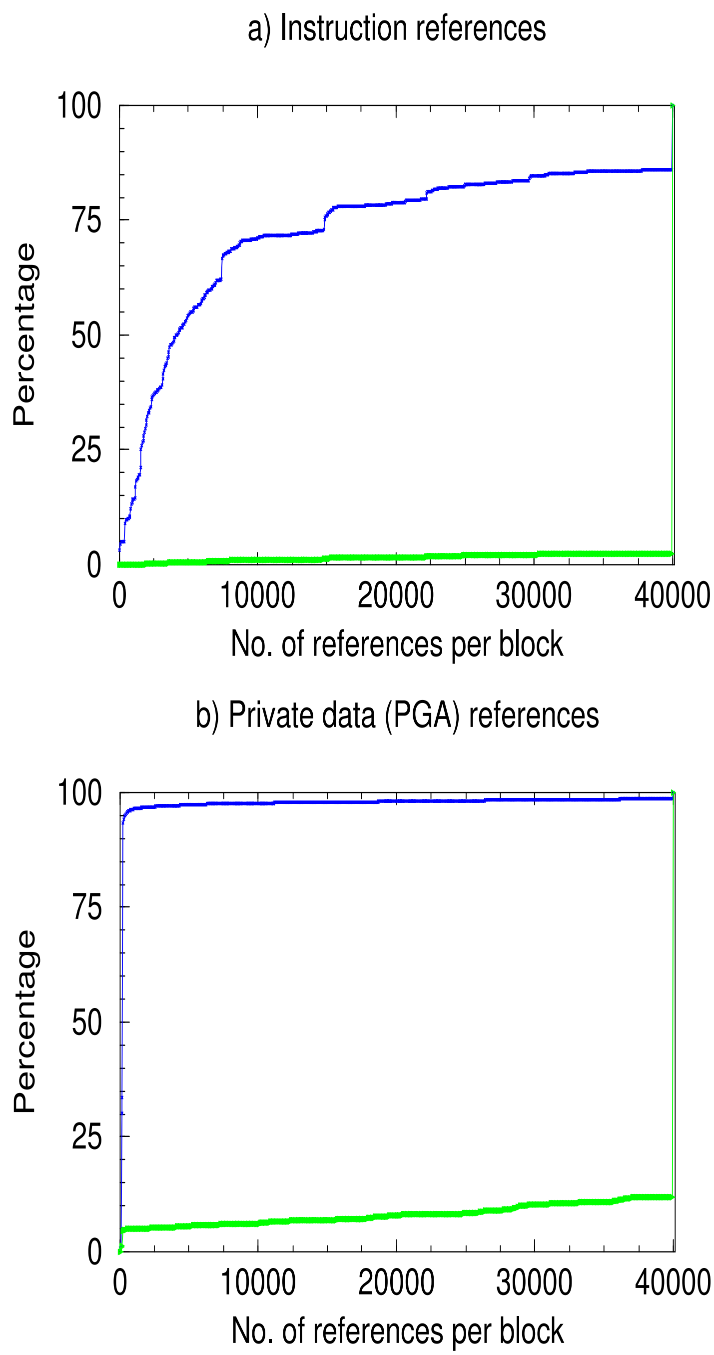


Figure 6.2: Locality profiles for DSS workload (instruction and PGA regions).

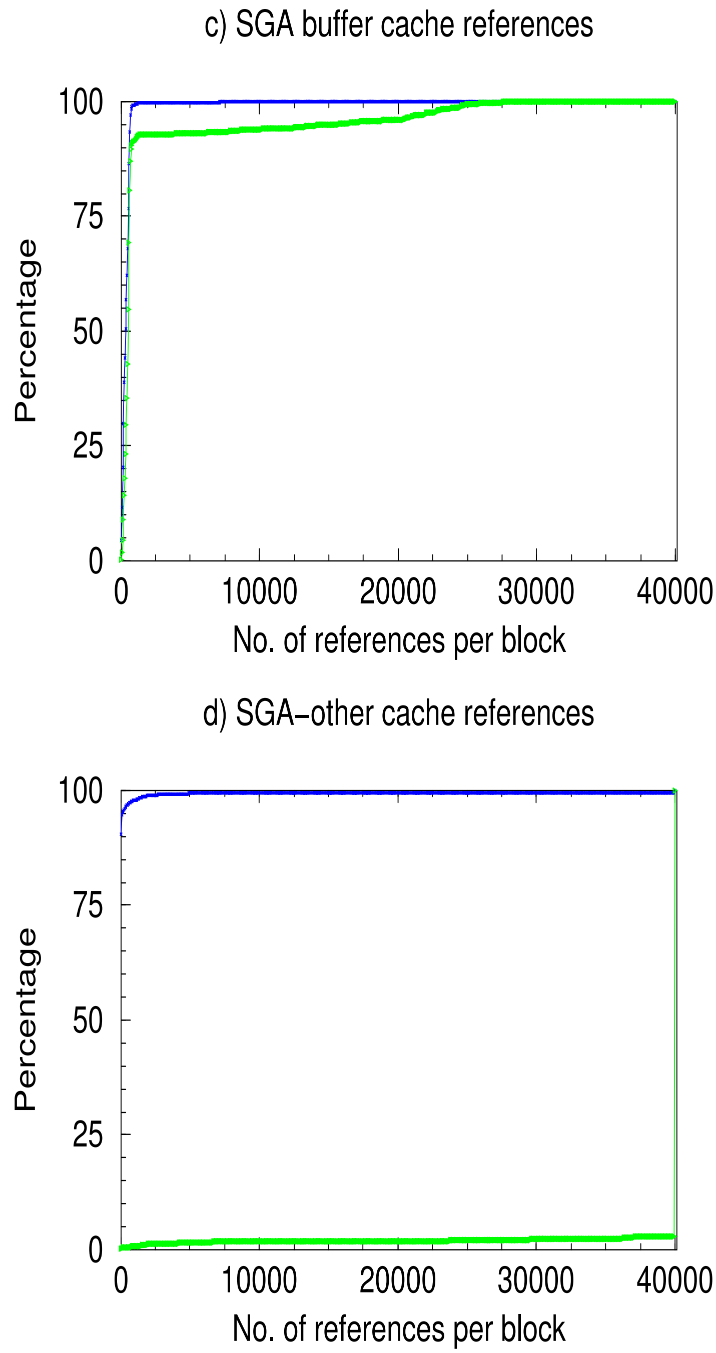


Figure 6.2 (continued): Locality profiles for DSS workload (SGA buffer cache and SGA-other regions).

in detail. Overall, we found that while the footprints (particularly for OLTP) are large for the various memory regions, there is good temporal locality in the most frequently accessed blocks, i.e., a small percentage of blocks account for most of the references. Thus, it is possible that even with multithreading, the "critical" working sets will fit in the caches, reducing the degradation on cache performance due to inter-thread conflicts.

Recall, however, that simultaneous multithreading interleaves per-thread cache accesses more finely than a single-threaded uniprocessor. Thus, inter-thread competition for cache lines will rise on an SMT, causing consecutive, per-thread block reuse to decline. If cross-thread accesses are made to distinct addresses, thus increasing inter-thread conflicts, SMT will have to exploit temporal locality more effectively than the uniprocessor. But if the accesses occur to thread-shared blocks, inter-thread conflicts and misses will decline. The latter should be particularly beneficial for the instruction segment, where the various threads tend to execute similar code.

In the next section, we explore these implications, using a detailed simulation of an SMT processor executing the OLTP and DSS workloads.

6.3 Multi-thread cache interference

This section quantifies and analyzes the cache effects of OLTP and DSS workloads on simultaneous multithreaded processors. On conventional (single-threaded) processors, a DBMS employs multiple server processes to hide I/O latencies in the workload. Context switching between these processes may cause cache interference (i.e., conflicts), as blocks from a newly-scheduled process evict useful cache blocks from descheduled processes; however, once a thread begins to execute, it has exclusive control of the cache for the duration of its execution quantum. With simultaneous multithreading, thread execution is interleaved at a much finer granularity (within a cycle, rather than at the coarser context-switch level). This fine-grained, simultaneous sharing of the cache potentially changes the nature of inter-thread cache interference. Understanding this interference is therefore key to understanding the performance of database workloads on SMT.

In the following subsections we identify two types of cache interference: *destructive interference* occurs when one thread's data replaces another thread's data in the cache, resulting in an increase in inter-thread conflict misses; *constructive interference* occurs when data loaded by one thread is accessed by another simultaneously-scheduled thread, resulting in fewer misses. We examine the effects of both destructive and constructive cache interference when running OLTP and DSS workloads on an SMT processor, and evaluate operating system and application techniques for minimizing inter-thread cache misses caused by destructive interference.

6.3.1 Misses in a database workload

We begin our investigation by analyzing per-segment misses for both OLTP and DSS workloads on an SMT processor. The results shown here were simulated on our 8-context SMT processor simulator described in Section 2. For some experiments we simulate fewer than 8 contexts as well, to show the impact of varying the number of simultaneously-executing threads.

In the previous section we saw the individual miss rates for the four database memory regions, executing on a single-threaded uniprocessor. Table 6.3 shows the proportion of total misses due to each region, when executing on our 8-context SMT processor. From Table 6.3, we see that the PGA region is responsible for the majority of L1 and L2 misses. For example, the PGA accounts for 60% of the L1 misses and 98% of the L2 misses for OLTP (and 7% and 58% of total references to L1 and L2, respectively), making it the most important region for analysis.

Table 6.3: Proportion of total misses (percent) due to each segment on an 8-context SMT. For the level 1 cache, we combined data and instruction misses.

	Cache	instruction text	PGA	SGA buffer cache	SGA-other
OLTP	L1	28.6	60.0	0.9	10.5
	L2	0.2	98.1	0.3	1.4
DSS	L1	0.0	96.0	3.6	0.3
	L2	0.0	99.9	0.1	0.0

The PGA contains the per-process data (e.g., private stacks and local variables) that are used by each server process. PGA data is laid out in an identical fashion, i.e., at the same virtual addresses, in each process' address space. Furthermore, there are several hot spots in the PGA that are accessed throughout the life of each process. Consequently, SMT's fine-grained multithreading causes substantial destructive interference between the same virtual addresses in different processes. These conflicts also occur on single-threaded processors, but to a somewhat lesser extent, because context switching is much coarser grained than simultaneous-multithreaded instruction issue. (PGA accounts for 71% of the misses on the single-threaded processor, compared to 84% on the 8-context SMT.)

The SMT cache organization we simulate is a virtually-indexed/physically-tagged L1 cache with a physically-indexed/physically-tagged L2 cache. This structure is common for modern processors; it provides fast lookup for the L1 cache and ease of management for the L2 cache. Given this organization, techniques that alter the per-process virtual-address-space layout or the virtual-to-physical mapping could affect the miss rates for the L1 and L2 caches, respectively, particularly in the PGA. We therefore evaluate combinations of two software mechanisms that might reduce the high miss rates: virtual-to-physical page-mapping schemes and application-based, per-process virtual-address-space offsetting.

6.3.2 Page-mapping policies

Because the operating system chooses the mapping of virtual to physical pages when allocating physical memory, it plays a role in determining L2 cache conflicts. Operating systems generally divide physical memory page frames into colors (or bins); two physical pages have the same color if they index into the same location in the cache. By mapping two virtual pages to different colors, the page-mapping policy can eliminate cache conflicts between data on the two pages and improve cache performance [66].

The two most commonly-used page-mapping policies are page coloring and bin hopping. Page coloring exploits spatial locality by mapping consecutive virtual pages to consecutive physical page colors. IRIX, Solaris/SunOS and Windows NT augment this basic page coloring algorithm by either hashing the process ID with the virtual address or using a random seed for a process's initial page color. In contrast, Digital UNIX uses bin hopping, also known as first-touch. Bin hopping exploits temporal locality by cycling through page colors sequentially as it maps new virtual pages. Because page mappings are established based on reference order (rather than address-space order), pages that are mapped together in time will not conflict in the cache.

Our experiments indicate that, because multithreading magnifies the number of conflict misses, the page-mapping policy can have a large impact on cache performance on an SMT processor. Table 6.4 shows the L2 cache miss rates for OLTP and DSS workloads for various mapping schemes. The local miss rate is the number of L2 misses as a percentage of L2 references; the global miss rate is the ratio of L2 misses to total memory references. Bin hopping avoids mapping conflicts in the L2 cache most effectively, because it is likely to assign identical structures in different threads to non-conflicting physical pages. Consequently, miss rates are minuscule, and are stable across all numbers of hardware contexts, indicating that the OLTP and DSS "critical" working sets fit in a 16MB L2 cache. In contrast, page coloring follows the data memory layout; since this order is common to all threads (in the PGA), page coloring incurs more conflict misses, and increasingly so with more hardware contexts. In fact, with 4 and 8 contexts on DSS, almost all L2 cache references are misses. Hashing the process ID with

Table 6.4: Global and local L2 cache miss rates (in percentages) for 16 threads running on an SMT with 1-8 contexts. The global miss rate is the ratio of L2 misses to total references; the local miss rate is the ratio of L2 misses to L2 references.

Workload	**Page-mapping technique**	**Type of L2 miss rate**	**Number of contexts**			
			1	**2**	**4**	**8**
OLTP	Bin hopping	global	0.3	0.3	0.3	0.3
		local	2.7	2.7	2.6	2.4
	Page coloring	global	3.4	3.5	5.1	6.7
		local	34.4	38.0	50.3	58.9
	Page coloring with process id hash	global	1.8	1.6	1.4	1.2
		local	17.3	16.1	12.0	8.7
DSS	Bin hopping	global	0.0	0.0	0.0	0.0
		local	5.3	4.4	0.4	0.3
	Page coloring	global	0.3	0.3	6.6	9.1
		local	39.9	41.6	94.8	96.1
	Page coloring with process id hash	global	0.2	0.2	0.2	0.2
		local	32.5	28.1	2.7	2.1

the virtual address improves page coloring performance, but it still lags behind bin hopping.

Note that some of these conflict misses could also be addressed with higher degrees of associativity or with victim caching, but these solutions may either slow cache access times (associativity) or may have insufficient capacity to hold the large number of conflict misses in OLTP and DSS workloads (victim caches).

6.3.3 Application-level offsetting

Although effective page mapping reduces L2 cache conflicts, it does not impact on-chip L1 data caches that are virtually-indexed. In the PGA, in particular, identical virtual pages in the different processes will still conflict in the L1, independent of the physical page-mapping policy. One approach to improving the L1 miss rate is to "offset" the conflicting structures in the virtual address spaces of the different processes. For example, the starting virtual address of each newly-created process or segment could be shifted by (page size * process ID) bytes. This could be done manually in the application or by the loader.

Table 6.5 shows the L1 miss rates for the three page-mapping policies, both with and without address-space offsetting. The data indicate that using an offset reduced the L1 miss rate of all numbers of hardware contexts roughly to that of a wide-issue superscalar. Without offsetting, L1 miss rates doubled for OLTP and increased up to 12-fold for DSS, as the number of hardware contexts was increased to 8.

6.3.4 Constructive interference

Simultaneous multithreading can exploit instruction sharing to improve instruction cache behavior, whether the instruction working set is large (OLTP) or small (DSS). In these workloads, each instruction block is touched by virtually all server threads. The heavy instruction sharing generates constructive cache interference, as threads frequently prefetch instruction blocks for each other.

Each server thread for OLTP executes nearly identical code, because transactions are similar. A single-threaded superscalar cannot take advantage of this code sharing, because its threads are resident only on a coarse scheduling granularity. For example, a particular routine may be executed only near the beginning of a transaction. By the time

Table 6.5: Local L1 cache miss rates (in percentages) for 16 threads running on an SMT, with and without offsetting of per-process PGA data. For these experiments, an offset of 8KB * thread ID as used.

