Simultaneous Multithreading

by

Dean Michael Tullsen

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

Doctor of Philosophy

University of Washington

1996

Approved by____
(Co-Chairperson of Supervisory Committee)

____
(Co-Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree____

Date____

Table of Contents

List of Figures

List of Tables

Acknowledgments

I owe thanks to many people who have contributed to this dissertation and other aspects of my academic career. I owe a particular debt to my advisors, Susan Eggers and Hank Levy, who have contributed greatly to the origination and execution of this research. In addition, they have contributed in so many ways to my development as a researcher and scholar, always holding me to high standards.

Many researches have provided discussion, ideas, information, or correction that helped form my research. Among those are Jean-Loup Baer, Norm Jouppi, Burton Smith, John Zahorjan, Andy Glew, Radhika Thekkath, Tor Jeremiassen, Dennis Lee, and Thu Nguyen. I want to particularly thank Jack Lo for many discussions on this work and the future directions of simultaneous multithreading. Joel Emer, Rebecca Stamm, and Tryggve Fossum really spurred this project on by their enthusiasm for the ideas in it. But much more than that, they helped set key directions for the research and added a tremendous amount of experience and credibility to the work by coming along as collaborators.

The Intel Foundation, the Computer Measurement Group, Microsoft Corporation, the General Electric Foundation, and the Henry Gray Foundation all provided much-appreciated funding for my research.

I never could have completed this degree if my wife, Nancy Tullsen, and our kids, Katie, Daniel, and Joshua had not modeled Phillipians 4:12, “content in any and every situation, whether well fed or hungry, whether living in plenty or in want.” Their constant love and support made everything easier and a lot more fun.

None of this would matter at all if not for the Lord Jesus Christ, my motivation and my savior.

Chapter 1

Introduction

This dissertation examines *simultaneous multithreading* (SMT), a processor architecture technique that combines the multiple-instruction-issue features of modern superscalars with the latency-hiding ability of multithreaded architectures; SMT permits multiple threads to issue instructions to the processor's functional units in a single cycle. The objective of simultaneous multithreading is to substantially increase processor utilization in the face of both long instruction latencies and limited available parallelism per thread.

Modern superscalar processors can detect and exploit high levels of parallelism in the instruction stream, being able to begin execution of 3 to 5 instructions every cycle (see Table 1.1). They do so through a combination of multiple independent functional units and sophisticated logic to detect instructions that can be issued (sent to the functional units) in parallel. However, their ability to fully use this hardware is ultimately constrained by instruction dependencies and long-latency instructions — limited available parallelism in the single executing thread. The effects of these are shown as vertical waste (completely wasted cycles) and horizontal waste (wasted issue opportunities in a partially-used cycle) in Figure 1.1(a), which depicts the utilization of issue slots on a superscalar processor capable of issuing four instructions per cycle.

Multithreaded architectures, such as HEP [Smi81a], Tera [ACC^{+}90], MASA [HF88], and Alewife [ALKK90] employ multiple threads with fast context switching between threads; fast context switches require that the state (program counter and registers) of multiple threads be stored in hardware. Latencies in a single thread can be hidden with instructions from other threads, taking advantage of inter-thread parallelism (the fact that in-

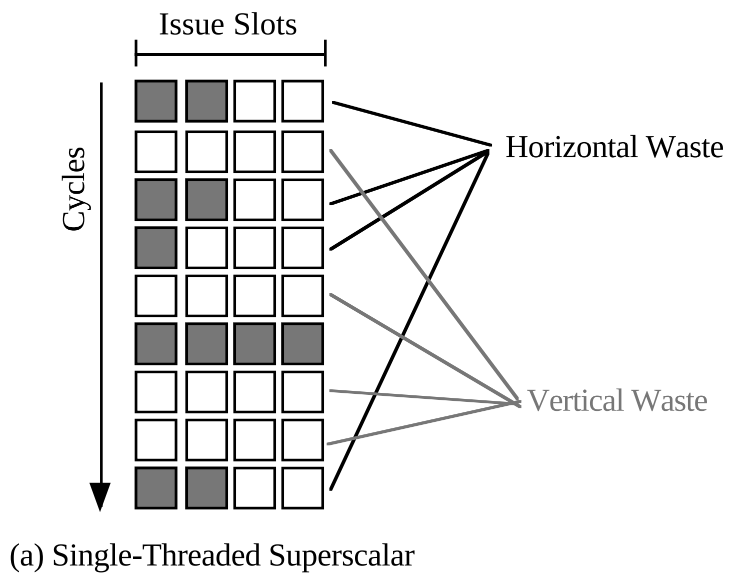
Table 1.1: **Issue rates for current superscalar processors.**

Processor	**Maximum Issue Rate**
Pentium Pro [Rep95]	3
Alpha 21164 [ER94]	4
MIPS R10000 [Rep94a]	5
PowerPC 604 [Den94]	4
Sun UltraSparc [Rep94b]	4
HP PA-8000 [Rep94c]	4

structions from different threads are nearly always independent). *Fine-grain* multithreaded architectures can context switch as often as every cycle at no cost. The ability to interleave instructions from different threads allows fine-grain machines to hide nearly all sources of latency, attacking vertical waste, as shown in Figure 1.1(b), which depicts fine-grain multithreading applied to a superscalar processor. These machines use inter-thread parallelism *between* cycles to hide latencies, but because they issue instructions from only one thread at a time, they cannot use inter-thread parallelism *within* a cycle to increase issue parallelism (the amount of parallelism available to the issue mechanism in a single cycle). Conventional multithreading, if applied to a superscalar, is thus limited by the amount of parallelism that can be found in a single thread in a single cycle.

Simultaneous multithreading combines the superscalar's ability to exploit high levels of instruction-level parallelism with multithreading's ability to expose inter-thread parallelism to the processor. By allowing the processor to issue instructions from multiple threads in the same cycle, SMT uses multithreading not only to hide latencies, but also to increase issue parallelism. Simultaneous multithreading exploits both vertical and horizontal waste, as shown in Figure 1.1(c). The ability to exploit horizontal waste will be critical to getting high utilization from the next generation of superscalar processors — in Chapter 3, we measure less than 20% utilization of an 8-issue single-threaded superscalar.

This dissertation shows that simultaneous multithreading has the potential to dramatically improve that utilization, outstripping other alternatives: e.g., fine-grain multithreading and single-chip multiprocessors. More importantly, this thesis demonstrates that those gains can be realized with a realistic architecture for simultaneous multithreading that requires



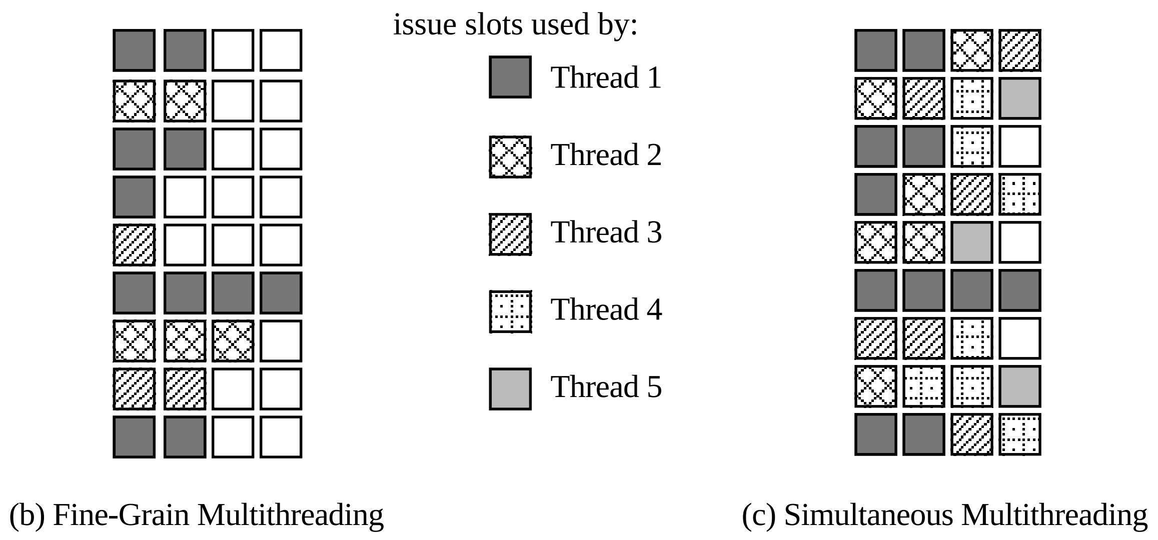

issue slots used by:
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
(b) Fine-Grain Multithreading
(c) Simultaneous Multithreading

Figure 1.1: **Utilization of issue slots on (a) a superscalar processor, (b) a superscalar processor with fine-grain multithreading, and (c) a simultaneous multithreading processor.**

surprisingly few changes to a conventional wide-issue superscalar processor.

The simultaneous multithreading architectures this dissertation describes have the ability to exploit virtually any form of parallelism available in the workload. They can exploit single-stream instruction-level parallelism, because even a single thread has full access to the execution resources of a superscalar processor. They can exploit the parallelism of independent programs in a multiprogrammed workload (process-level parallelism), as demonstrated by the results shown in this dissertation. They can also exploit fine-, medium-, or coarse-grain parallelism in a single program that can be expressed (either by the programmer or the compiler) in terms of separate (not necessarily independent) tasks or threads.

Each thread in the processor is simply an instruction stream associated with a particular hardware context. Regardless of whether the threads come from separate programs, or parallelism within a single program, or a combination of both, the scheduling and mixing of instructions is done by hardware; the compiler need not be involved at all. In fact, in all of the data presented in this dissertation, we assume no compiler support for simultaneous multithreading.

Previous studies have examined architectures that exhibit simultaneous multithreading through various combinations of VLIW, superscalar, and multithreading features; we discuss these in detail in Chapter 2. Our work differs from and extends that work in the base architecture we use for comparison, in the content and direction of the work, and the methodology used (including the accuracy and detail of our simulations, the workload, and the wide-issue compiler optimization and scheduling technology). The primary differences between this work and other research is summarized in Section 1.3.

The inherent advantages of simultaneous multithreading with respect to processor utilization are clear, but potential obstacles exist that may prevent us from seeing those advantages in practice. Those obstacles arise in two areas: (1) the effects of sharing processor resources among multiple threads and (2) the complexity of implementing simultaneous multithreading. This dissertation addresses both of these obstacles.