Page-mapping technique	**Application offsetting**	**OLTP**				**DSS**			
		Number of contexts				**Number of contexts**			
		1	**2**	**4**	**8**	**1**	**2**	**4**	**8**
Bin hopping	no offset	8.2	8.9	12.3	16.0	1.2	1.4	15.0	18.8
	offset	8.4	8.5	8.6	8.7	1.2	1.3	1.6	2.0
Page coloring	no offset	7.9	8.6	12.5	17.0	1.2	1.3	17.7	25.7
	offset	8.3	8.5	8.7	8.8	1.2	1.3	1.6	2.2
Page coloring with process id hash	no offset	8.1	8.9	12.9	18.5	1.2	1.4	15.0	19.3
	offset	8.4	8.7	8.9	9.1	1.2	1.3	1.5	2.2

the routine is re-executed by the same server process, the code has been kicked out of the cache. This occurs frequently, as the instruction cache is the largest performance bottleneck on these machines. On an 8-context SMT, however, the finer-grain multithreading increases the likelihood that a second process will re-execute a routine before it is replaced in the cache. This constructive cache interference reduces the instruction cache miss rate from 14% to 9%, increasing processor throughput to the point where I/O latencies become the largest bottleneck, as discussed below.

Constructive interference does not require "lock-step" execution of the server threads. To the contrary, scheduling decisions and lock contention skew thread execution; for example, over the lifetime of our 16 thread simulations, the "fastest" thread advanced up to 15 transactions ahead of the "slowest" thread.

With DSS, the instruction cache hit rate is already almost 100% for one context, so constructive interference has no impact.

6.3.5 Summary of multi-thread cache interference

This section examined the effects of cache interference caused by fine-grained multithreaded instruction scheduling on an SMT processor. Our results, which are somewhat surprising, demonstrate that, with appropriate page mapping and offsetting algorithms, an 8-context SMT processor can maintain L1 and L2 cache miss rates roughly commensurate with the rates for a single-threaded superscalar. Even for a less aggressive memory configuration than the one we normally simulate (e.g., 64KB instruction cache, 32KB data caches and 4MB L2 caches), destructive interference remains low. Only when the L2 cache size is as low as 2MB — conservative even for today's database servers — does inter-thread interference have an impact. We have also shown that constructive interference in the I-cache benefits performance on the SMT relative to a traditional superscalar.

Overall, with proper software-mapping policies, the cache behavior for database workloads on SMT processors is roughly comparable to conventional processors. In both cases, however, the absolute miss rates are high and will still cause substantial stall time for executing processes. Therefore, the remaining question is whether SMT's latency-tolerant architecture can absorb that stall time, providing an increase in overall performance. This is the subject of the following section.

6.4 SMT performance on database workloads

This section presents the performance of OLTP and DSS workloads on an SMT processor, compared to a single-threaded superscalar. We compare the various software algorithms for page coloring and offsetting with respect to their impact on instruction throughput, measured in instructions per cycle. The results tell us that SMT is very effective for executing database workloads when applying the appropriate software policies.

Figure 6.3 compares instruction throughput of SMT and a single-threaded superscalar for the alternative page-mapping schemes, both with and without address offsets. From this data we draw two conclusions. First, although the combination of bin hopping and application offsetting provides the best instruction throughput (2.3 IPC for OLTP, 3.9 for DSS) on an 8-wide SMT, several other alternatives are close behind. The marginal performance differences give designers flexibility in configuring SMT systems: if the DBMS provides offsetting in the PGA, the operating system has more leeway in its choice of page-mapping algorithms; alternatively, if an application does not support offsetting, bin hopping can be used alone to obtain almost comparable performance.

Second, it is clear from Figure 6.3 that SMT is highly effective in tolerating the high miss rates of this workload, providing a substantial improvement in instruction

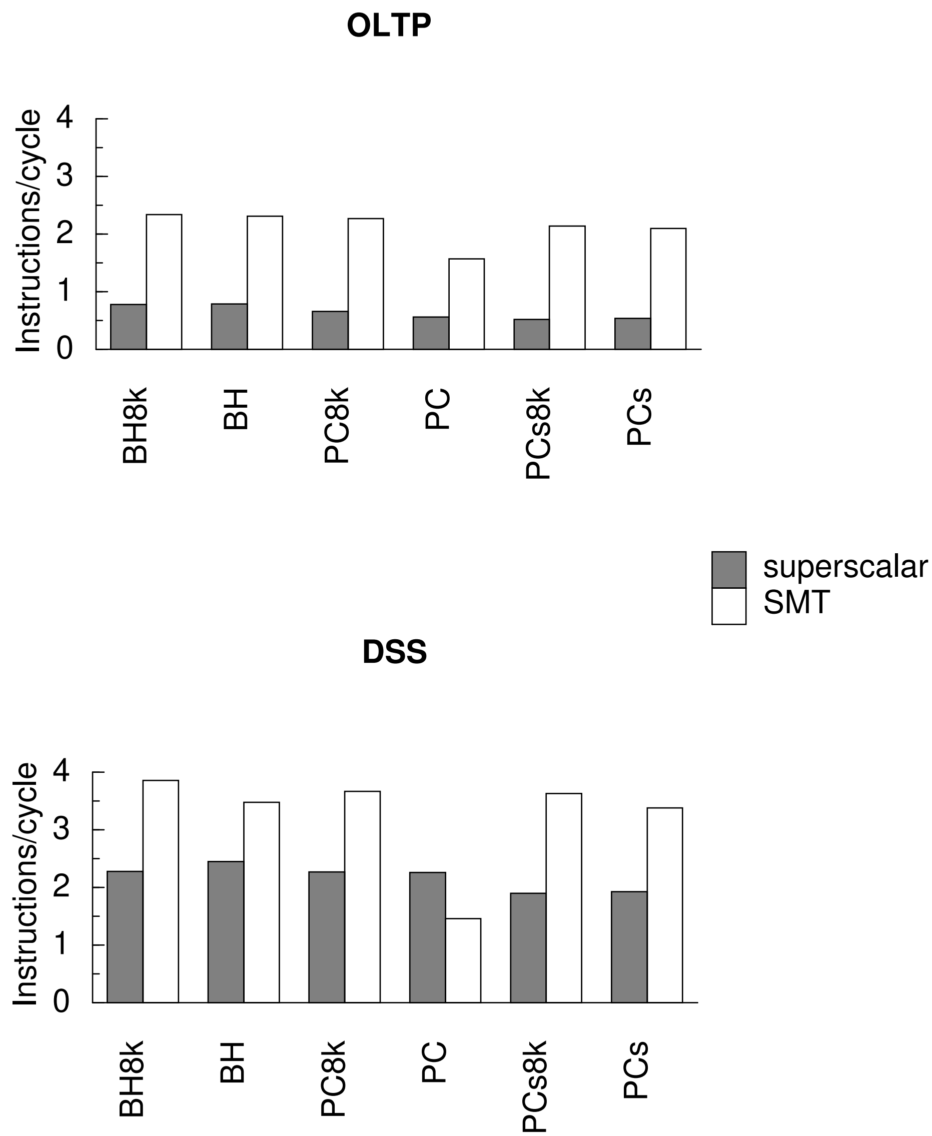


Figure 6.3: Comparison of throughput for various page-mapping schemes on a superscalar and 8-context SMT.

The bars compare bin hopping (BH), page coloring (PC), and page coloring with an initial random seed (PCs), with and without 8KB virtual address offsets.

throughput over the superscalar. For DSS, for example, the best SMT policy (BH8k) achieves a 57% performance improvement over the best superscalar scheme (BH). Even more impressive, for the memory-bound OLTP, the SMT processor shows a 200% improvement in utilization over the superscalar (BH8k for both cases).

SMT achieves this performance gain by minimizing destructive cache interference, while exploiting latency hiding hardware. When offsetting is combined with the proper page-mapping scheme, the OLTP and DSS “critical” working sets fit in the SMT cache hierarchy; an SMT can thus achieve miss rates comparable to a single-threaded superscalar.

6.4.1 The effect of I/O

With memory system performance under control, the main hindrance to better SMT instruction throughput on the OLTP workload is processor idle time due to I/O. Idle cycles occur when all executing threads, both resident and swapped out, are waiting for an I/O event. As more contexts are used, these I/O latencies begin to dominate and cause throughput to level off, as shown in Figure 6.4. With our current 8-context SMT configuration and the 1M cycle I/O latency assumed for future machines, idle time accounted for 8.6% of total execution cycles.

In lieu of I/O system redesign, the primary technique available for reducing idle time is increasing the number of processes. Additional processes can introduce cache interference, so this technique should only be used when idle time is actually a problem. Table 6.6 shows that on a superscalar or 2-context SMT, throughput is worse with 32 processes, because of the cache degradation. On a more aggressive SMT processor (4 or more contexts), however, 32 server processes is actually more appropriate than 16. An aggressive SMT processor can both tolerate the additional cache misses and use additional processes to hide I/O latencies. As shown in Table 6.6, throughput increased to 3.76 on the OLTP workload and reduced idle time to 1.5% of total execution time. Thus, an SMT can actually achieve a speedup of 4.6 over a superscalar!

6.4.2 Architectural analysis of SMT performance gains

Table 6.7 provides additional architectural insight into these large throughput gains, focusing on SMT’s ability to hide instruction and data cache misses, as well as branch

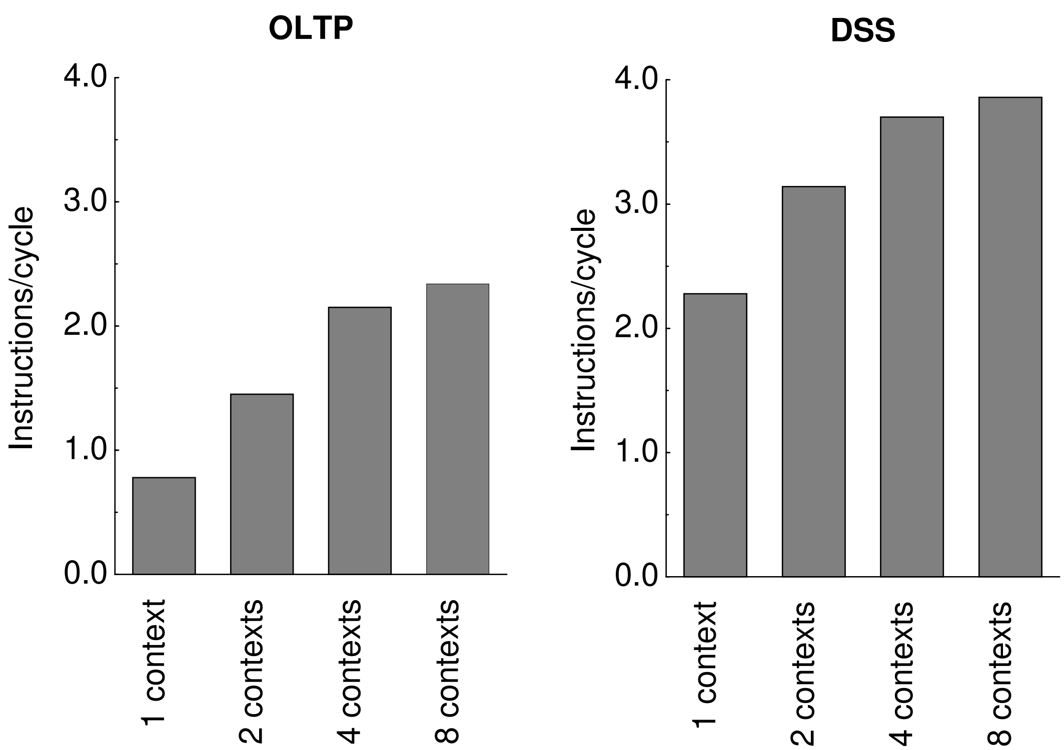


Figure 6.4: SMT throughput with 16 server processes, bin-hopping and application-level offsetting.

Table 6.6: OLTP throughput with 16 and 32 server processes, bin-hopping, and application-level offsetting.

	OLTP			
	Number of contexts			
IPC	**1**	**2**	**4**	**8**
16 processes	0.78	1.45	2.15	2.34
32 processes	0.72	1.38	2.53	3.76

mispredictions. Comparing the average number of outstanding D-cache misses for SMT and the superscalar, we see SMT's greater proficiency at hiding data cache miss latencies. For OLTP, SMT shows a three-fold increase (over the superscalar) in the

Table 6.7: Architectural metrics for superscalar (SS) and 8-context SMT on OLTP and DSS workloads.

Metric	OLTP		DSS	
	SS	SMT	SS	SMT
Avg. # of outstanding D-cache misses	0.66	2.08	0.48	0.75
Wrong-path instructions fetched (%)	60.0	40.0	20.7	9.9
Zero-fetch cycles (%)	55.4	15.5	29.6	1.8
Zero-issue cycles (%)	57.5	18.5	34.9	2.3
6-issue cycles (%)	8.6	32.8	22.4	58.6

amount of memory system parallelism, while DSS shows a 1.5-fold improvement. Since memory latency is more important than memory bandwidth in these workloads, increased memory parallelism translates to greater processor throughput.

Simultaneous multithreading also addresses fetch bottlenecks resulting from branch mispredictions and instruction cache misses. The superscalar fetches 50% and 100% more wrong-path (i.e., wasted) instructions than SMT for OLTP and DSS, respectively. By interleaving instructions from multiple threads, and by choosing to fetch from threads that are making the most effective utilization of the execution resources [133], SMT reduces the need for (and more importantly, the cost of) speculative execution [77]. SMT also greatly reduces the number of cycles in which no instructions can be fetched due to misfetches or I-cache misses. On the DSS workload SMT nearly eliminates all zero-fetch cycles. On OLTP, fetch stalls are reduced by 78%; zero-fetch cycles are still 15.5%, because OLTP instruction cache miss rates are higher.

Finally, the last two metrics illustrate instruction issue effectiveness. The first is the number of cycles in which no instructions could be issued: SMT reduces the number of zero-issue cycles by 68% and 93% for OLTP and DSS, respectively. The second is the percentage of cycles in which the two architectures successfully issued 6 instructions (the maximum possible for this all-integer workload). Not surprisingly, SMT has a huge advantage over the superscalar in finding 6 instructions to issue, showing a 3.8-fold increase for OLTP and a 2.6-fold gain for DSS.

Overall, these results demonstrate that simultaneous multithreading can tolerate memory latencies, exploit inter-thread instruction sharing, and limit inter-thread interference on memory-intensive database workloads. The three-fold throughput improvement for the memory-bound OLTP workload, in particular, shows that SMT's latency tolerance makes SMT an extremely strong candidate architecture for future database servers.

6.5 Related work

This chapter examines two primary issues: database workload performance and cache interference in multithreaded processors. Some other researchers have also investigated these areas, and we discuss the related work in this section.

6.5.1 Database workload performance studies

Although several studies have examined the performance of commercial database workloads, we are aware of only one other study that has looked at their performance on multithreaded architectures. Eickemeyer et al. [32] used trace-driven simulation to evaluate the benefits of coarse-grain multithreading for TPC-C on the OS/400 database. By using two or three threads, throughput increased by 70%; but with more than 3 threads, no further gains were achieved. Because their coarse-grain architecture only switched threads on second-level cache misses, multithreading could not hide L1 instruction and data cache misses, which are very common in commercial workloads. In contrast, simultaneous multithreading interleaves multiple threads on a cycle-by-cycle basis. Therefore it hides latencies more effectively, and can utilize more hardware contexts to achieve greater throughput gains.

While several other studies characterized memory-system behavior of database workloads, only one study analyzed memory access patterns in detail. Trancoso et al. [127] used a public-domain DBMS (Postgres95) to examine the memory access patterns of several DSS queries on cache-coherent shared-memory multiprocessors, and contrasted the cache effects on various data structures in the Postgres95 DBMS. Our study does not examine individual data structures, but contrasts the effects of OLTP and DSS workloads on the behavior of database memory regions in a widely-used commercial database application.

Our paper also extends the cache behavior analysis presented in other prior work [24, 40, 83]. Franklin et al. [40] identified the scarcity of loops and context switches as contributors to high instruction cache miss rates in commercial applications. Maynard et al. [83] highlighted the large instruction footprints and high instruction cache miss rates of OLTP workloads. In another study, Cvetanovic and Bhandarkar [24] used performance counters on the DEC Alpha chip family (21064 and 21164) to identify the performance characteristics of a range of applications, including two commercial workloads. They found that compared to technical workloads, commercial databases stressed the caches and TLBs more heavily, and also spent more time in the operating system and I/O subsystem.

Other prior research has also identified bottlenecks, such as pin bandwidth and I/O, in OLTP workloads. Perl and Sites [98] demonstrated that both high bandwidth and low latency are required to effectively run OLTP and other commercial applications. Their experiments assumed single-cycle instruction latencies and on-chip cache sizes of the Alpha 21064 (8KB I, 8KB D) and 21164 (8KB I, 8KB D, and 96KB unified on-chip L2). In our experiments, we model a more aggressive future processor, with larger on-chip caches and a higher bandwidth L1-L2 bus (256-bit bus with 1 bus transfer per processor clock). Pin bandwidth was therefore a smaller bottleneck; the larger caches and constructive instruction cache interference (due to SMT) also reduced the bandwidth demands.

Rosenblum et al. [104] found that both CPU idle time and kernel activity were significant when running an OLTP workload on Sybase. CPU idle time was greater than 30%, because of disk I/O; kernel activity accounted for 38% of non-idle execution time. However, their configuration had only one server process to handle 20 clients. Our experiments showed that idle time and kernel activity can both be reduced with a well-tuned commercial DBMS. For example, I/O latencies can be hidden more effectively by using more server processes (as is typically done in audit runs of TPC benchmarks).

Finally, a few studies have used OLTP and DSS workloads to evaluate multiprocessor systems or investigate other performance issues. Thakkar and Sweiger [120] identified disk I/O and the system bus as bottlenecks for the TP1 benchmark on a Sequent Symmetry. Lovett and Clapp [78] focused on the L2 cache behavior and scalability of the Sequent STiNG CC-NUMA multiprocessor. Verghese et al. [139] evaluated page migration and replication for CC-NUMA architectures. Torrellas et al.

[126] used Oracle running TP1 to characterize OS cache activity. Kawaf et al. [62] and Piantedosi et al. [99] described optimizations for improving TPC-C performance on Digital AlphaServers.

6.5.2 Cache interference in multithreaded processors

Cache effects of multithreading has been the subject of several studies, but these studies have focused on different aspects of cache interference. Most of the studies looked at interference in multiprogrammed workloads (as opposed to parallel or multithreaded applications), and only one study examined software techniques for minimizing interference [122]. We first discuss related studies that investigate cache conflicts in SMT processors.

Interference in SMT processors

In Section 4.5.1, we discussed two prior studies [133][134], in which we also measured the effects of cache interference, but for multiprogrammed workloads of scientific applications.

Another previous study [76] investigated parallel applications, and found that while SMT increased both the demands on the memory system and the number of cache conflicts, these additional conflict misses degraded performance by less than 1%. Two factors contributed to this minimal effect. First, most L1 conflict misses were captured by a high-bandwidth L2 cache, so the average miss penalty increased only slightly. Second, multithreading, out-of-order execution, and write buffering were able to tolerate this increase in average memory latency.

Hily and Seznec [54] examined the effect of multiprogrammed workloads on cache behavior in SMT processors. They made three conclusions: (1) if L2 cache contention is not modeled, SMT results may be overoptimistic, (2) smaller L2 cache block sizes are more appropriate for an SMT because they reduce the L2 cache occupancy and therefore contention, and (3) L1 caches should be associative.

We have not directly investigated these issues in this study. As mentioned above, however, we found that in an SMT memory system with two levels of on-chip cache, an L2 cache with high associativity and high bandwidth was important for alleviating the effects of inter-thread cache interference [76]. In this dissertation, our memory hierarchy reflects some of the same considerations. Our L1 data cache is two-way associative and our simulator models L2 cache contention. With our memory system, though, L2 cache contention has been less of an issue, because we have assumed a more aggressive memory system targeted for high-end next-generation processors. Our larger on-chip caches and wider L1-L2 bus reduce the demands placed on the L2.

There are also several other differences between our studies. First, Hily and Seznec assume perfect branch prediction and an infinite number of functional units; both assumptions are likely to increase the demands placed on the memory subsystem. Second, our cache miss penalties are longer (12 cycles vs. 5 cycles for L1 misses, and 80 cycles vs. 30 cycles for L2 misses). Third, their study concentrates on the design of the memory system, while this chapter focuses on the sources of cache interference and software techniques for avoiding it. Finally, we examine multithreaded commercial database workloads, while they study a multiprogrammed workload of scientific and engineering applications.

Nemirovsky and Yamamoto [89] used a simple single-level cache model to quantify degradation in cache behavior due to multithreading. Corroborating the findings of Hily and Seznec [54], Nemirovsky and Yamamoto concluded that in multithreaded processors, cache associativity is important and larger line sizes can degrade cache performance. Their experiments were based on multiprogrammed workloads of up to 4 applications. Each application was individually traced for 5 million cycles to generate a separate address trace. Then to simulate multithreading, these address traces were artificially interleaved in a round-robin fashion. This methodological approach is an approximation of simultaneous multithreading, and is less accurate than an emulation-driven approach. In practice, memory accesses are bursty due to variations in parallelism and other program characteristics. An artificial interleaving cannot capture these dynamic effects.

Yamamoto et al. [148] and Gulati and Bagherzadeh [47] both found that the cache miss rates in simultaneous multithreading processors increased when more threads were used. But neither study characterized the source of interference, nor did they propose techniques for minimizing conflicts.

Other cache interference studies

Several other studies have examined the effects of multithreading on cache performance. These studies, though, examine coarse-grained multithreaded processors, as opposed to SMT processors. Thekkath and Eggers [122] examined the effectiveness of multiple contexts on conventional, coarse-grained multithreaded architectures. They found that cache interference between threads varied depending on the benchmark and that software locality optimizations could minimize interference. For locality-optimized programs, the total number of misses remained fairly constant as the number of contexts was increased. When multiple working sets had to co-exist in the cache, intra-thread misses turned into inter-thread misses. For unoptimized programs, however, misses were more significant and resulted in performance degradations.

Weber and Gupta [142] also studied the effects of cache interference and found increases in inter-thread misses in coarse-grained multithreaded architectures. Agarwal [1] and Saavedra-Barrera et al. [107] used analytic models for studying the efficiency of multithreaded processors. Both models included factors for cache interference which they correlated with the results obtained by Weber and Gupta. None of the studies included the effects of bank contention in their results, since they did not use multiple banks.

Nayfeh and Olukotun [86] investigated the benefits of a shared cluster cache on a single-chip multiprocessor. They found that inter-thread interference can cause degradation for multiprogrammed workloads. For parallel applications, a shared cache could actually obtain superlinear speedups in some cases because of prefetching effects.

6.6 Chapter summary

This chapter explored the behavior of database workloads on simultaneous multithreaded processors, concentrating in particular on the challenges presented to the memory system and the need for carefully managing SMT's thread-shared memory system resources. For our study, we collected traces of the Oracle DBMS executing under Digital Unix on DEC Alpha processors, and processed those traces with simulators of wide-issue superscalar and simultaneous multithreaded processors.

The conclusions from our simulation study are three-fold. First, while database workloads have large footprints, there is substantial reuse, resulting in a small "critical" working set that can fit in the cache, if cache conflicts can be reduced. Second, while cache conflicts are high, particularly in the per-process private data region of the database, they often result from poorly-suited mapping policies in the operating system. By selecting the appropriate mapping policy, and using application-based address offsetting, conflict misses on an 8-context SMT can be reduced to the point where they are roughly commensurate with miss rates for a superscalar. Third, once cache conflicts are reduced to the superscalar level, an SMT processor can achieve substantial performance gains over a superscalar with similar resources. We show a 3-fold improvement in throughput (IPC) for the OLTP workload and a 1.5-fold increase for DSS. These performance gains occur because of SMT's ability to tolerate latencies through fine-grained sharing of all processor resources among the executing threads; this is demonstrated by our detailed measurements indicating that SMT achieves a striking improvement in memory-system utilization and fetch and issue efficiency, when compared with the single-threaded superscalar.

Chapter 7

Software-directed register deallocation

While Chapters 5 and 6 examined resource management techniques to reduce contention and interference, this chapter investigates mechanisms for increasing the *utilization* of shared resources. Specifically, we concentrate on managing the shared register file more efficiently.

As discussed in Chapter 2, the register file in an SMT processor is a primary source of design complexity and introduces a difficult design tradeoff. While large register files are required to service the architectural and renaming register needs of the multiple thread contexts, smaller register files require less chip area and have faster access times. The goal of this chapter is to investigate register management techniques that enable an SMT processor to optimize both die area and performance.

Our techniques address the inefficient use of physical registers in modern out-of-order processors. Although existing register renaming hardware can effectively determine when to allocate a new physical register, the hardware is limited in knowing when physical registers can be deallocated. We propose architectural extensions that permit the software to communicate register last use information to the renaming hardware. This technique, which we call *software-directed register deallocation*, enables the hardware to free dead registers immediately upon their last use, thereby improving register utilization. This improved management of physical registers has two benefits for an SMT processor: (1) it can reduce the number of registers required and improve performance for a given register file size, and (2) it may help reduce the implementation complexity of the register file.

The remainder of this chapter is organized as follows. Section 7.1 again leads off with our experimental methodology. Section 7.2 describes register renaming hardware and provides motivation for our techniques. Section 7.3 describes and evaluates compiler-directed register deallocation mechanisms that identifying register last uses. In Section 7.4, we describe how these techniques can be used by the operating system to

Table 7.1: Configuration and latency parameters of the SMT memory system used in this study.

	L1 I-cache	**L1 D-cache**	**L2 cache**
Size	128 KB / 32 KB	128 KB / 32 KB	16 MB / 2 MB
Associativity	two-way	two-way	direct- mapped
Line size (bytes)	64	64	64
Banks	4	4	1
Accesses/cycle	2	2	1/2
Cache fill time (cycles)	2	2	4
Latency to next level	10	10	68

deallocate registers in idle hardware contexts. We discuss related work in Section 7.5 and summarize in Section 7.6.

7.1 Methodology

For our register file experiments, we again used the compilation and simulation infrastructure described in Chapter 5 to measure the performance of SPECfp95 and SPLASH-2 benchmarks.

Register file management is affected by cache size and memory latencies; smaller caches increase miss rates, and because more latencies have to be hidden, register pressure increases. The opposite is true for larger caches. To investigate this interaction between cache size and register pressure, we experimented with two different memory hierarchies (Table 7.1). The larger memory configuration represents a probable SMT memory hierarchy for machines in production a few years in the future.The smaller configuration serves two purposes: (1) it models today's memory hierarchies, as well as those of tomorrow's low-cost processors, such as multimedia co-processors, and (2) it provides a more appropriate ratio between data set and cache size, modeling programs with larger data sets or data sets with less data locality than those in our benchmarks [110].

We also examined a variety of register file sizes, ranging between 264 and 352, to gauge the sensitivity of the register file management techniques to register size. With more than 352 registers, other processor resources, such as the instruction queues, become performance bottlenecks. At the low end, at least 256 registers are required to hold the architectural registers for all eight contexts,[1] and we provide an additional 8 renaming registers for a total of 264. Smaller register files are attractive for several reasons. First, they have a shorter access time; this advantage could be used either to decrease the cycle time (if register file access is on the critical path) or to eliminate the extra stages we allow for register reading and writing (Section 2.2.2). Second, they take up less area. Register files in current processors occupy a negligible portion (roughly 1%) of the chip area, but a large, multi-ported SMT register file could raise that to around 10%, an area allocation that might not be acceptable [19]. Third, smaller register files consume less power.

Because of the length of the simulations, we limited our detailed simulation results to the parallel computation portion of the applications (the norm for simulating parallel applications). For the initialization phases of the applications, we used a fast simulation mode that warmed the caches, and then turned on the detailed simulation mode once the main computation phases were reached.

7.2 Register renaming

Current out-of-order superscalar processors use register renaming to expose more instruction-level parallelism. Register renaming eliminates false (output and anti-) dependences that are introduced when the compiler's register allocator assigns an arbitrary number of pseudo-registers to the limited number of architectural registers in the instruction set architecture. Dependences are broken by dynamically aliasing each defined architectural register to a *different* physical register, enabling formerly dependent instructions to be executed in parallel.

As briefly discussed in Chapter 2, SMT employs a register mapping scheme (Figure 7.1) similar to that used in the MIPS R10000 [149]. The register renaming hardware is responsible for three primary functions: (1) physical register allocation, (2) register operand renaming, and (3) register deallocation. For a typical instruction that has two register operands and a destination register, renaming hardware will perform each of these three tasks.

Physical register allocation occurs on demand. When an instruction defines an architectural register, the hardware obtains an available physical register from the free register list. If no physical registers are available, instruction fetching stalls. Assuming a register is available, the mapping table is then updated to contain a mapping between the architectural register and the physical register. For *register operand renaming*, the hardware uses the mapping table to find the physical register corresponding to each

1. in the absence of mechanisms to avoid or detect and recover from deadlock.

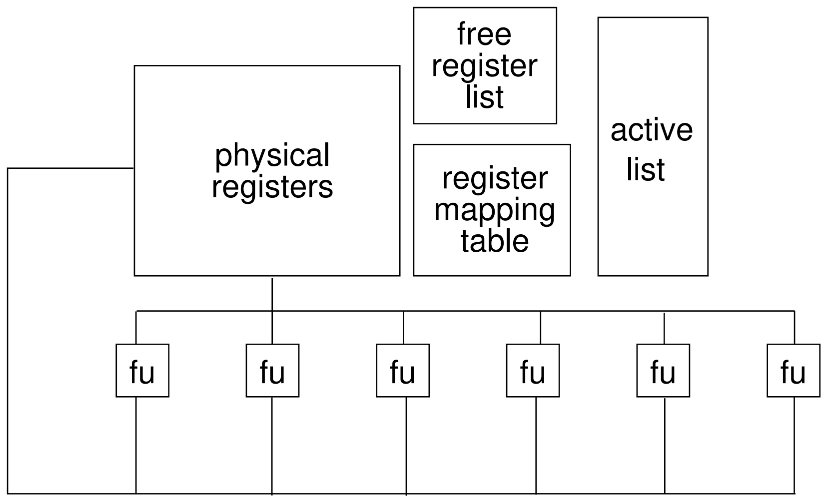


Figure 7.1: Register renaming hardware.

The register file is arranged as a pool of physical registers that are accessed by the functional units (fu). The free register list, register mapping table, and active list contain bookkeeping information to manage the mapping of architectural registers to physical registers.

register operand. After renaming, the processor core refers to the operands by their physical register number, rather than architectural number. An out-of-order processor must keep these architectural-to-physical register mappings consistent in the presence of speculative execution and out-of-order execution. At each unresolved (speculative) branch, the renaming hardware checkpoints the register mapping table. When a branch is resolved, the checkpoint copy is either thrown away (if the branch was predicted correctly) or restored (if the branch was predicted incorrectly).