1.1 Obstacle 1: Sharing Processor Resources Among Threads

In our model of simultaneous multithreading, nearly all processor resources are completely shared among the hardware contexts, or threads, potentially putting much higher demands on these shared resources. Higher demand on the functional units and issue bandwidth is desirable because it raises instruction throughput, but in so doing we may also place higher demands on the caches, the TLBs, the branch prediction logic, the fetch bandwidth, and the instruction scheduling mechanism. When comparing simultaneous multithreading to single-threaded superscalar execution in this dissertation, each of these structures typically remains the same size for both architectures, so that the extra contention and loss of locality is fully measured.

The multiprogrammed workload used in this dissertation, in fact, presents a worst-case scenario for those structures. With each thread executing a separate program, there is no overlap between the working sets of different threads, either in data or code. An eight-thread workload, then, grows the dynamic working set of the applications by a factor of eight over a single-threaded workload. Cache miss rates, branch mispredictions, and TLB misses increase significantly with a multithreaded (multiprogrammed) workload; however, results throughout this dissertation will show that although the total number of latency-causing events increases as we add threads, the processor's ability to hide latencies rises much more quickly. The net result of these two competing effects, then, is a strong net increase in performance with more threads. This is least obvious with the branch prediction logic, where mispredictions do not cause latencies so much as they cause the machine to waste resources. However, even in that case, we find that a simultaneous multithreading machine is much more tolerant of branch mispredictions than a single-threaded processor (see Section 5.6).

Chapter 5 demonstrates an even more counter-intuitive result — that the instruction scheduling mechanism and the fetch mechanism can each become significantly more efficient when shared by multiple threads. For each of the shared structures, then, we show that either the structure itself becomes more efficient when shared (the fetch bandwidth, the instruction scheduling mechanism) or processor performance becomes much less dependent on the efficiency of the mechanism (caches, TLBs, branch prediction) with multiple thread execution.

1.2 Obstacle 2: The Complexity of Simultaneous Multithreading

A simultaneous multithreading processor is more complex than a corresponding single-threaded processor. Potential sources of that complexity include:

- fetching from multiple program counters — Following multiple instruction streams will either require parallel access to the instruction cache(s) and multiple decode pipelines, or a restricted fetch mechanism that has the potential to be a bottleneck.

- multiple register sets — Accessing multiple register sets each cycle requires either a large single shared register file or multiple register files with a complex interconnection network between the registers and the functional units. In either case, access to the registers will be slower than in the single-threaded case.

- instruction scheduling — In our SMT processors, the scheduling and mixing of instructions from various threads is all done in hardware. In addition, if instructions from different threads are buffered separately for each hardware context, the scheduling mechanism is distributed (and more complex), and the movement of instructions from buffers to functional units requires another complex interconnection network.

Chapter 5 presents an architecture for simultaneous multithreading that solves each of these problems, requiring few significant changes to a conventional superscalar processor.

1.3 Contributions of This Dissertation

This dissertation makes the following contributions:

- It presents a new approach to the analysis of limits to single-stream instruction-level parallelism, identifying a cause for every wasted issue opportunity, thus characterizing the source of all idle execution cycles.

- It identifies simultaneous multithreading as a processor architecture technique and analyzes it in isolation from other architectural features.

- It identifies several models of simultaneous multithreading which potentially differ in implementation complexity and evaluates differences in their performance.

- It compares simultaneous multithreading with alternative architectural models that attempt to improve single-chip instruction throughput, specifically wide-issue superscalars, fine-grain multithreading, and single-chip multiprocessors.

- It analyzes the performance differences between shared and private-per-thread caches for a multiple-hardware-context machine, such as a simultaneous multithreading processor or a multiprocessor.

- It presents a detailed and implementable architecture that exploits simultaneous multithreading without excessive changes to a conventional wide-issue superscalar processor.

- It demonstrates that support for simultaneous multithreading only slightly impacts the performance of a single thread executing alone.

- It investigates several design issues for a simultaneous multithreading processor, including cache, fetch unit, and issue mechanism design.

- It shows that fetch unit performance can be substantially improved by (1) partitioning the fetch bandwidth among multiple threads and (2) being selective about which threads to fetch each cycle.

- It presents a response time model of simultaneous multithreading, showing that an SMT processor provides significant response time improvements over a single-threaded processor for a multiprogrammed workload.

1.4 Organization

This dissertation is organized as follows. Chapter 2 discusses related work, including previous work on existing models of hardware multithreading and some proposed architectures that feature simultaneous multithreading. Chapter 3 provides motivation for the simultaneous multithreading approach, analyzing a projected future single-context superscalar processor. Chapter 4 explores the potential for simultaneous multithreading. Several high-level models of simultaneous multithreading are defined, each with different performance/implementation complexity tradeoffs. In addition, the performance of SMT in general is contrasted with alternative architectures aimed at maximizing single-chip throughput. These architectures include a wide-issue superscalar processor, that same processor with fine-grain multithreading, and a variety of single-chip multiprocessor configurations.

Chapter 5 presents a realistic architecture for SMT that demonstrates that the potential performance advantages of simultaneous multithreading demonstrated in Chapter 4 are available without adding undue complexity to a conventional superscalar architecture. Chapter 6 presents an analytic response-time model of a simultaneous multithreading processor and system, demonstrating that the increased throughput of simultaneous multithreading also produces significant improvements in response time for individual applications. Chapter 7 summarizes the results.

Chapter 2

Related Work

This dissertation builds on and benefits from previous research in several areas. This chapter describes previous work on instruction-level parallelism, on several traditional (coarse-grain and fine-grain) multithreaded architectures, and on two architectures (the M-Machine and the Multiscalar architecture) that are not simultaneous multithreaded, but do have multiple contexts active simultaneously. It also discusses other studies of architectures that exhibit simultaneous multithreading and contrasts this dissertation with these in particular. Last, we describe other work containing analytic models of multithreading processors.

2.1 Instruction-Level Parallelism

Previous work that measured, characterized, and identified limits to the instruction-level parallelism (ILP) in various workloads adds to the motivation for simultaneous multithreading. The strong body of evidence that per-thread ILP is limited validates the need to supplement instruction-level parallelism with the ability to exploit thread-level parallelism (inter-thread parallelism).

Each of the following papers modeled a range of machine models, from practical to theoretical. Although each of these studies provides large numbers of results, of most interest to this dissertation are practical, but aggressive, models of superscalar execution. Specific results along those lines are quoted here. Butler *et al.* [BYP$^+$91] found instruction-level parallelism of 1 to 2.5 instructions per cycle for an 8-issue processor with realistic branch prediction, out-of-order and speculative execution, and a 32-entry instruction scheduling window. Wall [Wal91] measured ILP of 5.8 instructions per cycle for a 32-issue processor with a 32-entry scheduling window, infinite branch prediction resources, perfect memory aliasing, out-of-order and speculative execution, and all instruction latencies of a single cycle. Smith *et al.* [SJH89] identify the fetch unit as a bottleneck in a superscalar machine, as we do for an SMT processor in Chapter 5. They measure a throughput of 2.6 instructions per cycle for a 4-issue machine with branch prediction and perfect register renaming. Theobald *et al.* [TGH92] measure ILP of 3.6 instructions per cycle for an out-of-order, speculative processor with realistic branch prediction, a 64-entry instruction window, infinite issue bandwidth, and all single cycle instruction latencies.

Although these results confirm that it is difficult to find sufficient single-thread parallelism to take full advantage of wide-issue superscalar processors, our results in Chapter 3 show that even these results are optimistic when a more realistic machine model is used. In particular, we model (in contrast to all but [SJH89]) realistic fetch limitations, a realistic cache hierarchy, TLB misses, and real instruction latencies (in contrast to the particular values quoted above from [Wal91] and [TGH92]).

The analysis of limitations to instruction-level parallelism in Chapter 3 of this dissertation provides a different perspective from those studies, which remove barriers to parallelism (i.e., apply real or ideal latency-hiding techniques) and measure the resulting performance. Chapter 3 identifies the cause of all lost instruction-issue opportunities, thus assigning a cost to each inhibitor of parallelism.

2.2 Multithreaded Architectures

This dissertation builds on previous research in multithreaded architectures. A multithreaded architecture stores the state of multiple contexts, or processes, in hardware, allowing it to quickly perform a context switch without incurring any software overhead. This allows those architectures to hide instruction latencies by executing instructions from other contexts.

Coarse-grain multithreaded architectures [ALKK90, SBCvE90, TE94] execute instructions from a single stream until encountering a long-latency operation, such as a cache miss or blocking synchronization, then switch to another context (typically at a cost of a few cycles to drain the pipeline). This simplifies the design (e.g., by not requiring simultaneous, or even consecutive-cycle, access to different register files) but limits the types of latencies that can be hidden.

Fine-grain multithreaded processors [Smi81a, ACC$^+$90, HF88, NA89, LGH94] have the ability to context-switch every cycle at no cost. This allows them to interleave instructions from different threads each cycle, thus hiding a wider range of processor latencies. The earliest multithreading machines used fine-grain multithreading to take advantage of pipelining without hardware to detect and exploit single-instruction-stream parallelism. The I/O unit of the CDC 6600 [Tho70] allowed 10 peripheral processors to share a 10-stage pipelined central hardware core. Each processor could start an instruction every tenth cycle.

The processors in the Denelcor HEP computer [Smi81a] also switched contexts (issued instructions from a new context) every cycle in round-robin fashion among the running threads. No single thread could have more than one instruction in the 8-stage pipeline, so 8 threads were required to get full utilization from the processor. But unlike the CDC, it could multiplex among more processes, as many as 64.

Kaminsky and Davidson [KD79] also propose fine-grain multithreading to share a processor pipeline, this time in the context of a single-chip processor.

The Tera MTA architecture [ACC$^+$90] does not impose a round-robin interleaving of instructions, instead selecting an instruction each cycle from among those threads whose next instruction is ready to execute. The Tera machine, like those already mentioned, includes no bypass logic in the pipeline and no data caches, instead depending on a high degree of thread parallelism to hide those latencies. This prevents these architectures from achieving high throughput in the absence of thread-level parallelism, but enables a faster processor clock rate.

Laudon *et al.*'s "interleaved" processor [LGH94] also switches context in round-robin fashion every cycle, but skips over threads which have encountered a long-latency operation, enabling full utilization with fewer threads than the HEP. This processor, which also maintains the scalar processor's bypass logic and data caches, allows a single thread to run at full speed relative to a single-threaded scalar machine. Farrens and Pleszkun [FP91] examine a similar scheme to share the pipeline of a CRAY-1 among two threads; however, they assume the presence of separate fetch/decode pipelines.

The P-RISC architecture [NA89] combines fine-grain multithreading with lightweight fork and join mechanisms to enable efficient execution of very fine-grain parallel tasks, such as those produced by dataflow code. MASA [HF88] uses fine-grain multithreading to hide latencies on each processor of a multiprocessor designed for running parallel lisp applications.