The final task of the renaming hardware, *register deallocation*, works in conjunction with instruction retirement. An active list keeps track of all uncommitted instructions in per-thread program order. As instructions retire, their physical registers are deallocated and become available for reallocation.

This register mapping hardware handles physical register allocation and renaming rather effectively, but fails to manage deallocation efficiently. In the next two subsections, we will describe two scenarios where an SMT processor could benefit from improved register deallocation.

Code segment	**Register mappings**	**Comments**
1 `ldl` *r20*,`addr1(r22)`	r20->p1	p1 allocated
2 `ldl` *r21*,`addr2(r23)`	r21->p2	
3 `addl r20,r21,`*r21*	r21->p3	p1's last use
...		*dead register distance*
n `ldl` *r20*,`addr4(r29)`	r20->p4	p1 deallocated

Figure 7.2: Register renaming example.

This example illustrates the inability of the renaming hardware to efficiently deallocate the physical register for r20. (The destination registers are italicized). For clarity, Instruction 1 defines r20, creating a mapping to a physical register, say p1. Instruction 3 is the last use of r20. However, p1 cannot be freed until another instruction redefines r20 and then commits (instruction *n* in this example). In the meantime, several instructions and potentially a large number of cycles can pass between the last use of p1 (r20) and its deallocation.

7.2.1 Register renaming hardware inefficiency

Once the last use of a register retires, the register is dead and could be deallocated. For example, as shown in Figure 7.2, register *r20* is dead after instruction *3* commits. The hardware, however, cannot identify the last uses of registers, because it has no knowledge of register lifetimes. Consequently, hardware can only safely deallocate a physical register when it commits another instruction that redefines its associated architectural register, instruction *n* in this example. This conservative behavior leads to inefficient register deallocation.

We use three metrics to quantify this inefficiency and motivate the need for improved register deallocation: (1) the frequency that physical registers are unavailable, (2) the number of registers that are dead each cycle, and (3) the time between the last use of a register and its deallocation, which we call the *dead-register distance*. Register unavailability is the percentage of total execution cycles in which the processor runs out of physical registers (causing fetch stalls); it is a measure of the severity of the problem caused by current hardware register-deallocation mechanisms. The average number of dead registers each cycle indicates how many physical registers could be reused, and

Table 7.2: Register unavailability for the large cache hierarchy. Data indicates the frequency (percentage of total execution cycles) that no registers were available when executing 8 threads. Bold entries (frequencies over 10%) represent severe stalling due to insufficient registers.

Number of registers	applu	hydro2d	swim	tomcatv	fft	LU	water-n
integer registers							
352	3.1	0.1	0.0	0.0	3.8	0.0	0.0
320	3.1	0.7	0.1	0.0	**16.1**	0.1	0.1
288	1.9	1.6	0.4	0.0	**21.5**	0.3	0.3
264	1.5	2.2	**21.9**	0.0	**58.2**	2.2	0.6
FP registers							
352	**27.7**	6.3	0.0	**18.0**	0.0	0.0	0.1
320	**40.2**	**17.7**	0.1	**20.3**	0.0	0.0	0.6
288	**64.0**	**521**	0.5	**58.4**	0.5	0.0	**14.1**
264	**87.8**	**83.0**	0.6	**90.1**	8.1	0.2	**73.9**

thus the potential for a software-based solution. Dead-register distance measures the average number of cycles between the completion of an instruction that last uses a register (instruction 3 from Figure 7.2) and that register's deallocation (the time when instruction n commits); it is a rough estimate of the likely performance gain of a solution.

Using our benchmarks, we measured how frequently physical registers are unavailable. Tables 7.2 and 7.3 indicate that, while the projected SMT design (352 registers) is sufficient for most applications, smaller register files often force instruction fetching to stall because all physical registers are already allocated. (Register pressure was particularly high for integer registers in fft and radix, and for floating-point registers in applu, hydro2d, tomcatv, and water-nsquared). Applications also ran out of registers more frequently with smaller cache hierarchies.

Although one solution is to simply design larger register files, a closer examination reveals that more efficient register deallocation could address the problem as well. In all cases where register-induced stalling was substantial (bold entries in Tables 7.2 and 7.3), a huge number of registers were also dead, as shown in Tables 7.4 and 7.5. Furthermore, these dead registers have been dead a long time, as indicated by the average dead register

Table 7.3: Register unavailability for the small cache hierarchy.

Number of registers	applu	hydro2d	swim	tomcatv	fft	LU	water-nsquared
Integer registers							
352	3.4	1.0	0.4	0.0	8.4	1.3	0.0
320	3.5	1.5	1.5	0.0	**16.2**	3.3	0.2
288	2.6	2.1	5.3	0.0	**25.2**	**10.5**	0.3
264	2.2	2.5	**44.2**	0.2	**64.9**	**22.7**	0.7
FP registers							
352	**32.3**	**34.4**	5.3	**45.7**	0.5	1.7	0.1
320	**44.6**	**48.0**	8.6	**60.2**	1.4	3.5	0.6
288	**67.5**	**70.1**	**13.2**	**79.8**	5.3	5.5	**17.0**
264	**88.1**	**87.2**	2.7	**94.0**	8.5	6.7	**74.7**

Table 7.4: Average number of dead registers per cycle when executing 8 threads for the large cache hierarchy. Bold entries are those where no registers were available more than 10% of execution cycles.

Number of registers	applu	hydro2d	swim	tomcatv	fft	LU	water-n
Integer registers							
352	74	69	29	52	135	110	168
320	74	68	29	53	**134**	110	168
288	75	64	29	52	**132**	110	168
264	77	57	**29**	51	**140**	110	168
FP registers							
352	**120**	153	73	**136**	68	42	199
320	**120**	**153**	73	**136**	68	42	199
288	**117**	**152**	73	**135**	67	42	**199**
264	**111**	**150**	73	**133**	71	42	**198**

Table 7.5: Average number of dead registers per cycle when executing 8 threads for the small cache hierarchy.

Number of registers	applu	hydro2d	swim	tomcatv	fft	LU	water-n
Integer registers							
352	74	62	30	54	132	147	168
320	74	61	30	55	**132**	147	167
288	74	58	30	55	**131**	**110**	168
264	76	54	**32**	54	**137**	**110**	168
FP registers							
352	**121**	**158**	77	**141**	68	88	199
320	**121**	**157**	76	**141**	68	81	216
288	**118**	**155**	**76**	**139**	69	50	**199**
264	**113**	**152**	76	**136**	72	50	**198**

Table 7.6: Dead register distance for 264 registers and the smaller cache hierarchy. The data indicate that registers are frequently not deallocated until many cycles after their last use has retired. Figures for other register sizes were similar. Bold entries are those where no registers were available more than 10% of execution cycles.

Benchmark	Int instructions	Int cycles	FP instructions	FP cycles
applu	57.6	214.6	**18.4**	**97.1**
hydro2d	59.1	155.4	**30.9**	**157.4**
swim	32.3	27.8	11.7	28.4
tomcatv	67.2	225.7	**22.6**	**120.0**
fft	**30.7**	**89.9**	20.4	65.7
LU	**56.9**	**85.6**	7.1	22.4
water-nsquared	32.7	215.4	**32.7**	**133.7**
Average	47.2	125.5	18.5	81.8

distances shown in Table 7.6. The data reveal that these dead registers could have been freed more promptly and therefore reallocated many instructions/cycles earlier. All this suggests that, if registers were managed more efficiently, performance could be recouped and even a 264-register file might be sufficient for avoiding register-induced stalls.

7.2.2 Register inefficiencies in SMT processors

In multithreaded processors, a second source of inefficient register utilization can also occur. This inefficiency results from the logical partitioning of registers to (per-thread) hardware contexts.

In fine-grained multithreaded architectures like the Tera [5], each hardware context includes a register file for one thread, and a thread only accesses registers from its own context, as shown in Figure 7.3a.[2] In contrast, in an SMT processor, a single register file can be shared among all contexts (Figure 7.3b). The SMT register file is structured as a single pool of physical registers and holds the state of all resident threads. Threads name their private architectural registers, and the renaming hardware maps these thread-private, architectural registers to the pool of thread-independent physical registers. Register renaming thus provides a transparent mechanism for dynamically sharing the register pool.

Although an SMT processor is best utilized when all hardware contexts are busy, some contexts may occasionally be idle. To maximize performance, no physical registers should be allocated to idle contexts; instead, all physical registers should be shared by the active threads. However, with existing register deallocation schemes, when a thread terminates, its architectural registers remain allocated in the processor until they are redefined by a new thread executing in the context. Consequently, the register file behaves more like a partitioned file (Figure 7.4), which is very inefficient. In an ISA with 32 architectural registers, 32 physical registers must be dedicated to each context to back the architectural state. This means that on an eight-context SMT with 352 registers, only 96 (352-8*32) physical registers are available for sharing among the active threads, even if several hardware contexts are idle. To increase register utilization, we use software-directed register deallocation to deallocate "idle" physical registers so that they are available for active contexts.

To address the two sources of register inefficiency, we propose software-directed register deallocation mechanisms. These mechanisms allow the compiler and operating

---

[2] Note that we are discussing different logical organizations for the register file. How the file is physically structured is a separate issue.

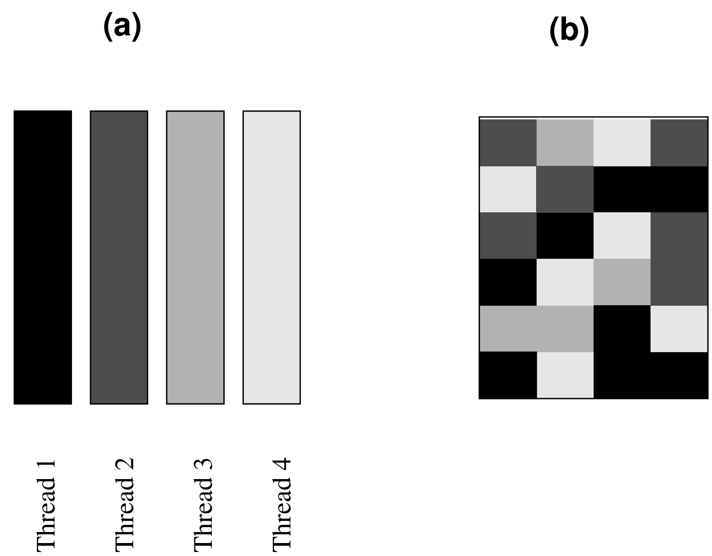


Figure 7.3: Logical register file configurations.

(a) is a Tera-style, partitioned register file; (b) is an SMT register file in which all threads share a common pool of physical registers.

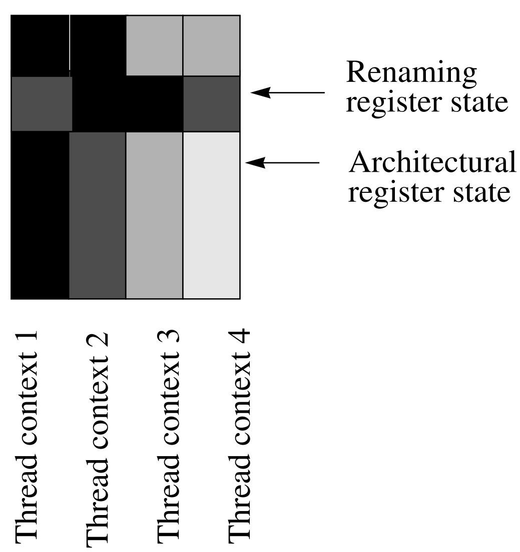


Figure 7.4: Register file utilization with idle contexts.

Although the register file is organized as a shared physical pool, idle contexts still consume physical registers. With current renaming hardware, physical registers for idle contexts are not deallocated. In this example, thread context 4 is idle, but a set of physical registers are still allocated to back the architectural state of thread.

system to assist the renaming hardware with register deallocation, by identifying dead registers that can be aggressively deallocated.

7.3 Compiler support for dead-register deallocation

As described in Section 7.2.1, renaming hardware only has knowledge of a register's redefinition, not its last use. Although the compiler can identify the last use of a register value using standard dataflow analysis, the compiler currently has no means for communicating this information to the hardware.

We propose five mechanisms that allow the compiler to convey register last-use information to the hardware, and show that they improve register utilization on SMT processors. The proposed mechanisms are either new instructions or fields in existing instructions that direct the renaming hardware to free physical registers.

7.3.1 Five compiler-based solutions

Free Register Bit communicates last-use information to the hardware via dedicated instruction bits, with the dual benefits of immediately identifying last uses and requiring no instruction overhead. The physical registers corresponding to these last uses can be deallocated by the hardware when the instruction commits. Although this scheme is unlikely to be implemented, because most instruction sets do not have two unused bits, it can serve as an upper bound on performance improvements that can be attained with the compiler's static last-use information. To simulate Free Register Bit, we modified the Multiflow compiler to generate a table, indexed by the PC, that contains flags indicating whether either of an instruction's register operands were last uses. On each simulated instruction, the simulator performed a lookup in this table to determine whether register deallocation should occur when the instruction is retired.

Free Register is a more realistic implementation of Free Register Bit. Rather than specifying last uses in the instruction itself, it uses a separate instruction to specify one or two registers to be freed. Figure 7.5b presents an example code sequence that uses the Free Register instruction to free integer registers 20-25.Our compiler generates a Free Register instruction (an unused opcode in the Alpha ISA) immediately after any instruction containing a last register use (if the register is not also redefined by the same instruction). Like Free Register Bit, it

a) base

```
ldl    r20,addr1(r22)
ldl    r21,addr2(r23)
addl   r20,r21,r24
addl   r21,0x1,r21
       ...
stl    r12,addr3(r21)
ldl    r20,addr4(r25)
```

b) Free Register

```
ldl    r20,addr1(r22)
freg   r22
ldl    r21,addr2(r23)
freg   r23
addl   r20,r21,r24
freg   r20
addl   r21,0x1,r21
       ...
stl    r12,addr3(r21)
freg   r12,r21
ldl    r20,addr4(r25)
freg   r25
```

(c) Free Mask

```
ldl   r20,addr1(r22)
ldl   r21,addr2(r23)
addl  r20,r21,r24
addl  r21,0x1,r21
      ...
stl   r12,addr3(r21)
ldl   r20,addr4(r25)
lda   r25,0x1000(r31)    ; mask for reg 12
ldah  r25,0x02f0(r25)    ; mask for regs 20,21,22,23,25
fmask r25                ; frees integer regs
                         ; specified by mask in r25
```

Figure 7.5: Register deallocation example.

These code fragments illustrate the register freeing mechanisms: a) is the original code fragment; b) shows the Free Register instructions necessary to free registers r12, r20, r21, r22, r23, and r25, c) is the Free Mask instructions necessary to free the same registers.

frees registers as soon as possible, but with an additional cost in dynamic instruction overhead.

Free Mask is an instruction that can free up to 32 registers, and is used to deallocate dead registers over a large sequence of code, such as a basic block or a set of basic blocks. Free Mask sacrifices the promptness of Free Register's deallocation for a reduction in instruction overhead, as shown in Figure 7.5. For our experiments, we inserted a Free Mask instruction at the end of each Multiflow trace. Rather than identifying dead registers in operand specifiers, the compiler generates a bit mask. In our particular implementation, the Free Mask instruction uses the lower 32-bits of a register as a mask to indicate which registers can be deallocated. The mask is generated and loaded into the register using a pair of lda and ldah instructions, each of which has a 16-bit immediate field.

Free Opcode is motivated by our observation that 10 opcodes were responsible for 70% of the dynamic instructions with last use bits set, indicating that most of the benefit of Free Register Bit could be obtained by providing special versions of those opcodes**.** In addition to executing their normal operation, the new instructions also specify that either the first, second, or both operands are last uses. In this study, we use the 15 opcodes listed in Table 7.7, obtained by profiling Free Register Bit instruction frequencies on applu, hydro2d and tomcatv.[3] Retrofitting these 15 instructions into an existing ISA should be feasible; for example, all can be added to the DEC Alpha ISA, without negatively impacting instruction decoding.

Free Opcode/Mask augments Free Opcode by generating a Free Mask instruction at the end of each trace. The use of a mask instruction addresses one of the potential shortcomings of the Free Opcode scheme; namely, that our opcode selection was based on a profile of a very small same of programs, This hybrid scheme addresses register last uses in instructions that are not covered by our particular choice of instructions for Free Opcode.

3. We experimented with between 10 and 22 Free Opcode instructions. The additional opcodes after the top 15 tended to occur frequently in only one or two applications, and using them brought limited additional benefits (exceptions were swim and radix).

Table 7.7: The opcodes used in Free Opcode. Note that for mult, stt, and fcmov, two new versions of each must be added. The versions specify whether the first, second, or both operands are last uses.

Integer		FP	
Opcode	**Operand**	**Opcode**	**Operand**
`addl`	1	`addt`	1
`subl`	1	`subt`	1
`mull`	1	`mult`	1, 2
`stl`	2	`stt`	1, both
`beq`	1	`fcmov`	1, both
`lda`	1		
`ldl`	1		

7.3.2 Implementation issues

Using the Free Register instruction as an example, Figure 7.6 illustrates how renaming hardware must be modified to provide support for our register deallocation mechanisms. To support register deallocation, we add a valid bit to each entry in the logical-to-physical register mapping table. We also must modify the renaming logic that occurs in the renaming and commit pipeline stages.

Free Register instruction *n* (the `freg r20` instruction taken from Figure 7.5b) specifies that architectural register *r20* is dead. When it enters the renaming stage, the renaming hardware determines which physical register will be deallocated by performing a lookup in the register mapping table. The hardware finds the mapping *r20->p1* and clears the valid bit for *r20*'s entry, specifying that *r20* is now dead. *p1* is then placed in *n*'s active list entry.

When *n* commits, the renaming hardware looks at each instruction's active list to see if a physical register should be deallocated. For instruction *n*, the *p4* is deallocated and returned to the free register list. Note that the deallocation process is the same as with "normal" instructions, except that the hardware may need to perform more deallocations in a given cycle. Current renaming hardware already can perform many deallocations each cycle. For example, the Alpha 21264 deallocates up to 13 registers each cycle to handle multiple instruction retirement or squashing. All five proposed register deallocation techniques use a similar mechanism. Free Mask is slightly more complex, because it can specify up to 32 registers; in this case deallocation could take multiple

a) Free register example

iInstruction number	Instruction	
m	`addl`	`r20,r21,r24`
n	`freg`	`r20`

b) Renaming hardware after addl is renamed, but before freg is renamed

register mapping table

r20	p1	1
r21	p4	1
r24	p5	1

register free list

p0
p3

active list entries

no.	src reg1	src reg2	dest reg	prev dest	...
m	p1	p4	p5	p2	
n	--	--	--	--	

c) Renaming hardware after freg is renamed

register mapping table

r20	p1	0
r21	p4	**1**
r24	p5	1

register free list

p0
p3

active list entries

no.	src reg1	src reg2	dest reg	prev dest	...
m	p1	p4	p5	p2	
n	**p1**	--	--	**p1**	

d) Renaming hardware after freg commits

register mapping table

r20	p1	0
r21	p4	1
r24	p5	1

register free list

p0
p3
p2
p1

active list entries

no.	src reg1	src reg2	dest reg	prev dest	...
m	--	--	--	--	
n	**--**	--	--	**--**	

Figure 7.6: Register deallocation implementation example.

cycles if necessary. (In our experiments, however, only 7.2 registers, on average, were freed by each mask.)

7.3.3 Experimental results

The five register deallocation schemes are compared in Figures 7.7 (large cache hierarchy) and 7.8 (small cache hierarchy), which charts their speedup versus no explicit register deallocation. While the Free Register Bit results show that compiler-directed register deallocation may have little or no benefit for large SMT register files, it can (potentially) improve performance significantly for small register files (95% on average, but ranging as high as 195%). This performance gain enables an SMT processor with a small register file to achieve large register file performance, as shown in Figures 7.9 and 7.10. The significance of this becomes apparent in the context of conventional register file design. Single-threaded, out-of-order processors often double their registers to support greater degrees of parallelism (e.g., the R10000 has 64 physical registers, the 21264 has 80). An SMT processor, on the other hand, need not double its architectural registers if the physical registers are effectively utilized. Our results show that an 8-context SMT with compiler-directed register deallocation only needs 8 additional physical registers in addition to the 256 (8 contexts * 32 architectural registers/context), or 3% of the architectural register state.

The Free Register and Free Mask results highlight the trade-off between these two alternative schemes. Free Register is more effective at reducing the number of dead registers, because it deallocates them more promptly, at their last uses. When registers are a severe bottleneck, as in applu, hydro2d, tomcatv (small cache only), and radix with small register files, Free Register outperforms Free Register Mask. Free Register Mask, on the other hand, incurs less instruction overhead; therefore it is preferable with larger register files and applications with lower register usage.

Free Opcode and its variant, Free Opcode/Mask,[4] are the schemes of choice. They strike a balance between Free Register and Free Mask by promptly deallocating registers, while avoiding instruction overhead. When registers were at a premium, Free Opcode(/Mask) achieved or exceeded the performance of Free Register; with the larger

---

4. We profiled a very small sample of programs to determine the best selection of opcodes for Free Opcode, and used Free Opcode/Mask to provide more flexibility in opcode choice. The speed-ups of the two schemes are very close, and which has the performance edge varies across the applications for 264 registers. Their relative performance also may change if a different set of programs is used to determine the hot opcodes. For example, by adding 6 single-precision floating point Free Opcodes to the single-precision swim, Free Opcode exceeded both Free Register and Free Mask.) Therefore we discuss Free Opcode and Free Opcode/Mask results together.

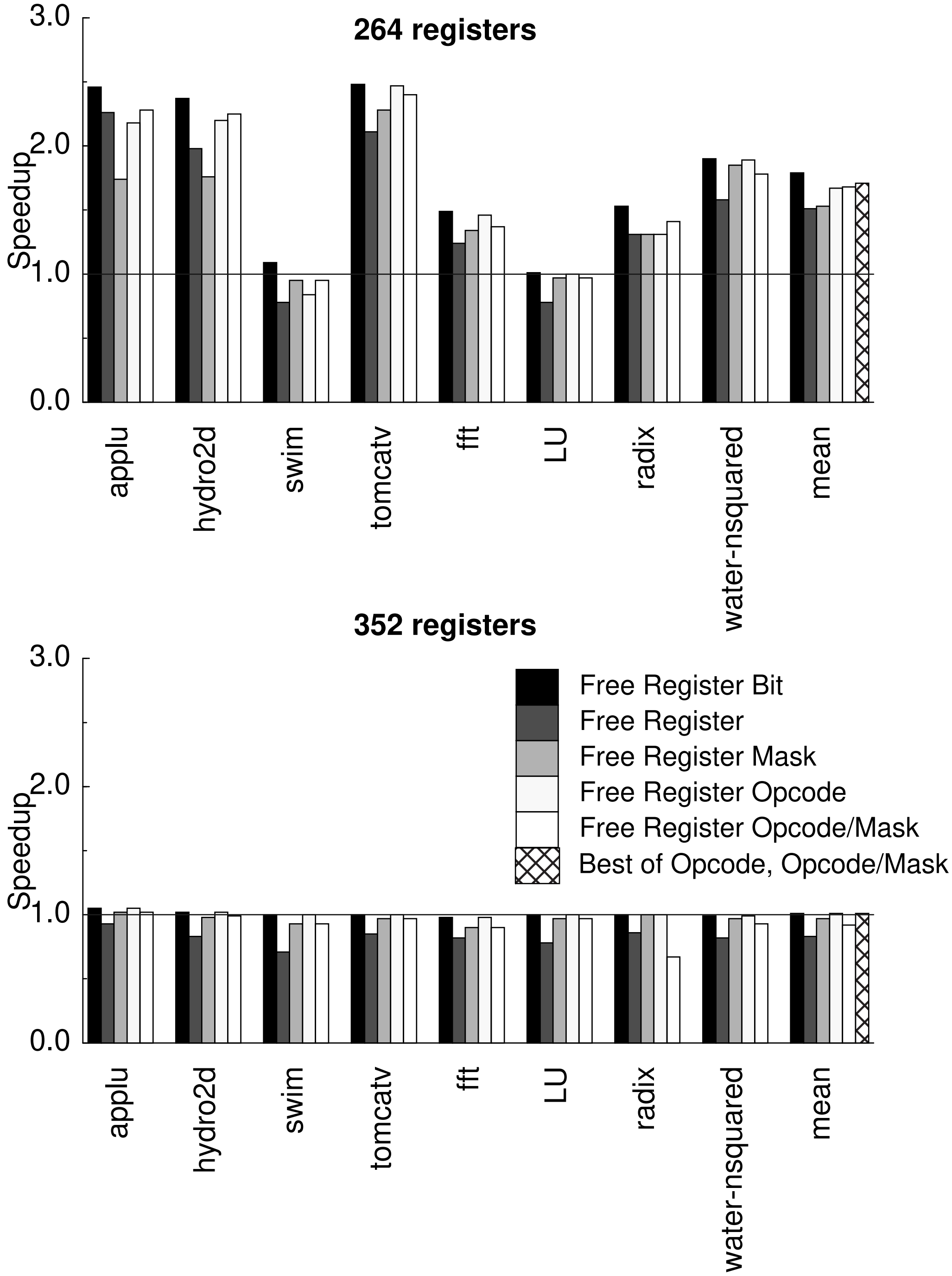


Figure 7.7: Register deallocation performance for the large cache size.

Each bar is the speedup over no deallocation with 8 threads. For each individual benchmark, we also took the better of Free Register Opcode and Free Register Opcode/Mask, and then took the mean of these results across the workload (shown by the Best of Opcode, Opcode/Mask bar).

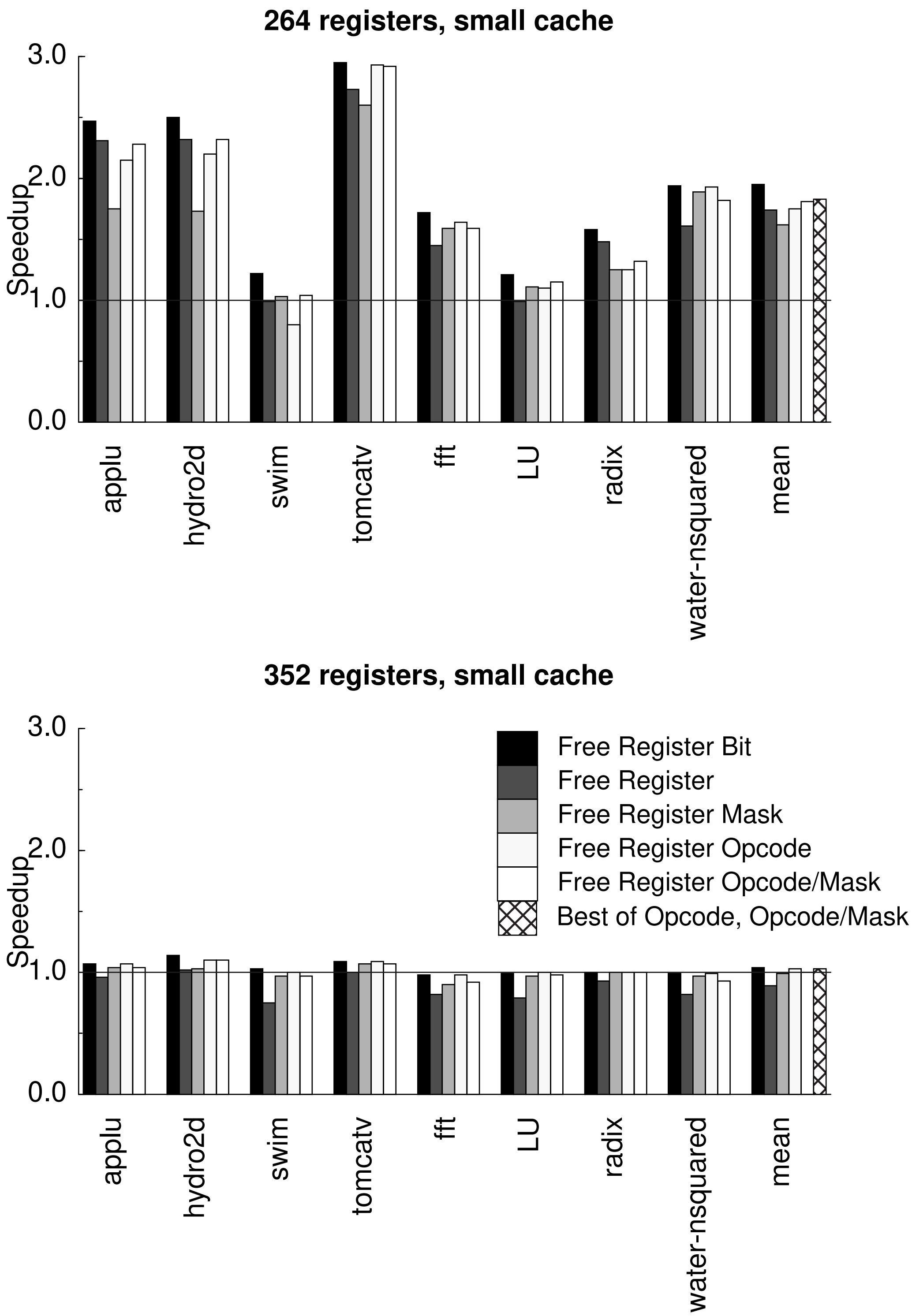


Figure 7.8: Register deallocation performance for the small cache size.