In Chapter 4, we extend these results on fine-grain multithreaded processors by showing how fine-grain multithreading runs on a superscalar processor.

A feature common to several implementations of fine-grain multithreading is the inability to run a single thread quickly in the absence of thread-level parallelism. That, however, is a result of design philosophy rather than anything inherent to multithreading, those machines having been optimized for high multiple-thread throughput. In this dissertation, we present an architecture that demonstrates that multithreading need not slow down single-thread throughput significantly. Further, it shows that with simultaneous multithreading, high single-thread throughput and high multiple-thread throughput are by no means incompatible.

2.3 Multiple-Issue, Multiple-Context Processors

This section notes three machine organizations which feature both multiple instruction issue and multiple contexts, yet do not fit our definition of simultaneous multithreading.

The Tera processor [ACC+90], described in the previous section, features both multithreading and multiple-operation-issue, as each Tera instruction is a 3-operation LIW instruction. This it is fine-grain multithreaded, with the compiler responsible for scheduling operations (from the same thread) that issue in the same cycle.

In the M-Machine [FKD+95] each processor cluster schedules LIW instructions onto execution units on a cycle-by-cycle basis similar to the Tera scheme. However, there is no simultaneous issue of instructions from different threads to functional units on individual clusters. It is essentially, then, a multiprocessor composed of fine-grain multithreaded, LIW processors.

The Multiscalar architecture [FS92, Fra93, SBV95] assigns fine-grain threads to processors, so competition for execution resources (processors in this case) is at the level of a task rather than an individual instruction. However, the architecture is much more tightly coupled than a typical multiprocessor, as the processors executing the tasks share register names, with data movement among virtually shared (but physically distinct) registers accomplished by hardware. Another unique feature of the multiscalar architecture is that it does speculative execution at the task level.

2.4 Simultaneous Multithreaded Architectures

Several studies have examined architectures that feature what this dissertation has labeled simultaneous multithreading, the ability for multiple hardware contexts to issue instructions to a processor's functional units in the same cycle. These studies demonstrate the viability of simultaneous multithreading in a diversity of architectural contexts and design philosophies. Many of the unique contributions of this dissertation are summarized in Section 1.3; others are given at the end of this section.

Hirata *et al.* [HKN$^+$92] present an architecture for a multithreaded superscalar processor and simulate its performance on a parallel ray-tracing application. They do not simulate caches or TLBs, and their architecture has no branch prediction mechanism. They show speedups as high as 5.8 over a single-threaded architecture when using 8 threads. Their architecture features distinct register files, holding both active and inactive threads. They simulate configurations that allow both single-instruction-per-thread issue, and multiple instruction issue. The processor features standby stations associated with each functional unit which hold dependence-free instructions. Scheduling of instructions onto standby stations/functional units uses a distributed mechanism for which they allow an extra stage in the pipeline.

Yamamoto *et al.* [YST$^+$94] present an analytical model of multithreaded superscalar performance, backed up by simulation. Their study models perfect branching, perfect caches and a homogeneous workload (all threads running copies of the same trace). They report increases in instruction throughput of 1.3 to 3 with four threads. In their architecture, ready-to-issue instructions are moved (out-of-order) from per-thread issue windows to a global scheduling queue, from which they are issued onto available functional units. They model multiple functional unit configurations. Yamamoto and Nemirovsky [YN95] simulate an SMT architecture with separate instruction fetch, separate instruction queues, shared caches and up to four threads. The workload is multiprogrammed and they see a throughput improvement just over 2 with four threads. They account for the complexity of the separate instruction queues (which require a distributed issue mechanism) by suggesting that each queue be half the size of the single-thread processor, resulting in only a 5% performance cost.

Keckler and Dally [KD92] describe an architecture that dynamically interleaves operations from VLIW instructions onto individual functional units. They report speedups as high as 3.12 for some highly parallel applications. Operations in the same VLIW instruction issue when their operands are available, operations in different instructions from the same thread are issued in-order. They use a single virtual register file, but it is distributed physically among processor clusters. They model different levels of hardware multithreading and different FU configurations running parallel benchmarks. They model both a perfect memory subsystem and a statistically-generated miss rate for the data cache. They discuss chip layout considerations and provide some chip area estimates for various components of a processor cluster.

Prasadh and Wu [PW91] use a similar issue mechanism to Keckler and Dally, mixing operations from the current VLIW instruction from each thread. They model multiple configurations of functional units, assume infinite caches (including per-thread instruction caches) and show a maximum speedup above 3 over single-thread execution for parallel applications. They also examine the register file bandwidth requirements for 4 threads scheduled in this manner. Each thread in their model accesses a separate register bank.

Daddis and Torng [DT91] plot increases in instruction throughput as a function of the fetch bandwidth and the size of the dispatch stack. The dispatch stack is the shared instruction window that issues all fetched instructions. Their system has two threads, unlimited functional units, unlimited issue bandwidth (but limited fetch bandwidth), and a zero-latency memory subsystem. They report a near doubling of throughput.

Gulati and Bagherzadeh [GB96] model a 4-issue machine with four hardware contexts and a single compiler-partitioned register file. They model an instruction window composed of four-instruction blocks, each block holding instructions from a single thread. Gulati and Bagherzadeh fetch from a single thread each cycle, and even look at thread selection policies, but find no policy with improvement significantly better than round robin.

In addition to these, Beckmann and Polychronopoulus [BP92], Gunther [Gun93], Li and Chu [LC95], and Govindarajan *et al.* [GNL95] discuss architectures that feature simultaneous multithreading. Each of these studies, however, represent a different design space than the architectures presented in this dissertation, as none of them have the ability to issue more than one instruction per cycle per thread. They represent architectures that can exploit thread-level parallelism but not single-stream instruction-level parallelism.

This dissertation makes many unique contributions, some of which are listed in Section 1.3, and differs from previous work both in scope and direction. While several studies examine architectures that include some form of SMT, this research focuses solely on simultaneous multithreading, examining in detail its impact on the whole machine and presenting an architecture focused on taking maximal advantage of SMT. This work is unique in its focus on making simultaneous multithreading a viable near-term option for mainstream processors; thus the presented architecture is unique in its combined goals of low architectural impact, high single-thread performance, and high multiple-thread performance. Another distinctive of this work is the detailed simulation of all processor and memory/cache latencies (including emulation of instructions following mispredicted branches) and instruction fetch, instruction scheduling, and register renaming limitations.

2.5 Analytic Models of Multithreading

Most previous analytic models of multithreading systems have focused on system-level throughput increases due to the effects of multithreading, not extending those results to response time models of per-application speedups in a multiuser system. In this dissertation we use detailed simulation to determine the throughput increases, but use that data for an analytic model of response time on a simultaneous multithreaded system. Measuring throughput through simulation provides greater accuracy; however, the advantage of modeling throughput is the ease with which low-level quantities can be varied.

Saavedra *et al.* [SBCvE90] present a utilization model of a coarse-grain multithreaded processor, accounting for context-switch time and cache interference.

Agarwal [Aga92] presents a model of a coarse-grain multithreaded processor. The model factors in network bandwidth, cache interference among threads, and the context-switch overhead. This model uses the increase in processor utilization as its measure of the benefit of multithreading.

Lim *et al.* [LB96] also model coarse-grain multithreading, again evaluating architectures in terms of the gains in processor utilization. They do not have a model of cache interference.

Nemawarkar *et al.* [NGGA93] use a more complex queueing network model of a coarse-grain multithreaded multiprocessor, where each node in the processor is modeled with multiple queues (CPU, memory, and the network switch). They use approximate mean value analysis to solve the queueing network, also using utilization as the measure of performance gain. They have no cache model, not varying the run-length (the average number of memory accesses between cache misses) with the number of threads.

Eickmeyer *et al.* [EJK$^+$96] include a response time model of a coarse-grain multithreaded processor, using trace-driven simulation to parameterize the model of cache interference.

Dubey, Krishna, and Flynn [DKF94] present a model of fine-grain multithreaded pipeline utilization based on an assumed distribution of inter-instruction interlock (dependence) distances. They use discrete-time Markov chains. Dubey, Krishna, and Squillante [DKS96] extend this work by providing models of both fine-grain and coarse-grain utilization, and apply approximate solution techniques to enable larger state spaces.

All of the previously mentioned studies modeled coarse-grain or fine-grain multithreading. Yamamoto *et al.* [YST$^+$94] presents an analytic model of a simultaneous multithreading processor. They simulate a multiprogrammed (but, as mentioned, all processes running the same code) workload, but solve only for system throughput, not response time. Their model is a low-level Markov model of functional unit utilization, including characterization of the workload in terms of data and control hazards, and assuming a perfect cache.

The model in Chapter 6 is distinct from these studies because (1) it models response time, and (2) it models simultaneous multithreading, the model parameterized by careful simulation of an SMT processor.

Chapter 3

Superscalar Bottlenecks: Where Have All the Cycles Gone?

This chapter motivates simultaneous multithreading by exposing the limits of wide superscalar execution, identifying the sources of those limitations, and bounding the potential improvement possible from specific latency-hiding techniques.

Modern processors have the ability to exploit increasingly high amounts of instruction-level parallelism in hardware. One aspect of this is a move toward wider, more powerful superscalar processors. The current crop of superscalar processors (see Table 1.1) can execute three to five instructions per cycle. It is likely that we will eventually see 8-issue or wider processors. This trend has been coupled with advances that allow us to expose more parallelism to the processor, such as register renaming (eliminating false dependences), out-of-order execution, speculative execution, and improved compiler instruction scheduling. However, this chapter shows that our ability to expose ILP is not keeping up with our ability to exploit it in hardware, resulting in severe underutilization of the execution resources.

3.1 Methodology for the Superscalar Evaluation

This chapter evaluates the utilization of a next-generation wide-issue superscalar processor. To do this, we have defined a simulation environment that represents a future processor running a workload that is highly optimized for execution on the target machine.

3.1.1 Simulation Environment

Our simulator uses emulation-based instruction-level simulation, like Tango [DGH91] and g88 [Bed90]. Like g88, it features caching of partially decoded instructions for fast emulated execution. It executes unmodified Alpha object code, but borrows significantly from MIPSI [Sir93], a MIPS-based superscalar simulator.