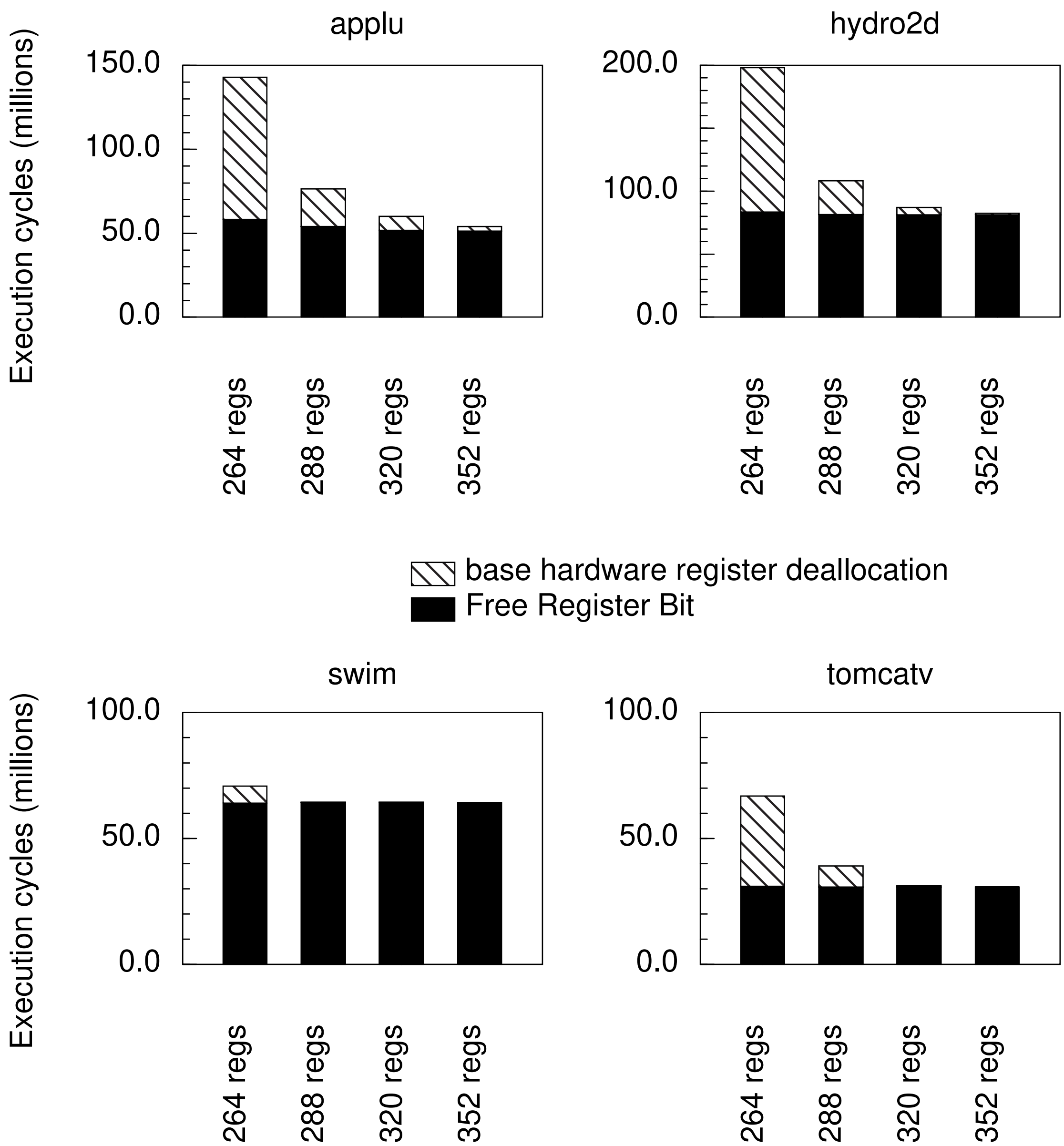


Figure 7.9: Comparison of execution time with and without Free Register Bit for a range of register file sizes, on the large cache size.

The height of the solid black bar represents the execution time when Free Register Bit is used. The total height of the bar corresponds to the execution time when no deallocation is performed. The relatively flat height of the black bars indicates that with Free Register Bit, smaller register files can achieve the performance of larger register files.

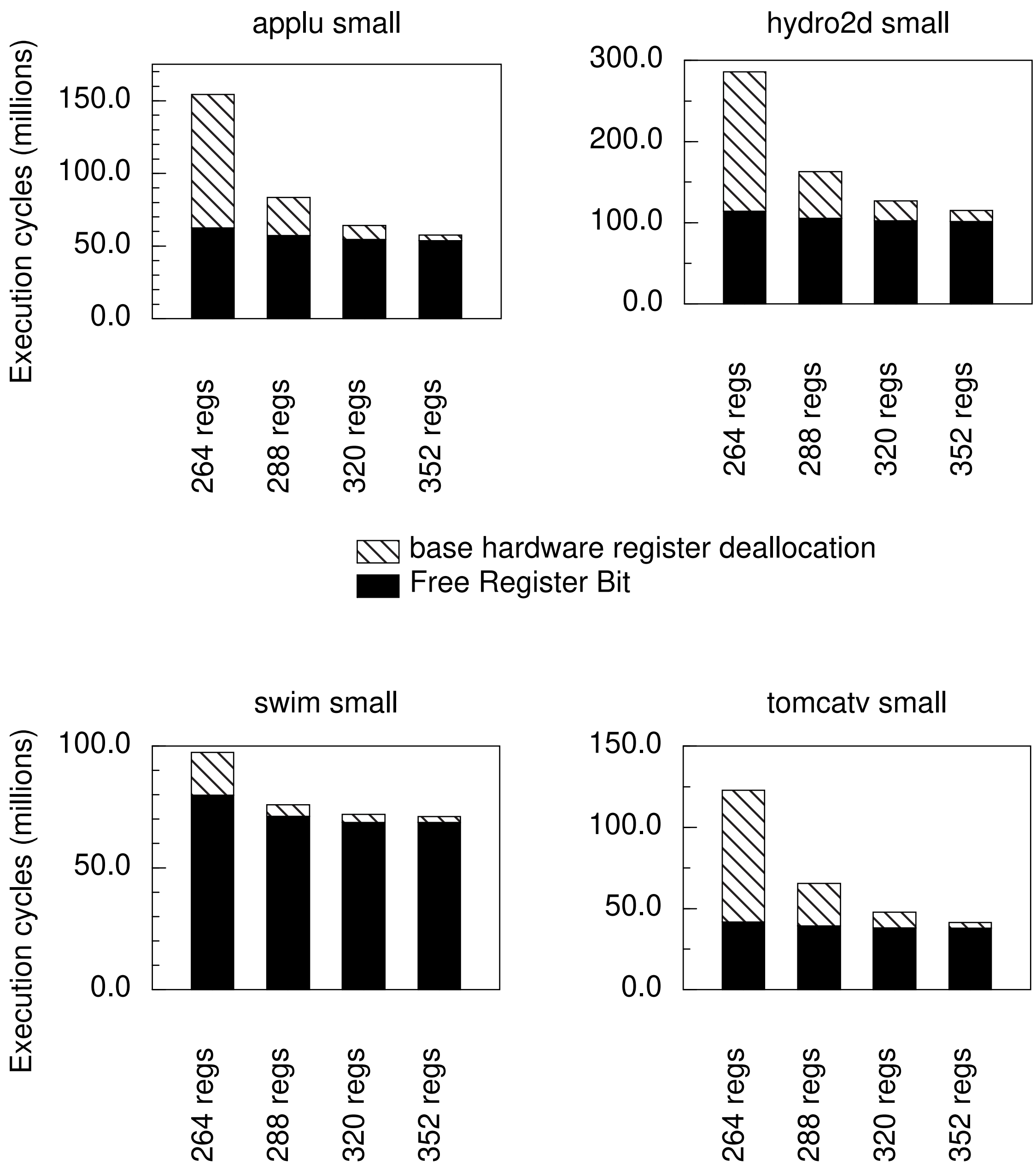


Figure 7.10: Comparison of execution time with and without Free Register Bit for a range of register file sizes for the small cache size.

register file and for applications with low register usage, Free Mask performance was attained or surpassed.

For most programs (all register set sizes and both cache hierarchies) Free Opcode(/Mask) met or came close to the optimal performance of Free Register Bit (see the Best of Opcode, Opcode/Mask bar in the figures). For example, it was within 1% on average for 264 registers, and 7% for 352, on the small cache hierarchy.

We expect that Free Opcode(/Mask) performance could be improved by tuning opcode selection further. Recall that our opcodes were selected by profiling a subset of the applications. By looking at a different or larger set of programs, we might identify different "hot" opcodes that could improve register deallocation. For example, swim has many dead registers created by the execution of single-precision FP instructions. If we had added six additional opcodes to target these dead registers, the Free Opcode scheme outperformed both Free Register and Free Mask on this application.

Furthermore, with the use of other hybrid schemes (perhaps judiciously combining Free Opcode, Free Mask, and Free Register), we expect that we could narrow the gap between a hybrid scheme and Free Register Bit even further, and that we will achieve the upper bound of compiler-directed register deallocation performance.

In summary, by providing the hardware with explicit information about register lifetimes, compiler-directed register deallocation can significantly improve performance on small SMT register files, so that they become a viable alternative even with register-intensive applications. Although particularly well-suited for SMT, register deallocation should benefit any out-of-order processor with explicit register renaming.

7.4 Operating system support for dead-register deallocation

The compiler is not the only piece of software that can utilize software-directed register deallocation. In a simultaneous multithreaded processor, the operating system can also leverage Free Register, Free Register Bit, or Free Mask to deallocate physical registers more aggressively when threads terminate.

When a thread finishes executing, the operating system decides what to schedule on the newly-available hardware context. There are three options, each of which has a different implication for register deallocation:

1. **Idle contexts**: If there are no new threads to run, the context will remain idle. The terminated thread's physical registers could be deallocated, so that they become available to active threads.
2. **Switching to a new thread**: Physical registers for a new thread's architectural

registers are normally allocated when it begins execution. A more efficient scheme would free the terminated threads's physical registers, allocating physical registers to the new thread on demand. Unallocated physical registers would then be available to other contexts.

3. **Switching to a swapped-out thread**: Context switch code loads the register state of the new thread. As these load instructions retire, physical registers used by the terminated thread are deallocated.

All three scenarios present an opportunity to deallocate a terminated thread's physical registers early. The operating system could use the Free Register Mask scheme from the previous section to explicitly deallocate all architectural registers. Alternatively, the operating system scheduler could execute a privileged, *context deallocation instruction (CDI)* in the context of the terminated thread. This instruction would trigger physical register deallocation for a thread, with the renaming hardware deallocating the terminating thread's physical registers once this instruction retires.

7.4.1 Implementation issues

The CDI instruction is essentially like a special case of Free Register Mask, where all bits in the mask are set. The CDI, though, has the advantage of requiring less instruction overhead, because we do not need to load the mask and we can deallocate integer and FP registers in the same instruction.

To implement a context deallocation instruction, three tasks must be performed: creating a new map table, invalidating the context's register mappings, and returning the registers to the free list. When a CDI enters the pipeline, the current map table is saved and a new map table with no valid entries is created; the saved map table identifies the physical registers that should be deallocated, while the new table will hold subsequent register mappings. Once the CDI retires, the saved map is traversed, and all mapped physical registers are returned to the free list. Finally, all entries in the saved map are invalidated. If the CDI is executed on a mispredicted path and consequently gets squashed, both the new and saved map tables are thrown away.

Much of the hardware required for these three tasks already exists to handle speculative execution in out-of-order processors with register mapping. When a branch enters the pipeline, a copy of the map table is created; when the branch is resolved, one of the map tables is invalidated, depending on whether the speculation was correct. If instructions must be squashed, the renaming hardware traverses the active list (or some other structure that identifies physical registers) to determine which physical registers should be returned to the free list. Although the CDI adds a small amount of logic to existing renaming hardware, it improves the efficiency of the SMT register file by deallocating registers more promptly.

7.4.2 Experimental results

To evaluate the performance of the CDI technique, we varied the number of active threads and register set sizes, and compared performance when idle registers were and were not deallocated. We modeled an OS scheduler that frees all physical registers for terminated threads, by making all physical registers available when a parallel application began execution.

The results of this comparison are shown in Figure 7.11. With no deallocation (Figure 7.11a), only renaming registers are shared among threads. Execution time therefore was greater for smaller register files, as more threads competed for fewer registers. With the CDI, shown in Figure 7.11b, performance was less sensitive to both parameters. In fact, the smaller register files had the same performance as larger ones when few threads were executing, because registers were not tied up by idle contexts. Except for the smallest configuration, CDI performance was stable with varying numbers of threads, because the parallelism provided by additional threads overcame the increased competition for registers; only the 264-register file had a performance sweet spot.

The speedups in Figure 7.12 show that the deallocating idle registers is beneficial for all register file sizes and numbers of threads. It provides the greatest benefits when there are more registers to share (several idle contexts) and when the register file is smaller. For example, with 320 registers and 4 idle contexts (4 threads), the performance gain was 8%, averaged over all applications. With only 288 or 264 registers, the advantage grew to 34% and 205%, and with 6 idle contexts (and 320 registers) to 15%. Taking both factors into account (288/264 registers, 6 idle contexts), idle context register-deallocation outperformed the base case by 51%/232%. Only when all contexts were active was this technique of no benefit; in this case the architectural state for all threads is resident in both schemes.

Idle context register-deallocation has a larger performance edge with smaller, less aggressive cache hierarchies, because hiding the longer memory latencies requires more in-flight instructions, and therefore more outstanding registers. This suggests that efficient register management is particularly important on memory-intensive workloads or applications with relatively poor data locality.

In summary, the results illustrate that partitioning a multithreaded register file

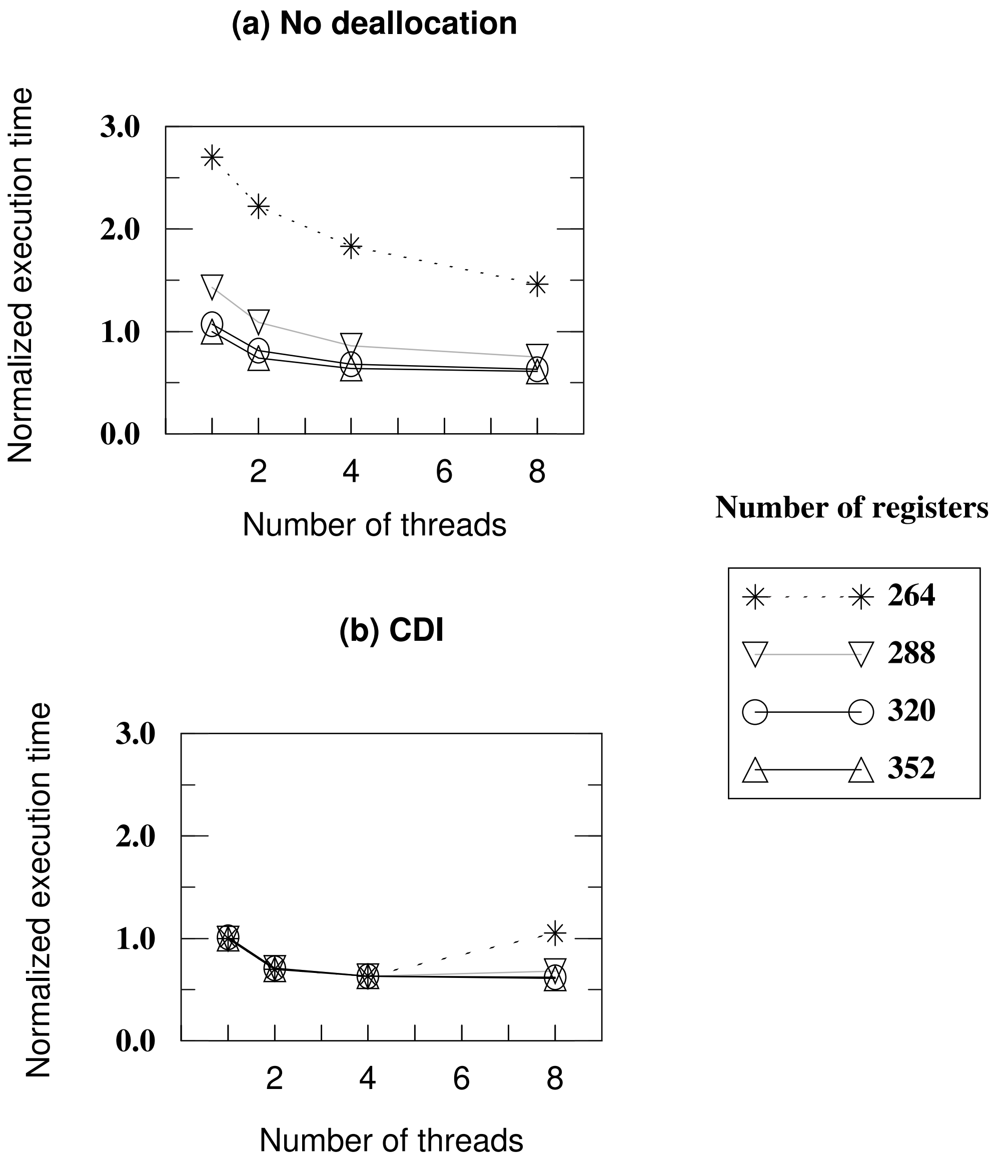


Figure 7.11: Execution time for the larger memory hierarchy.