Our simulator models the execution pipelines, the memory hierarchy (both in terms of hit rates and bandwidths), the TLBs, and the branch prediction logic of a wide superscalar processor. It is based on the Alpha AXP 21164 [ER94], augmented for wider superscalar execution. The model also deviates from the Alpha in some respects to support increased single-stream parallelism, such as more flexible instruction issue, improved branch prediction, and larger, higher-bandwidth caches, consistent with our attempt to be modeling a next-generation processor. This simulated machine also represents the base on which the simultaneous multithreading simulators for all following chapters are built, although several assumptions about the underlying architecture change significantly for the architectures examined in Chapter 5.

The simulated configuration contains 10 functional units of four types (four integer, two floating point, three load/store and 1 branch) and a maximum issue rate of 8 instructions per cycle. We assume that all functional units are completely pipelined. Table 3.1 shows the instruction latencies used in the simulations, which are derived from the Alpha 21164.

The caches (Table 3.2) are multi-ported by interleaving them into banks, similar to the design of Sohi and Franklin [SF91]. An instruction cache access occurs whenever the program counter crosses a 32-byte boundary; otherwise, the instruction is fetched from the prefetch buffer. We model lockup-free caches and TLBs. TLB misses require two full memory accesses and no execution resources.

We support limited dynamic execution. Dependence-free instructions are fetched into an eight-instruction-per-thread scheduling window; from there, instructions are scheduled onto functional units in order; however, instructions can bypass previous instructions when a ready instruction fails to issue due to unavailability of a functional unit. Thus, we never waste an issue slot due to functional unit conflicts. Instructions not scheduled due to functional unit availability have priority in the next cycle. We complement this straightforward issue model with the use of state-of-the-art static scheduling, using the Multiflow trace scheduling compiler [LFK+93]. This reduces the benefits that might be gained by full dynamic execution. We use this model of instruction issue, which is primarily still in-order execution, for two reasons. First, the analysis done in this chapter, identifying the cause of each wasted issue slot, would be impossible on an out-of-order processor.

Table 3.1: **Simulated instruction latencies**

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 (L1 cache hit, no bank conflicts)	2
load (L2 cache hit)	8
load (L3 cache hit)	14
load (memory)	50
control hazard (br or jmp predicted)	1
control hazard (br or jmp mispredicted)	6

With the current assumptions, we can always identify a single instruction responsible for each wasted issue slot, and that instruction will be stalled for one (or a few) reasons. With an out-of-order machine, every instruction in the machine is partially responsible for any idle issue slots. The baseline architecture of Chapter 5, where the ability to extract these types of detailed component measurements is less important, is an out-of-order processor. Second, the highest performing processor available at the time of these studies was the Alpha 21164, an in-order processor. This machine is also the base architecture upon which our simulator is built.

A 2048-entry, direct-mapped, 2-bit branch prediction history table [Smi81b] supports branch prediction; the table improves coverage of branch addresses relative to the Alpha, which only stores prediction information for branches that remain in the I cache. The prediction history table is direct-mapped and does 2-bit prediction. Conflicts in the table are not resolved. To predict return destinations, we use a 12-entry return stack like the 21164 (one return stack per hardware context). Our compiler does not support Alpha-style hints for computed jumps; we simulate the effect with a 32-entry jump table, which records the last jumped-to destination from a particular address.

Table 3.2: **Details of the cache hierarchy**

	ICache	**DCache**	**L2 Cache**	**L3 Cache**
Size	64 KB	64 KB	256 KB	2 MB
Assoc	DM	DM	4-way	DM
Line Size	32	32	32	32
Banks	8	8	4	1
Transfer time/bank	1 cycle	1 cycle	2 cycles	2 cycles

3.1.2 Workload

Our workload is most of the SPEC92 benchmark suite [Dix92]. In these experiments, all of the benchmarks are run to completion using the default data set(s) specified by SPEC.

We compile each program with the Multiflow trace scheduling compiler, modified to produce Alpha code scheduled for our target machine. The applications were each compiled with several different compiler options; the executable with the lowest single-thread execution time on our target hardware was used for all experiments.

3.2 Results

Using the simulated machine already described, we measure the issue utilization, i.e., the percentage of issue slots that are filled each cycle, for most of the SPEC benchmarks. We also record the cause of each empty issue slot. For example, if the next instruction cannot be scheduled in the same cycle as the current instruction, then the remaining issue slots this cycle, as well as all issue slots for idle cycles between the execution of the current instruction and the next (delayed) instruction, are assigned to the cause of the delay. When there are overlapping causes, all cycles are assigned to the cause that delays the instruction the most; if the delays are additive, such as an instruction TLB miss and an I cache miss, the wasted cycles are divided up appropriately. The following list specifies all possible sources of wasted cycles in our model, and some of the latency-hiding or latency-reducing techniques that might apply to them. Previous work [Wal91, BYP$^+$91, LW92], in contrast, quantified some of these same effects by removing barriers to parallelism and measuring the resulting increases in performance.

Sources of waste are:

- **Instruction cache miss —** Issue slots are wasted in this case when the processor cannot issue instructions because the fetch unit is stalled by a cache miss. The wasted issue slots due to I cache misses could be reduced by a larger, more associative, or faster instruction cache hierarchy, or hardware instruction prefetching.

- **Instruction TLB miss —** The instruction cannot be fetched because of a TLB miss. This cost could be reduced by decreasing the TLB miss rate (e.g., increasing the TLB size), hardware instruction prefetching, or faster servicing of TLB misses.

- **Data cache miss —** The current instruction is waiting for the result of an earlier load that missed in the cache. This could be reduced by a larger, more associative, or faster data cache hierarchy, hardware or software prefetching, improved static instruction scheduling, or more sophisticated dynamic execution.

- **Data TLB miss —** The current instruction is waiting for data from an earlier load that missed in the data TLB. This could be reduced by decreasing the data TLB miss rate (e.g., increasing the TLB size), hardware or software data prefetching, or faster servicing of TLB misses.

- **Branch misprediction —** The right instruction could not be issued because the fetch unit was fetching the wrong instructions due to a mispredicted branch. This cost could be reduced by an improved branch prediction scheme (e.g., larger structures, more accurate prediction scheme), or a lower branch misprediction penalty (shorter pipeline, earlier detection of mispredictions).

- **Control hazard —** Because this machine does not do speculative execution, we impose a short (1 cycle) control hazard between a conditional branch or computed jump and the following instructions. This cost could be reduced by allowing speculative execution, or doing more aggressive if-conversion using the conditional moves in the Alpha instruction set.

- **Load delays (first-level cache hits)** — Loads that hit in the cache have a single-cycle latency. This cost could be reduced by improved static instruction scheduling or out-of-order (dynamic) scheduling.

- **Short integer delay** — Most integer instructions have 1- or 2-cycle latencies. This cost could be reduced by shorter latencies in some cases, or the delays could be hidden by improved instruction scheduling.

- **Short floating point delay** — Most floating point operations require 4 cycles. This cost could be reduced by improved instruction scheduling or shorter latencies.

- **Long integer, long fp delays** — Multiply is the only long integer operation, divide is the only long floating point operation. These could be reduced by shorter latencies or improved instruction scheduling.

- **Memory conflict** — Accesses to the same memory location are not allowed in the same cycle, stalling the second access. This could be eliminated in some cases with improved instruction scheduling.

Our results, shown in Figure 3.1, demonstrate that the functional units of our wide superscalar processor are highly underutilized. From the composite results bar on the far right, we see issue slot utilization of only 19% (the "processor busy" component), which represents an average execution of less than 1.5 instructions per cycle on the 8-issue machine.

These results also indicate that there is no dominant source of wasted issue bandwidth. Although there are dominant items in individual applications (e.g., mdljsp2, swm, fpppp), the dominant cause is different in each case. In the composite results we see that the largest cause (short FP dependences) is responsible for 37% of the issue bandwidth, but there are six other causes that account for at least 4.5% of wasted cycles. Even completely eliminating any one factor will not necessarily improve performance to the degree that this graph might imply, because many of the causes overlap.

Not only is there no dominant cause of wasted cycles — there appears to be no dominant solution. It is thus unlikely that any single latency-tolerating technique will produce a dramatic increase in the performance of these programs if it only attacks specific types of latencies. Instruction scheduling targets several important segments of the wasted issue

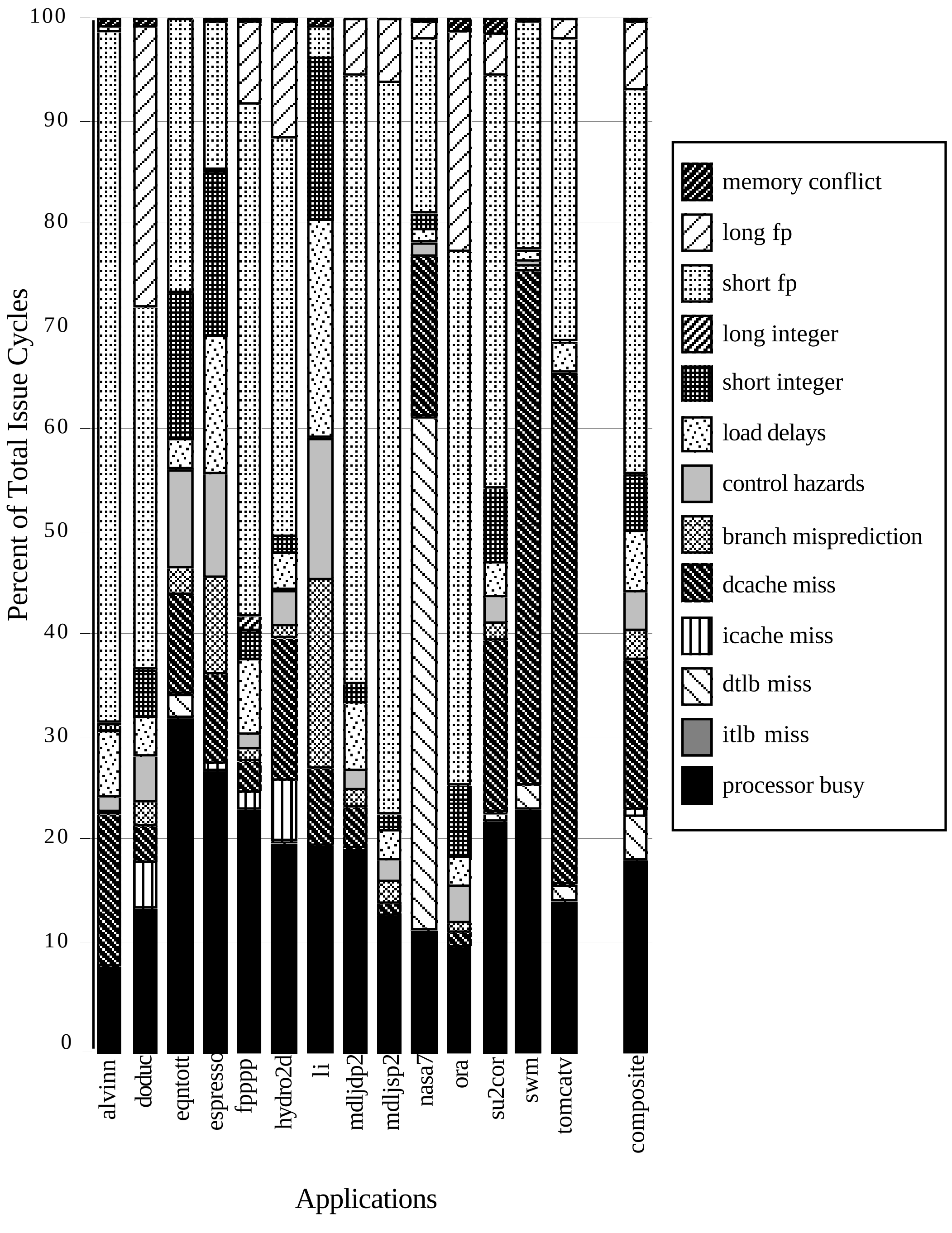


Figure 3.1: **Sources of all unused issue cycles in an 8-issue superscalar processor.** *Processor busy* **represents the utilized issue slots; all others represent wasted issue slots.**

bandwidth, but we expect that our compiler has already achieved most of the available gains in that regard. Current trends have been to devote increasingly larger amounts of on-chip area to caches, yet even if memory latencies are completely eliminated, we cannot achieve 40% utilization of this processor. If specific latency-hiding techniques have limited potential, then any dramatic increase in parallelism needs to come from a general latency-hiding solution, that is, a technique that can hide all sources of latency, regardless of cause. Multithreading is such a technique. In particular, fine-grain and simultaneous multithreading each have the potential to hide all sources of latency, but to different degrees.

This becomes clearer if we classify wasted cycles as either vertical waste (completely idle cycles) or horizontal waste (unused issue slots in a non-idle cycle), as shown previously in Figure 1.1(a). In our measurements, 61% of the wasted cycles are vertical waste, the remainder are horizontal waste. Traditional multithreading (coarse-grain or fine-grain) can fill cycles that contribute to vertical waste. Doing so, however, recovers only a fraction of the vertical waste; because of the inability of a single thread to completely fill the issue slots each cycle, traditional multithreading converts much of the vertical waste to horizontal waste, rather than eliminating it.

Simultaneous multithreading has the potential to recover all issue slots lost to *both* horizontal and vertical waste.

3.3 Summary

Processor utilization on the next generation of wide-issue superscalar processors will be low — we have shown less than 20% utilization of an 8-issue superscalar. This result was based on the SPEC benchmarks, a workload that produces relatively high processor utilization due to their high cache hit rates and branch prediction accuracy. Low utilization leaves a large number of execution resources idle each cycle.

The next chapter demonstrates how effectively simultaneous multithreading recovers those lost issue slots, and how it compares with some alternative architectural models.

Chapter 4

The Potential For Simultaneous Multithreading

This chapter introduces and demonstrates the potential for simultaneous multithreading (SMT), a technique that permits several independent threads to issue instructions to multiple functional units each cycle. In the most general case, the binding between thread and functional unit is completely dynamic. The objective of SMT is to substantially increase processor utilization in the face of both long memory latencies and limited available parallelism per thread. It combines the multiple-instruction-issue features of modern superscalar processors with the latency-hiding ability of multithreaded architectures. In this chapter, we (1) introduce several SMT models that potentially differ in terms of their implementation complexity, (2) evaluate the performance of those models relative to each other and to superscalar execution and fine-grain multithreading, (3) investigate some cache hierarchy design options for SMT processors, and (4) demonstrate the potential for performance and real-estate advantages of SMT architectures over small-scale, on-chip multiprocessors.

Current microprocessors employ various techniques to increase parallelism and processor utilization; however, each technique has its limits. For example, modern superscalars, such as the DEC Alpha 21164 [ER94], Intel Pentium Pro [Rep95], PowerPC 604 [Den94], MIPS R10000 [Rep94a], Sun UltraSparc [Rep94b], and HP PA-8000 [Rep94c], issue three to five instructions per cycle from a single thread. Multiple instruction issue has the potential to increase performance, but is ultimately limited by limited available parallelism within the single executing thread. The effects of these are demonstrated in Chapter 3. Multithreaded architectures, on the other hand, employ multiple threads with fast context switching between threads. Traditional multithreading hides memory and functional unit latencies and attacks vertical waste (see Figure 1.1). However, because it cannot hide horizontal waste, this technique is limited to the amount of parallelism that can be found in a single thread in a single cycle. As issue width increases, the ability of traditional multithreading to utilize processor resources will decrease.

This chapter evaluates the potential improvement, relative to wide superscalar architectures and conventional multithreaded architectures, of various simultaneous multithreading models. To place our evaluation in the context of modern superscalar processors, we simulate a base architecture derived from the 300 MHz Alpha 21164 [ER94], enhanced for wider superscalar execution; our SMT architectures are extensions of that basic design.

Our results show the limits of traditional multithreading to increase instruction throughput in future processors. We show that a fine-grain multithreaded processor (capable of switching contexts every cycle at no cost) utilizes only about 40% of a wide superscalar, regardless of the number of threads. Simultaneous multithreading, on the other hand, provides significant performance improvements in instruction throughput, and is only limited by the issue bandwidth of the processor. It raises processor utilization to nearly 80% in our experiments.

A more traditional means of achieving parallelism is the conventional multiprocessor. As chip densities increase, single-chip multiprocessors will become a viable design option [FKD^{+}95]. The simultaneous multithreaded processor and the single-chip multiprocessor are two close organizational alternatives for increasing on-chip instruction throughput. We compare these two approaches and show that simultaneous multithreading is potentially superior to multiprocessing in its ability to utilize processor resources. For example, a single simultaneous multithreaded processor with 10 functional units outperforms by 24% a conventional 8-processor multiprocessor with a total of 32 functional units, when they have equal issue bandwidth.

This chapter is organized as follows. Section 4.1 defines our simulation environment and the workloads that we measure. Sections 4.2 and 4.3 present the performance of a range of SMT architectures and compares them to the superscalar architecture and a fine-grain multithreaded processor. Section 4.4 explores the effect of cache design alternatives on the performance of simultaneous multithreading. Section 4.5 compares the SMT approach with conventional multiprocessor architectures. We summarize our results in Section 4.6.

4.1 Methodology

Our goal is to evaluate several architectural alternatives: wide superscalars, traditional multithreaded processors, simultaneous multithreaded processors, and small-scale multiple-issue multiprocessors. To do this, we have developed a simulation environment that defines an implementation of a simultaneous multithreaded architecture; that architecture is a straightforward extension of next-generation wide superscalar processors, running a real multiprogrammed workload that is highly optimized for single-threaded execution on our target machine. That simulator is then extended to model various flavors of multiple-issue, multiple context processors.

4.1.1 Simulation Environment

The simulator used for these results is adapted from the simulator described in Chapter 3. It simulates the same basic machine model described in that section, having the same 10 functional units, three-level cache hierarchy, TLBs, and branch prediction mechanisms.

For our multithreaded experiments, we assume support is added for up to eight hardware contexts. We support several models of simultaneous multithreaded execution, described in Section 4.2. In most of our experiments instructions are scheduled in a strict priority order, i.e., context 0 can schedule instructions onto any available functional unit, context 1 can schedule onto any unit unutilized by context 0, etc. Our experiments show that the overall instruction throughput of this scheme and a completely fair scheme are virtually identical for most of our execution models; only the relative speeds of the different threads change. The results from the priority scheme present us with some analytical advantages, as will be seen in Section 4.3, and the performance of the fair scheme can be extrapolated from the priority scheme results.

In this chapter, which evaluates the *potential* for simultaneous multithreading, we do not assume any changes to the basic pipeline to accommodate simultaneous multithreading. This gives us an upper bound (in terms of the pipeline) to performance. However, even if the pipeline assumptions are optimistic, the resulting inaccuracy will be quite small — in Chapter 5, where we pin down the low-level architecture, we add two pipeline stages to account for changes to support simultaneous multithreading and show that with single-thread execution the extra stages have less than a 2% impact on throughput. That effect is lower still for multithreaded execution.

4.1.2 Workload

Our workload is taken from the SPEC92 benchmark suite [Dix92]. To gauge the raw instruction throughput achievable by multithreaded superscalar processors, we chose uniprocessor applications, assigning a distinct program to each thread. This models a multithreaded workload achieved by multiprogramming rather than parallel processing. In this way, throughput results are not affected by synchronization delays, inefficient parallelization, etc. — effects that would make it more difficult to see the performance impact of simultaneous multithreading alone. Also, it presents us with a workload with no overlap in data or code working sets between threads, providing maximum contention for shared resources.

In the single-thread experiments described in the previous chapter, all of the benchmarks are run to completion using the default data set(s) specified by SPEC. The multithreaded experiments are more complex; to reduce the effect of benchmark difference, a single data point is composed of B runs, each T * 500 million instructions in length, where T is the number of threads and B is the number of benchmarks. Each of the B runs uses a different ordering of the benchmarks, such that each benchmark is run once in each priority position. To limit the number of permutations, we use a subset of the benchmarks equal to the maximum number of threads (8). The actual programs used are *alvinn, doduc, eqntott, espresso, fpppp, li, ora,* and *tomcatv,* five floating-point intensive and 3 integer programs. The simulation runs are long enough that some programs complete execution. When that happens the program begins again (possibly with another data set); this eliminates potential tail effects where the simulation would run for a period of time with less than the stated number of threads.

As in the previous chapter, we compile each program with the Multiflow trace scheduling compiler, modified to produce Alpha code optimized and scheduled for single-thread execution on our target machine. The applications were each compiled with several different compiler options; the executable with the lowest single-thread execution time on our target hardware was used for all experiments. By maximizing single-thread parallelism through our compilation system, we avoid overstating the increases in parallelism achieved with simultaneous multithreading.

4.2 Models Of Simultaneous Multithreading

The following models reflect several possible design choices for a combined multithreaded, superscalar processor. The models differ in how threads can use issue slots and functional units each cycle; in all cases, however, the basic machine is a wide superscalar with 10 functional units and capable of issuing 8 instructions per cycle (the same core machine as Section 3).

The models are:

- **Fine-Grain Multithreading.** Only one thread issues instructions each cycle, but it can use the entire issue width of the processor. This hides all sources of vertical waste, but does not hide horizontal waste. It is the only model that does not feature simultaneous multithreading. Among existing or proposed architectures, this is most similar to the Tera processor [ACC$^+$90], which issues one 3-operation LIW instruction per cycle.