This figure compares the base hardware (no deallocation) with idle-context register-deallocation. For each graph, execution time is normalized to 352-register, single-thread execution. Results with the smaller SMT memory hierarchy had identical trends.

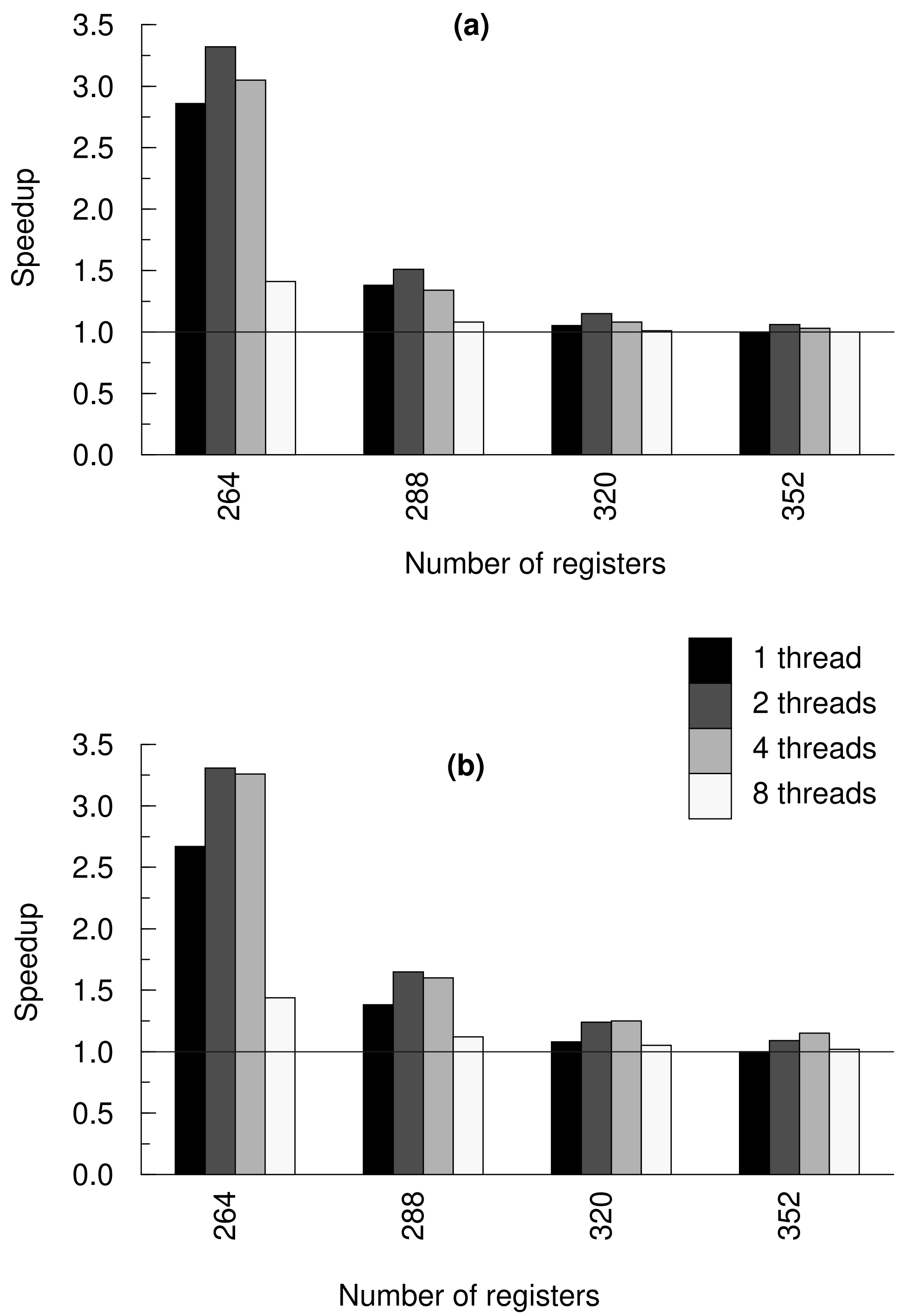


Figure 7.12: Speedups when freeing registers in idle contexts for the larger (a) and smaller (b) memory hierarchies, at different register file sizes.

restricts its ability to expose parallelism. Operating system support for deallocating registers in idle contexts, which enables the register file to be fully shared across all threads both improves performance, and makes it less dependent on the size of the register file and the number of active threads.

7.5 Related work

Several researchers have investigated register file issues similar to those discussed in this paper. Large register files are a concern for both multithreaded architectures and processors with register windows. Waldspurger and Weihl [140] proposed compiler and runtime support for managing multiple register sets in a register file. The compiler tries to identify an optimal number of registers for each thread, and generates code using that number. The runtime system then tries to dynamically pack the register sets from all active threads into the register file. Nuth and Dally's [92] named state register file caches register values by dynamically mapping active registers to a small, fast set of registers, while backing the full register name space in memory.

To reduce the required chip area in processors with register windows, Sun designed 3-D register files [130]. Because only one register window can be active at any time, the density of the register file can be increased by overlaying multiple register cells so that they share wires.

The issue of register lifetimes and last uses has also been discussed in several papers. Farkas et al. [34] compared the register file requirements for precise and imprecise interrupts and their effects on the number of registers needed to support parallelism in an out-of-order processor. They also characterized the lifetime of register values, by identifying the number of live register values present in various stages of the renaming process.

Martin et al. [82] communicate dead-register information through a mechanism very similar to our free-register instruction. The focus of that paper is to use the information to dynamically eliminate unnecessary load and store instructions, particularly around procedure calls, but they do note its advantages in physical register-size reduction, even for the single-thread architectures they consider. We consider more effective mechanisms for communicating live ranges, and examine both single-thread and multithreaded scenarios.

Franklin and Sohi [38] and Lozano and Gao [80] found that register values have short lifetimes, and often do not need to be committed to the register file. Both proposed compiler support to identify last uses and architectural mechanisms to allow the hardware to ignore writes to reduce register file traffic and the number of write ports, but neither applied these concepts to register deallocation. Pleszkun and Sohi [100] proposed a mechanism for exposing the reorder buffer to the compiler, so that it could generate better schedules and provide speculative execution. Sprangle and Patt [115] proposed a statically-defined tag ISA that exposes register renaming to the compiler and relies on basic blocks as the atomic units of work. Part of the register file is used for storing basic block effects, and the rest handles values that are live across basic block boundaries.

Gonzalez et al. [43] also identify the need for more efficient usage of physical registers. They propose a virtual-physical register scheme that decouples the two functions of physical registers: storage allocation and identifying data dependences. Virtual-physical registers are used to handle the data dependences, but physical registers are not allocated until later in the pipeline, when there actually is a value to be stored. Their study reaffirms the observations made in this chapter regarding register inefficiencies. Their techniques, however, are independent of ours and could be applied in conjunction to improve register utilization both at the front-end and the back-end of the processor pipeline.

Several researchers have investigated the benefits of partitioned register files. Multiflow [22], Janssen and Corporaal [60] and Capitanio et al. [14] partitioned register files to reduce chip area and implementation complexity. The compiler's instruction scheduler was responsible for managing the partitions, and could reduce the performance degradation. Llosa et al. [75] proposed the non-consistent dual register file, which partitions the register file so that each partition contains its own independent local register values. However, some "global" values are stored in both partitions.

Each of these partitioned register files are designed for single-threaded architectures, in which all registers are still available to the executing application. On an SMT processor, however, a partitioned file would severely limit the dynamic scheduling of multiple threads, thus degrading performance, as shown in this chapter.

Finally, Kiyohara et al. [67] proposed the use of new opcodes to address an extended register file, thus increasing the effective number of physical registers available to an application. Their register connection scheme used a register mapping table similar to the register renaming hardware provided by SMT and out-of-order processors. Our software-directed register deallocation techniques could also be applied to improve performance of a processor with register connection.

7.6 Chapter summary

In this chapter, we have proposed new software-directed techniques that permit a reduction in register file size without sacrificing performance. Our software-directed register deallocation mechanisms allow the hardware and software to cooperate and manage a shared register file more efficiently, thus increasing utilization of the registers. Fundamental to these techniques is the global sharing of registers among threads, both for architectural register and renaming register needs. By introducing new instructions or additional fields in the ISA, we allow the operating system and compiler to signal physical register deallocation to the processor, thereby greatly decreasing register waste. The result is more effective register use, permitting either a reduction in register file size or an increase in performance for a given file size.

We have introduced explicit software-directed deallocation in two situations. First, by allowing the compiler to signal the last use of a register, the processor need not wait for a redefinition of that register in order to reuse it. We proposed several mechanisms for signalling last register use, and showed that on small register files, average speedups of 1.6 can be obtained by using the most efficient of these mechanisms. Second, when a context becomes idle, the operating system can indicate that the idle context's physical registers can be deallocated. This permits those registers to be freed in order to serve the renaming needs of other executing threads. Our results show that such notification can significantly boost performance for the remaining threads, e.g., a register file with 264 registers demonstrates performance equivalent to a 352-register file when only 4 threads are running.

Chapter 8

Conclusions

Simultaneous multithreaded processors allow multiple threads to issue multiple instructions to a processor's functional units in a single cycle. In contrast to other proposed next-generation architectures, SMT processors enable threads to share functional units and other processor resources at a much finer granularity, thereby achieving substantial increases in processor utilization and performance.

An often-cited criticism of SMT and other multithreaded architectures, however, is that multithreading can introduce interference or conflicts in shared structures, such as the cache. If too severe, these conflicts may negate the benefits of exploiting thread-level parallelism.

This dissertation has explored the implications of dynamic, inter-thread resource sharing on SMT processors. We have examined three primary issues: the performance benefits of resource sharing, the potential costs of inter-thread sharing, and techniques for utilizing shared resources efficiently.

To quantitatively demonstrate the benefits of dynamic resource sharing, we compared simultaneous multithreading to a single-chip multiprocessor with comparable execution resources. These experiments showed that SMT processors can achieve a 54% speedup over a single-chip MP with similar execution resources. Multiprocessors frequently suffer from resource waste because resources must be statically partitioned among CPUs. In contrast, simultaneous multithreading dynamically shares processor structures between threads, greatly increasing utilization. This flexible architecture enables a processor to use instruction-level and thread-level parallelism interchangeably and adapt as parallelism varies across a workload.

This dissertation has also explored the effects of thread-level parallelism on shared processor structures. Although thread-level parallelism can place greater stress on shared resources, we have found that inter-thread interference frequently results from software that is unaware of the underlying SMT architecture. By adopting the proper software policies, however, threads can utilize shared resources in an SMT-conscious manner, minimizing or even eliminating interference. This dissertation examined several of these techniques.

We presented an SMT compiler strategy that tunes several existing optimizations to simultaneous multithreaded architectures. Many compiler optimizations, such as loop iteration scheduling, software speculative execution, and loop tiling, must be modified on an SMT processor, because the optimizations depend on the characteristics of the underlying architecture. Our SMT-specific policies recognize the two primary characteristics that distinguish SMT processors from conventional superscalars and shared-memory multiprocessors: a shared memory subsystem and shared functional units. By taking these two characteristics into consideration, our SMT-specific optimization policies enable threads to share resources more effectively, thereby heightening SMT performance.

We also described operating system and application-level techniques for minimizing cache interference in memory-intensive commercial database workloads. Our analysis reveals that although database workloads have very large memory footprints, the critical working sets of database applications are often cacheable. As a result, simultaneous multithreading need not cause inter-thread cache conflict misses as long as memory addresses are managed carefully. When using a bin hopping page mapping policy and an application-level address offsetting technique, we measured speedups of 3 and 1.5 on OLTP and DSS workloads, respectively, compared to a wide superscalar.

Finally, we proposed software-directed register deallocation, a joint software/hardware mechanism for managing shared register files more efficiently. By communicating register last-use information to renaming hardware, a 25% reduction in register file size can be achieved without sacrificing performance.

To make the most effective use of shared hardware, an SMT system relies on the cooperation of both the hardware and software, from the microarchitecture up through the OS and compiler to the application. By efficiently managing shared resources in an SMT-conscious manner, an SMT system has the opportunity to fully exploit the benefits of thread-level parallelism and dynamic resource sharing.

8.1 Future work

Given the benefits of thread-level parallelism, we believe that simultaneous multithreading will proliferate into next-generation, commodity microprocessors. As this transition occurs and as SMT processors become more widespread, a number of related

SMT research issues will gain prominence, including dynamic software policies, thread priority issues, and techniques for improving sequential application performance. We will briefly describe each of these issues.

8.1.1 Dynamic policies

While this dissertation focused on static software policies for SMT processors, dynamic or adaptive solutions will also become very important. Dynamic techniques can enhance an SMT processor in at least two ways.

First, dynamic information could be used to minimize interference in multiprogrammed workloads. When the application mix varies dynamically, static software policies may not be able to address all the potential interactions between applications. Consequently, run-time feedback will be necessary to ensure that a given set of applications or tasks is run in the most efficient manner. Profiling information and other monitoring hardware are two forms of run-time information that could enable the compiler and operating system to use shared SMT hardware even more efficiently. The two main unanswered questions are: what run-time information is most useful, and how can software, such as the operating system scheduler, use the information most effectively?

Second, run-time feedback can enable applications to adapt to the execution conditions on the processor. For example, in the Tera MTA [5], applications are often compiled so that they can dynamically adapt to the number of threads available. The hardware provides a thread creation instruction that requests some number of threads. If they are available, the threads are created. If no threads are available, the instruction returns a failure. This instruction enables the application to adapt to the availability of resources on the processor.

8.1.2 Single-thread priority vs. overall throughput

Another important issue is the reconciliation of policies that emphasize priority-thread or single-thread performance versus overall machine throughput. When running a typical workstation workload or while running benchmarks, one foreground application may be most important. But when running a database workload, for example, throughput is more critical.

As we saw in Chapter 5, the two goals of high priority and throughput may be at odds in some situations. For example, when optimizing for highest overall machine throughput, software speculative execution was inappropriate for non-loop-based applications. When optimizing for greatest single-thread performance, however, speculative execution may still be desirable.

8.1.3 Parallelization of sequential applications

With separate program counters and register state, multithreading provides a nice abstraction for decomposing applications into parallel execution units. On traditional architectures, however, thread creation overhead and synchronization costs have inhibited the aggressive use of threading to improve the performance of applications that are traditionally sequential.

With its dynamic, inter-thread resource sharing, simultaneous multithreading provides an architecture that enables new techniques for improving the performance of these sequential applications. As we have seen in this dissertation, simultaneous multithreading enables the aggressive use of thread-level parallelism to increase processor performance. Because threads can share processor resources efficiently, synchronization, communication, and thread creation overheads may be much lower. These efficiencies can enable more aggressive parallelization techniques, either through aggressive speculation in hardware or through compiler-based parallelization.

While other researchers have begun to explore this space with techniques such as thread-level speculation [94, 117] and threaded multiple path execution [141], we believe that this is a fertile area for future research. Simultaneous multithreading has shown its ability to efficiently execute multiprogrammed and parallel workloads. The next step will be for hardware and software to leverage shared resources in an SMT processor to improve the performance of sequential applications.

References

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

[2] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. APRIL: a processor architecture for multiprocessing. In *17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] A. Aiken and A. Nicolau. Optimal loop parallelization. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 308–317, June 1988.

[4] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[5] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[6] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 166–178, July 1995.