- **SMT:Full Simultaneous Issue.** This is a completely flexible simultaneous multithreaded superscalar: all eight threads compete for each of the issue slots each cycle. This has the potential to be the most complex of the SMT models. The following models each represent restrictions to this scheme that decrease hardware complexity.

- **SMT:Single Issue, SMT:Dual Issue,** and **SMT:Four Issue**. These three models limit the number of instructions each thread can issue, or have active in the scheduling window, each cycle. For example, in a SMT:Dual Issue processor, each thread can issue a maximum of 2 instructions per cycle; therefore, a minimum of 4 threads would be required to fill the 8 issue slots in one cycle.

- **SMT:Limited Connection.** In this model, each hardware context is directly connected to exactly one of each type of functional unit. For example, if the hardware supports eight threads and there are four integer units, each integer unit could receive instructions from exactly two threads. The partitioning of functional units among threads is thus less dynamic than in the other models, but each functional unit is still shared (the critical factor in achieving high utilization). Since the choice of functional units available to a single thread is different than in our original target machine, we recompiled for a 4-issue (one of each type of functional unit) processor for this model.

Although each of these schemes is designed to limit the complexity of one or more components, the degree to which they succeed at that is very much determined by the specific architectural implementation. For example, Chapter 5 presents an architecture for the full simultaneous issue model that eliminates many of the implementation issues addressed by the other models. However, most proposed architectures that feature some form of simultaneous multithreading use per-thread instruction buffering, some kind of distributed instruction issuing mechanism, and distinct per-thread register files. All of those components could benefit from the reduced complexity of these SMT models. The following comments on implementation complexity, then, pertain to the more general body of proposed simultaneous multithreading architectures more than to the specific architecture proposed in the following chapter.

Many of these complexity issues are inherited from our wide superscalar design rather than from multithreading, per se. Even in the SMT:full simultaneous issue model, the inter-instruction dependence checking, the ports per register file, and the forwarding logic scale with the issue bandwidth and the number of functional units, rather than the number of threads. Our choice of ten functional units seems reasonable for an 8-issue processor. Current superscalar processors have between 4 and 9 functional units.

The following list summarizes the potential impact of the proposed machine models on some particular hardware components.

Register ports — The number of register ports per register file (assuming separate per-thread register files, rather than a single physically-shared register file) is a function of how many instructions a single thread can issue in a single cycle. It is highest (comparable to a single-threaded 8-issue superscalar) for the fine-grain model and SMT:full simultaneous issue, and is reduced for each of the other SMT models, with SMT:single issue being the least complex.

Inter-instruction dependence checking — Assuming that the determination of issuable instructions is distributed among threads, the number of instructions that have to be checked in parallel also grows with the number of instructions a single thread can issue in a cycle. This same factor also encompasses the number of instructions that would have to be decoded in parallel if we assume separate fetch units operating independently. In those cases, once again the complexity is highest for the fine-grain model and SMT:full simultaneous issue (but again comparable to a single-threaded 8-issue superscalar), and is reduced for each of the other SMT models.

Forwarding logic — The amount of forwarding (bypass) logic required is primarily a function of the number of functional units, which is a constant for all of the simulated models. The limited connection model, however, requires no forwarding between functional units of the same type, significantly reducing the total number of connections. There is nothing inherent in fine-grain multithreading that requires fewer or more forwarding connections; however, several proposed multithreaded architectures, such as the Tera, have no forwarding in their pipelines, assuming that high degrees of multithreading will hide all pipeline delays. This represents a different design philosophy than what we assume in this research.

Hardware Instruction scheduling — A distributed instruction scheduling mechanism that combines instructions from different threads at the point of issue will be significantly more complex than that of a single-threaded superscalar. Only the fine-grain model would eliminate this, as instructions could be scheduled completely within the one context that will have control of the issue mechanism the next cycle. The SMT:limited connection model reduces the distributed scheduling mechanism somewhat by lessening the degree of contention for individual functional units.

The fine-grain model is not necessarily the least complex of our multithreading models, only having an advantage in the instruction scheduling complexity. The SMT:single issue model is least complex in terms of register ports and inter-instruction dependence checking. The SMT:limited connection model is the only model that is less complex than SMT:full simultaneous issue in all the listed categories.

Other researchers' proposed simultaneous multithreading architectures (see Section 2.4) all fall into either the SMT:single issue or SMT:full simultaneous issue categories.

4.3 The Performance of the Simultaneous Multithreading Models

Figures 4.1 through 4.4 show the performance of the various models as a function of the number of threads. Figures 4.1, 4.2, and 4.3 show the throughput by thread for each of the models, and Figure 4.4 shows the total throughput for the six machine models. The total height of the bars in the first three figures represents the total instruction throughput for that configuration; the segments of each bar indicate the throughput component contributed by each thread.

The first two bar-graphs (Figures 4.1 and 4.2) highlight two interesting points in the

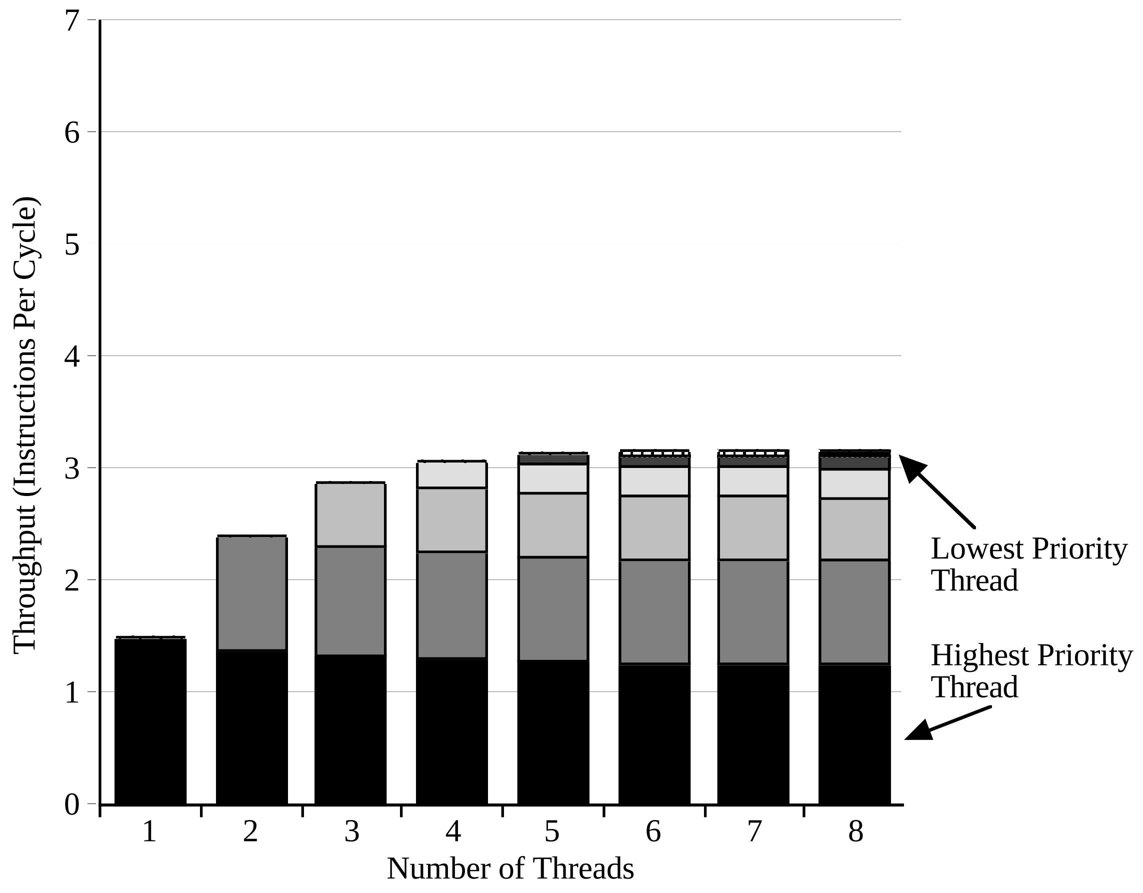


Figure 4.1: **Instruction throughput as a function of the number of threads with fine-grain multithreading. The lowest segment of each bar is the contribution of the highest priority thread to the total throughput.**

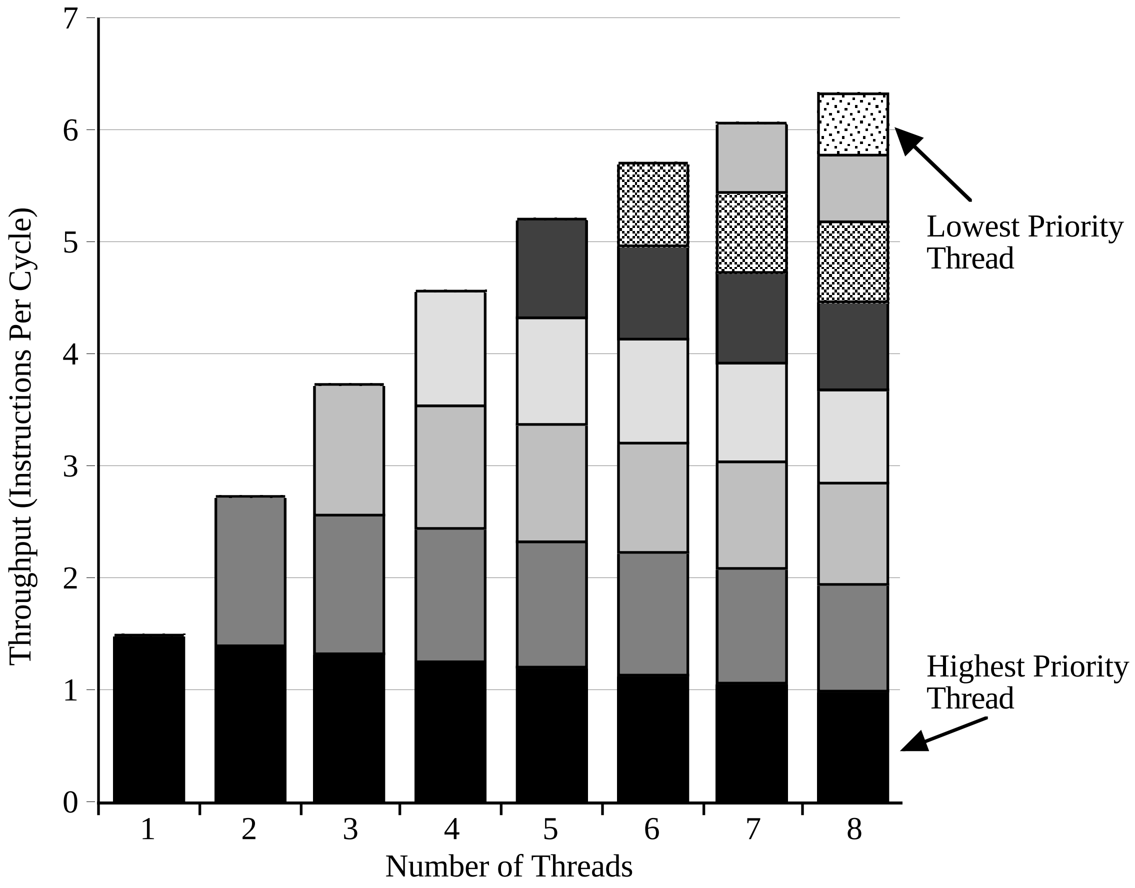


Figure 4.2: **Instruction throughput as a function of the number of threads with SMT:Full Simultaneous Issue.**

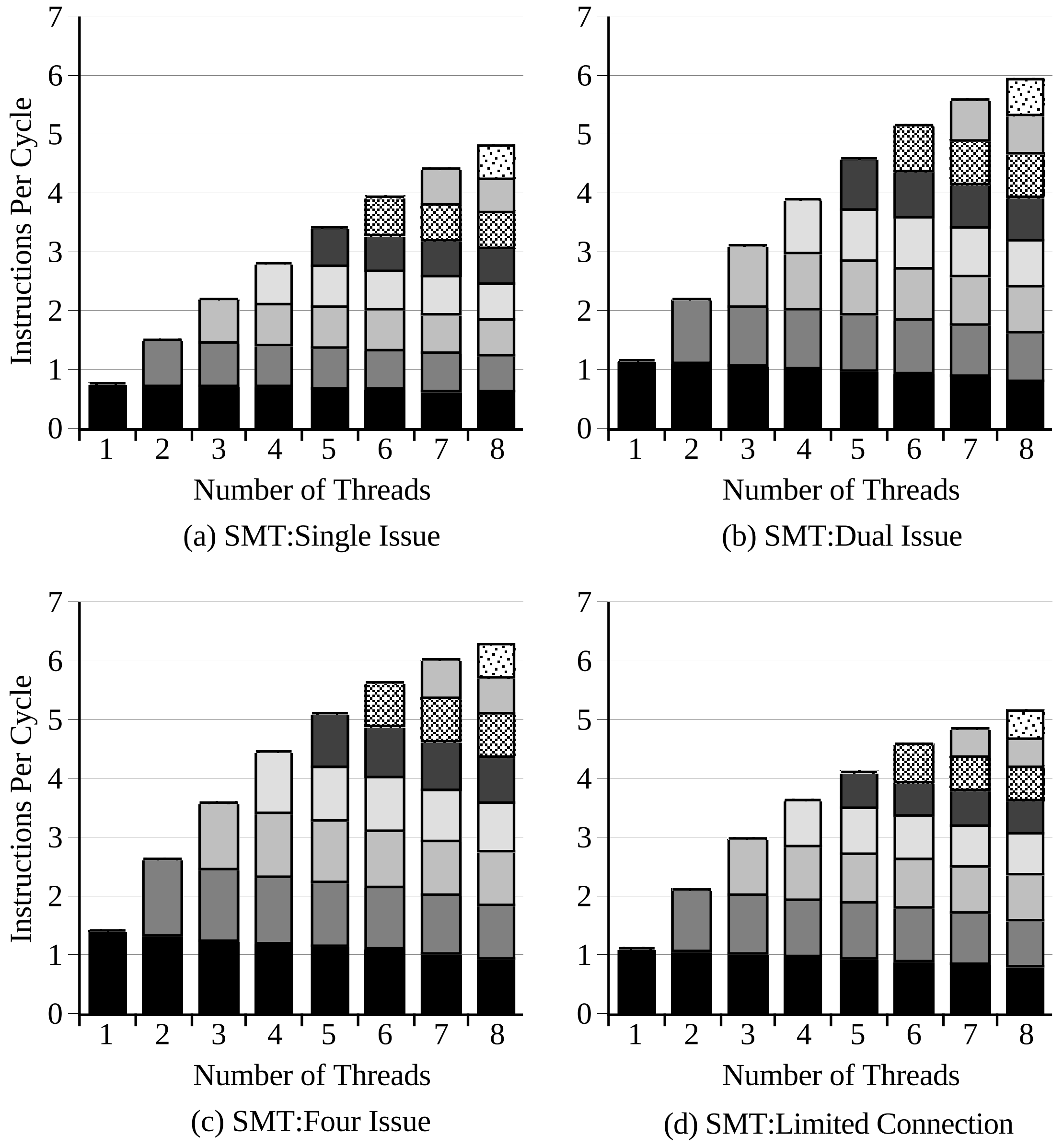


Figure 4.3: **Instruction throughput for the other four SMT models.**

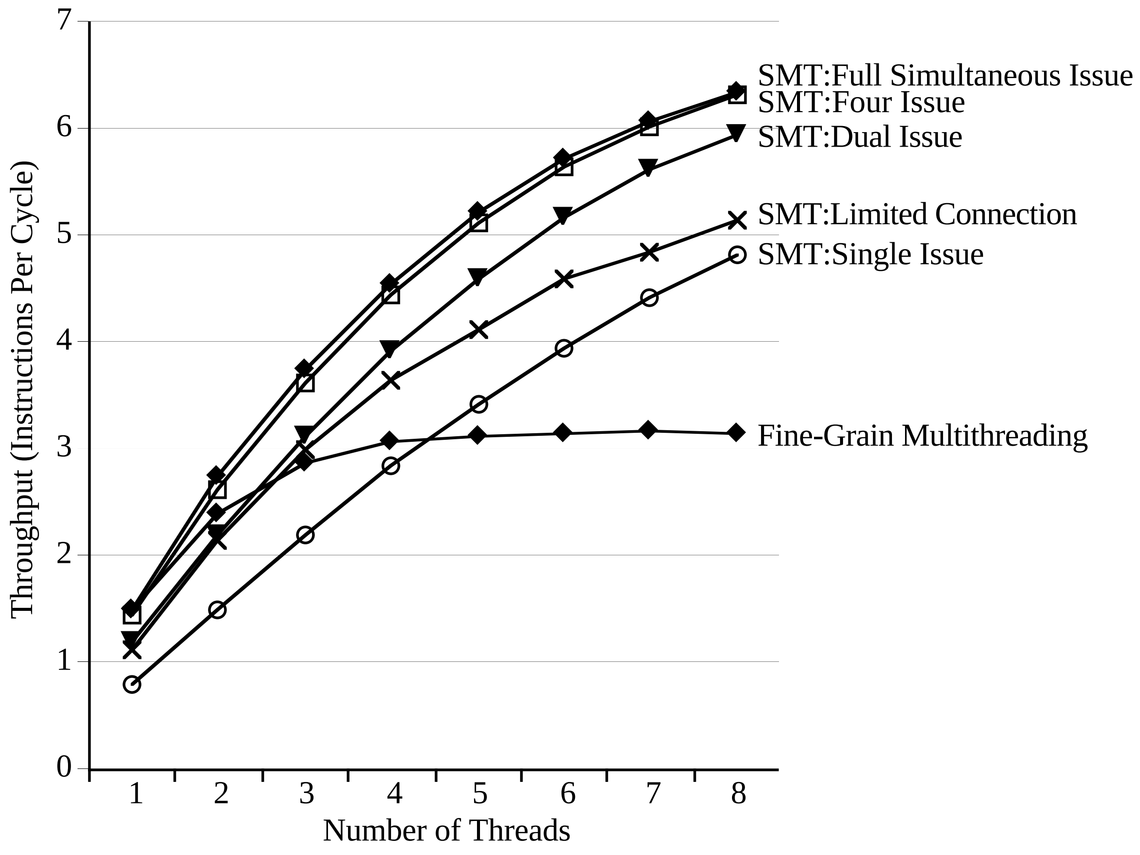


Figure 4.4: **Total instruction throughput as a function of the number of threads for each of the six machine models.**

multithreaded design space: fine-grained multithreading (only one thread per cycle, but that thread can use all issue slots) and SMT: Full Simultaneous Issue (many threads per cycle, any thread can potentially use any issue slot).

The fine-grain multithreaded architecture (Figure 4.1) provides a maximum speedup (increase in instruction throughput) of 2.1 over single-thread execution (from 1.5 IPC to 3.2). The graph shows that nearly all of this speedup is achieved with only four threads, and that after that point there is little advantage to adding more threads in the model. In fact, with four threads, the vertical waste has been reduced to less than 3%, which bounds any further gains beyond that point. This result is similar to previous studies [ALKK90, Aga92, LGH94, GHG+91, WG89, TE94] for both coarse-grain and fine-grain multithreading on *scalar* (single-issue) processors, which have concluded that multithreading is only beneficial for 2 to 5 threads. These limitations do not apply to simultaneous multithreading, however, because of its ability to exploit horizontal waste.

Figures 4.2 through 4.4 show the advantage of the simultaneous multithreading models, which achieve maximum speedups over single-thread superscalar execution ranging from 3.2 to 4.2, with an issue rate as high as 6.3 IPC. The speedups are calculated using the full simultaneous issue, 1-thread result to represent the single-threaded superscalar.

With SMT, it is not necessary for any single thread to be able to utilize the entire resources of the processor in order to get maximum or near-maximum performance. The four-issue model gets nearly the performance of the full simultaneous issue model, and even the dual-issue model is quite competitive, reaching 94% of full simultaneous issue at 8 threads. The limited connection model approaches full simultaneous issue more slowly due to its less flexible scheduling. Each of these models becomes increasingly competitive with full simultaneous issue as the ratio of threads to functional units increases.

With the results shown in Figure 4.4, we see the possibility of trading the number of hardware contexts against hardware complexity in other areas. For example, if we wish to execute around four instructions per cycle, we can build a four-issue or full simultaneous machine with 3 to 4 hardware contexts, a dual-issue machine with 4 contexts, a limited connection machine with 5 contexts, or a single-issue machine with 6 contexts. The Tera processor [ACC+90] is an extreme example of trading pipeline complexity for more contexts; it has no forwarding in its pipelines and no data caches, but supports 128 hardware contexts.