[7] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Hensinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *16th ACM Symposium on Operating System Principles*, pages 1–14, October 1997.

[8] L. A. Barroso, K. Gharachorloo, and E. D. Bugnion. Memory system characterization of commercial workloads. In *25th Annual International Symposium on Computer Architecture*, pages 1-12, June 1998.

[9] C. Beckmann and C. Polychronopoulos. Microarchitecture support for dynamic scheduling of acyclic task graphs. In *25th Annual International Symposium on Microarchitecture*, pages 140–148, December 1992.

[10] J. Boyle, R. Butler, T. Diaz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, Inc., 1987.

[11] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255, October 1997.

[12] B. Calder and D. Grunwald. Fast and accurate instruction fetch and branch prediction. In *21st Annual International Symposium on Computer Architecture*, pages 2–11, April 1994.

[13] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[14] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *25th International Symposium on Microarchitecture*, pages 292–300, December 1992.

[15] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing '92*, pages 114–124, November 1992.

[16] S. Carr, K. S. McKinley, and C. W. Tseng. Compiler optimizations for improving data locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

[17] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 239–245, April 1995.

[18] A. E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *IEEE Computer*, 14(9):18–27, December 1981.

[19] J. Chen, D. Lee, J. Lo, T. Thorp, and P. V. Vleet. An 18-ported register file fora simultaneous multithreading processor. CSE568 Course Project Writeup, University of Washington, March 1997.

[20] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 205–217, June 1995.

[21] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 279–290, June 1995.

[22] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. Papworth, and P. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, C-37(8):967–979, August 1988.

[23] H. Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[24] Z. Cvetanovic and D. Bhandarkar. Characterization of Alpha AXP performance using TP and SPEC workloads. In *21st Annual International Symposium on Computer Architecture*, pages 60–70, April 1994.

[25] G. Daddis, Jr. and H. Torng. The concurrent execution of multiple instruction streams on superscalar processors. In *International Conference on Parallel Processing*, pages I:76–83, August 1991.

[26] J. B. Dennis. Data flow supercomputers. *IEEE Computer*, pages 48–56, November 1980.

[27] K. Dixit. New CPU benchmark suites from SPEC. In *COMPCON '92 digest of papers*, pages 305–310, February 1992.

[28] P. K. Dubey, A. Krishna, and M. J. Flynn. Analytic modeling of multithreaded pipeline performance. In *27th Hawaii International Conference on System Sciences*, pages 361–367, January 1994.

[29] P. K. Dubey, A. Krishna, and M. S. Squillante. Performance modeling of a multithreaded processor spectrum. In *The State-of-the-Art in Performance Modeling and Simulation of Advanced Computer Systems*, pages 1–25. Gordon and Breach, 1996.

[30] J. Edmondson and P. Rubinfield. An overview of the 21164 AXP microprocessor. In *Hot Chips VI*, pages 1–8, August 1994.

[31] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. In *IEEE Micro*, pages 12–19, October 1997.

[32] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *23rd Annual International Symposium on Computer Architecture*, pages 203–212, May 1996.

[33] R. Espasa and M. Valero. Exploiting instruction- and data-level parallelism. In *IEEE Micro*, pages 20–27, October 1997.

[34] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *Second Annual International Symposium on High-Performance Computer Architecture*, pages 40–51, January 1996.

[35] M. K. Farrens and A. Pleszkun. Strategies for achieving improved processor throughput. In *18th Annual International Symposium on Computer Architecture*, pages 362–369, May 1991.

[36] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. The M-Machine multicomputer. In *28th Annual International Symposium on Microarchitecture*, pages 146–156, November 1995.

[37] M. J. Flynn and A. Podvin. Shared resource multiprocessing. In *IEEE Computer*, volume 5, pages 20–28, March/April 1972.

[38] M. Franklin and G. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grained parallel processors. In *25th International Symposium on Microarchitecture*, pages 236–245, December 1992.

[39] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *19th Annual International Symposium on Computer Architecture*, pages 58–67, May 1992.

[40] M. T. Franklin, W. P. Alexander, R. Jauhari, A. M. G. Maynard, and B. R. Olszewski. Commercial workload performance in the IBM POWER2 RISC System/6000 processor. *IBM Journal of Research and Development*, 38(5):555–561, April 1994.

[41] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[42] V. V. Gokhale. Design of the 64-bit option for the Oracle7 relational database management system. *Digital Technical Journal*, 8(4):76–82, 1996.

[43] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-physical registers. In *Fourth Annual International Symposium on High-Performance Computer Architecture*, pages 175–184, February 1998.

[44] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[45] R. Govindarajan, S. Nemawarkar, and P. LeNir. Design and performance evaluation of a multithreaded architecture. In *First IEEE Symposium on High-Performance Computer Architecture*, pages 298–307, January 1995.

[46] V. Grafe, G. Davidson, J. Hoch, and V. Holmes. The Epsilon dataflow processor. In *16th Annual International Symposium on Computer Architecture*, pages 36–45, June 1989.

[47] M. Gulati and N. Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. In *Second International Symposium on High-Performance Computer Architecture*, pages 291–301, February 1996.

[48] B. Gunther. *Superscalar performance in a multithreaded microprocessor*. PhD thesis, University of Tasmania, December 1993.

[49] J. R. Gurd, C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[50] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, October 1996.

[51] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[52] R. Halstead and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *15th Annual International Symposium on Computer Architecture*, pages 443–451, May 1988.

[53] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[54] S. Hily and A. Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report PI-1086, IRISA, February 1997.

[55] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[56] P. Hsu and E. Davidson. Highly concurrent scalar processing. In *13th Annual International Symposium on Computer Architecture*, pages 386–395, June 1986.

[57] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1/2):229–248, May 1993.

[58] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *15th Annual International Symposium on Computer Architecture*, pages 131–140, June 1988.

[59] International Business Machines Corp. RISC System/6000 model J30. http://www.rs6000.ibm.com/hardware/enterprise/j50.html.

[60] J. Janssen and H. Corporaal. Partitioned register files for TTAs. In *28th International Symposium on Microarchitecture*, pages 303–312, December 1995.

[61] W. Kaminsky and E. Davidson. Developing a multiple-instruction-stream single-chip processor. *IEEE Computer*, 12(12):66–76, December 1979.

[62] T. S. Kawaf, D. J. Shakshober, and D. C. Stanley. Performance analysis using very large memory on the 64-bit AlphaServer system. *Digital Technical Journal*, 8(3):58–65, 1996.

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

[64] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee. Exploiting fine-grain thread level parallelism on the multi-ALU processor. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[65] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing, 6th International Workshop*, pages 301–319. Springer-Verlag, August 1993.

[66] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer and Systems*, 10(4):338–359, November 1992.

[67] T. Kiyohara, S. Mahlke, W. Chen, R. Bringmann, R. Hank, S. Anik, and W. Hwu. Register connection: A new approach to adding registers into instruction set architectures. In *20th Annual International Symposium on Computer Architecture*, pages 247–256, May 1993.

[68] J. T. Kuehn and B. J. Smith. The Horizon supercomputing system: Architecture and software. In *Supercomputing ’88*, pages 28–34, November 1988.

[69] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[70] M. Lam and R. Wilson. Limits of control flow on parallelism. In *19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[71] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[72] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, October 1994.

[73] Y. Li and W. Chu. The effects of STEF in finely parallel multithreaded processors. In *First IEEE Symposium on High-Performance Computer Architecture*, pages 318–325, January 1995.

[74] B.-H. Lim and R. Bianchini. Limits on the performance benefits of multithreading and prefetching. In *ACM Sigmetrics Conference on the Measurement and Modeling of Computer Systems*, pages 37–46, May 1996.

[75] J. Llosa, M. Valero, and E. Ayguade. Non-consistent dual register files to reduce register pressure. In *First Annual International Symposium on High-Performance Computer Architecture*, pages 22–31, January 1995.

[76] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer and Systems*, 15(3):322–354, August 1997.

[77] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *30th International Symposium on Microarchitecture*, pages 114–124, December 1997.

[78] T. Lovett and R. Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In 23*rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[79] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1/2):51–142, May 1993.

[80] C. L. Lozano and G. Gao. Exploiting short-lived variables in superscalar processors. In *28th International Symposium on Microarchitecture*, pages 292–302, December 1995.

[81] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective

compiler support for predicated execution using the hyperblock. In *25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[82] M. Martin, A. Roth, and C. Fischer. Exploiting dead value information. In *30th International Symposium on Microarchitecture*, pages 125–135, December 1997.

[83] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, October 1994.

[84] S. McFarling. Combining branch predictors. Technical Report TN-36, DEC-WRL, June 1993.

[85] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–75, September 1992.

[86] B. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *21st Annual International Symposium on Computer Architecture*, pages 166–175, April 1994.

[87] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *23rd Annual International Symposium on Computer Architecture*, pages 67–77, May 1996.

[88] S. Nemawarkar, R. Govindarajan, G. Gao, and V. Agarwal. Analysis of multithreading multiprocessors with distributed shared memory. In *Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 114–121, December 1993.

[89] M. Nemirovsky and W. Yamamoto. Quantitative study of data caches on a multistreamed architecture. In *Workshop on Multithreaded Execution, Architecture and Compilation*, February 1998.

[90] R. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *16th Annual International Symposium on Computer Architecture*, pages 262–272, June 1989.

[91] R. Nikhil, G. Papadopoulos, and Arvind. *T: a multithreaded massively parallel architecture. In *19th Annual International Symposium on Computer Architecture*, pages 156–167, May 1992.

[92] P. R. Nuth and W. J. Dally. The named-state register file: Implementation and performance. In *First Annual International Symposium on High-Performance Computer Architecture*, pages 4–13, January 1995.

[93] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1996.

[94] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Soft-

ware and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University Computer Systems Laboratory, February 1997.

[95] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[96] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *17th Annual International Symposium on Computer Architecture*, pages 82–91, May 1990.

[97] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *18th Annual International Symposium on Computer Architecture*, pages 342–351, May 1991.

[98] S. E. Perl and R. L. Sites. Studies of Windows NT performance using dynamic execution traces. In *Second USENIX Symposium on Operating System Design and Implementation*, pages 169–183, October 1996.

[99] J. A. Piantedosi, A. S. Sathaye, and D. J. Shakshober. Performance of TruCluster systems under the TPC-C benchmark. *Digital Technical Journal*, 8(3):46–57, 1996.

[100] A. R. Pleszkun and G. S. Sohi. The performance potential of multiple functional unit processors. In *15th Annual International Symposium on Computer Architecture*, pages 37–44, May 1988.

[101] R. Prasadh and C.-L. Wu. A benchmark evaluation of a multi-threaded RISC processor architecture. In *International Conference on Parallel Processing*, pages I:84–91, August 1991.

[102] B. R. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *14th Annual Workshop on Microprogramming*, pages 183–197, October 1981.

[103] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22:12–35, January 1989.

[104] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *15th ACM Symposium on Operating System Principles*, pages 285–298, December 1995.

[105] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *30th International Symposium on Microarchitecture*, pages 138–148, December 1997.

[106] J. Rumbaugh. A data flow multiprocessor. *IEEE Transactions on Computers*, C-26(2):138–146, February 1977.

[107] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 1990.

[108] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single-chip processor. In *16th Annual International Symposium on Computer Architecture*, pages 46–53, June 1989.

[109] Silicon Graphics, Inc. The Onyx system family. http://www.sgi.com/Products/hardware/Onyx/Products/sys_lineup.html.

[110] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, 27(7):42–50, July 1993.

[111] M. Slater. SuperSPARC premiers in SPARCstation 10. *Microprocessor Report*, pages 11–13, May 1992.

[112] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Real Time Signal Processing IV*, pages 241–248, 1981.

[113] J. E. Smith and S. Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *IEEE Computer*, 30(9):68–74, September 1997.

[114] G. S. Sohi, S. E. Breach, and T. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[115] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *27th International Symposium on Microarchitecture*, pages 143–147, December 1994.

[116] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[117] J. G. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Fourth Annual International Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[118] S. Storino, A. Aipperspach, J. Borkenhagen, and S. Levenstein. A commercial multithreaded RISC processor. In *International Solid-State Circuits Conference*, February 1998.

[119] Sun Microsystems, Inc. Ultra HPC series overview. http://www.sun.com/hpc/products/index.html.

[120] S. S. Thakkar and M. Sweiger. Performance of an OLTP application on Symmetry multiprocessor system. In *17th Annual International Symposium on Computer Architecture*, pages 228–238, May 1990.

[121] The Standard Performance Evaluation Council. *SPEC CPU '95 Technical Manual*. August 1995.

[122] R. Thekkath and S. J. Eggers. The effectiveness of multiple hardware contexts. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 328–337, October 1994.

[123] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placment on multi-

threaded architectures. In *21st Annual International Symposium on Computer Architecture*, pages 176–186, April 1994.

[124] M. D. Thistle and B. J. Smith. A processor architecture for horizon. In *Supercomputing '88*, pages 35–41, November 1988.

[125] J. Thornton. *Design of a Computer—The Control Data 6600*. Scott, Foresman and Co., Glenview, IL, 1970.

[126] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174, October 1992.

[127] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *Third Annual International Symposium on High-Performance Computer Architecture*, pages 250–260, February 1997.

[128] Transaction Processing Performance Council. *TPC Benchmark B Standard Specification Revision 2.0*. June 1994.

[129] Transaction Processing Performance Council. *TPC Benchmark D (Decision Support) Standard Specification Revision 1.2*. November 1996.

[130] M. Tremblay, B. Joy, and K. Shin. A three dimensional register file for superscalar processors. In *28th Hawaii International Conference on System Sciences*, pages 191–201, January 1995.

[131] J.-Y. Tsai, Z. Jiang, E. Ness, and P.-C. Yew. Performance study of a concurrent multithreaded processor. In *Fourth Annual International Symposium on High-Performance Computer Architecture*, pages 24–35, February 1998.

[132] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *1996 International Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, October 1996.

[133] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[134] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[135] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grain synchronization on a simultaneous multithreading processor. *UCSD CSE Technical Report #CS98-587 and University of Washington Technical Report #UW-CSE-98-06-02*, June 1998.

[136] G. Tyson and M. Farrens. Techniques for extracting instruction level parallelism on MIMD architectures. In *26th International Symposium on Microarchitecture*, pages 128–137, December 1993.

[137] G. Tyson, M. Farrens, and A. R. Pleszkun. MISC: A multiple instruction stream computer. In *25th International Symposium on Microarchitecture*, pages 193–196, December 1992.

[138] J. Vasell. A fine-grain threaded abstract machine. In *1994 International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, August 1994.

[139] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, October 1996.

[140] C. A. Waldspurger and W. E. Weihl. Register relocation: Flexible contexts for multithreading. In *20th Annual International Symposium on Computer Architecture*, pages 120–129, May 1993.

[141] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[142] W. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. In *16th Annual International Symposium on Computer Architecture*, pages 273–280, June 1989.

[143] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[144] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[145] A. Wolfe and J. Shen. A variable instruction stream extension to the VLIW architecture. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, April 1991.

[146] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[147] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques (PACT 95)*, pages 49–58, June 1995.

[148] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirovsky. Performance estimation of multistreamed, superscalar processors. In *27th Hawaii International Conference on System Sciences*, pages I:195–204, January 1994.

[149] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, pages

28–40, April 1996.

Vita

Jack Lee-jay Lo was born on May 4, 1970 in Troy, New York. He graduated from Stanford University in 1991, receiving the B.S. degree (with distinction) in Computer Science. The following year, he received the M.S. degree in Computer Science, also from Stanford. He then left sunny California to continue his graduate studies at the University of Washington, where he received his Ph.D. degree in 1998.