The increases in processor utilization are a direct result of threads dynamically sharing processor resources; however, sharing also has negative effects. The effects of two different types of sharing on the full simultaneous issue model can be seen in Figure 4.2. We see the effect of competition for issue slots and functional units in that the lowest priority thread (at 8 threads) runs at 55% of the speed of the highest priority thread. We also observe the impact of sharing other system resources (caches, TLBs, branch prediction table); with full simultaneous issue, the highest priority thread, which is fairly immune to competition for issue slots and functional units, degrades significantly as more threads are added (a 35% slowdown at 8 threads). Competition for non-execution resources, then, plays nearly as significant a role in this performance region as the competition for execution resources.

Others have observed that caches are more strained by a multithreaded workload than a single-thread workload, due to a decrease in locality [McC93, WG89, Aga92, TE94]. Our measurements pinpoint the exact areas where sharing degrades performance, in the caches and in other shared resources. Sharing the caches is the dominant effect, as the wasted issue cycles (from the perspective of the first thread) due to I cache misses grows from 1% at one thread to 14% at eight threads, while wasted cycles due to data cache misses grows from 12% to 18%. The data TLB waste also increases, from under 1% to 6%. In the next section, we will investigate the cache problem. For the data TLB, we found that, with our workload, increasing the shared data TLB from 64 to 96 entries brings the wasted cycles (with 8 threads) down to 1%, while providing private TLBs of 24 entries reduces it to under 2%, regardless of the number of threads.

In summary, our results show that simultaneous multithreading surpasses limits on the performance attainable through either single-thread execution or fine-grain multithreading, when run on a wide superscalar. We have also seen that simplified implementations of SMT with limited per-thread capabilities can still attain high instruction throughput. These improvements come without any significant tuning of the architecture for multithreaded execution; in fact, we have found that the instruction throughput of the various SMT models is somewhat hampered by the sharing of the caches and TLBs. The next section investigates designs that are more resistant to the cache effects.

4.4 Cache Design for a Simultaneous Multithreaded Processor

This section focuses on the organization of the first-level (L1) caches, comparing the use of private per-thread caches to shared caches for both instructions and data. (We assume that L2 and L3 caches are shared among all threads.) This comparison serves two purposes.

First, our measurements have shown a performance degradation (for individual threads) due to cache sharing in simultaneous multithreaded processors. We would expect private per-thread caches to provide more thread isolation (making the performance of one thread less sensitive to the presence of other threads), if that is a desired property.

Second, this comparison highlights a difference between simultaneous multithreading and single-chip multiprocessors, which is investigated further in Section 4.5. That section looks at the partitioning of functional units, while this section examines the partitioning of the cache. While both an SMT processor and an on-chip multiprocessor could be configured to either share or statically partition either the instruction or data caches, a multiprocessor will have distributed and private load/store units and fetch units, making private L1 caches more natural. An SMT processor will have shared load/store units, and either a shared instruction fetch mechanism (as in Chapter 5), or at least physically close fetch mechanisms, making shared caches the more obvious design choice.

The L1 caches in Figure 4.5 are all 64 KB total; for example, the *I shared/D private* configuration has a single shared I cache of 64 KB, while the data cache is eight private caches, each 8 KB in size and accessible by only a single thread each. Not all of the private caches will be utilized when fewer than eight threads are running. All experiments use the 4-issue model.

Figure 4.5 exposes several interesting properties for multithreaded caches. We see that shared caches optimize for a small number of threads (where the few threads can use all available cache), while private caches perform better with a large number of threads. For example, the *I shared/D shared* cache ranks first among all models at 1 thread and last at 8 threads, while the *I private/D private* cache gives nearly the opposite result. However, the tradeoffs are not the same for both instructions and data. A shared data cache outperforms a private data cache over all numbers of threads (e.g., compare *I private/D shared* with *I private/D private*), while instruction caches benefit from private caches at 8 threads. One reason for this is the differing access patterns between instructions and data. Private I caches eliminate conflicts between different threads in the I cache, while a shared D cache allows a single thread to issue multiple memory instructions to different banks.

Two configurations appear to be good choices. With little performance difference at 8 threads, the cost of optimizing for a small number of threads is small, making *I shared/D shared* an attractive option. However, if we expect to typically operate with all or most thread slots full, the *I private/D shared* gives the best performance in that region and is

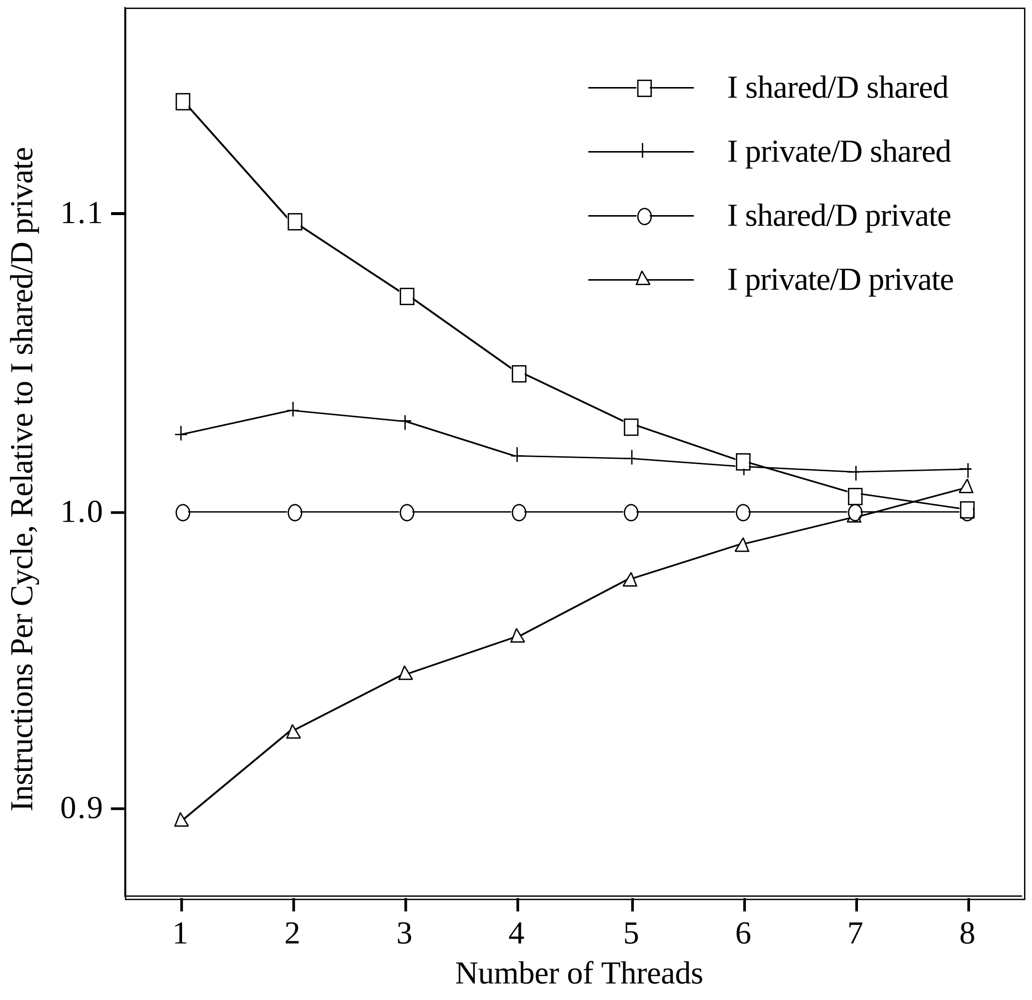


Figure 4.5: **Results for the simulated cache configurations, shown relative to the throughput (instructions per cycle) of the I shared/D private cache results.**

never less than the second best performer with fewer threads. The shared data cache in this scheme allows it to take advantage of more flexible cache partitioning, while the private instruction caches provide some thread isolation.

Private caches do indeed provide more consistent thread performance, as the performance of the highest priority thread only degrades by 18% with private caches when going from 1 thread to 8 in the system. With shared caches, the highest priority thread degrades by 33%. However, this thread isolation comes almost entirely from slowing down the 1-thread results rather than improving 8-thread performance, as the shared caches outperform private caches by 20% with only one thread.

The ease of sharing caches in a simultaneous multithreading processor, then, is a significant advantage relative to multiprocessors, which lend themselves much more easily to private caches. This advantage, particularly with the data cache, is seen to a small extent with many threads, where the increased flexibility of the shared data cache outperforms the static partitioning of the private caches. It is seen to a much greater extent with fewer threads where the shared cache is not wasted by unused thread slots. When running parallel workloads with data and code sharing, the advantages of the shared caches will be even greater [LEL$^+$96].

The next section examines the tradeoffs between simultaneous multithreading and multiprocessors further, focusing on the manner in which those architectures partition the functional units and issue bandwidth.

4.5 Simultaneous Multithreading Versus Single-Chip Multiprocessing

As chip densities continue to rise, single-chip multiprocessors will provide an obvious means of achieving parallelism with the available real estate. This section compares the performance of simultaneous multithreading to small-scale, single-chip multiprocessors (MP). On the organizational level, the two approaches are similar: both have multiple register sets, multiple functional units, and high issue bandwidth on a single chip. The key difference is in the way those resources are partitioned and scheduled: the multiprocessor statically partitions resources, devoting a fixed number of functional units to each thread; the SMT processor allows the partitioning to change every cycle. Clearly, scheduling is more complex for an SMT processor; however, we will show that in other areas the SMT model requires fewer resources, relative to multiprocessing, to achieve a desired level of performance.

For these experiments, we tried to choose SMT and MP configurations that are reasonably equivalent, although in several cases we biased in favor of the MP. Because of the differences in these machines and how they use resources, making all resources constant across the two architectures did not necessarily make for a fair comparison. For this reason, we ran a series of experiments, generally keeping all or most of the following equal: the number of register sets (i.e, the number of threads for SMT and the number of processors for MP), the total issue bandwidth, and the specific functional unit configuration. A consequence of the last item is that the functional unit configuration is often optimized for the multiprocessor and represents an inefficient configuration for simultaneous multithreading.

All experiments use 8 KB private instruction and data caches (per thread for SMT, per processor for MP), a 256 KB 4-way set-associative shared second-level cache, and a 2 MB direct-mapped third-level cache. Section 4.4 showed that there is an advantage to sharing caches, even with a multiprogrammed workload. In these comparisons, however, we want to keep the caches constant and focus on the execution resources. We chose to use private I and D caches, the most natural configuration for the multiprocessor, for all experiments.

We evaluate MPs with 1, 2, and 4 issues per cycle on each processor. We evaluate SMT processors with 4 and 8 issues per cycle; however we use the SMT:Four Issue model (defined in Section 4.2) for all of our SMT measurements (i.e., each thread is limited to four issues per cycle). Using this model minimizes some of the potential complexity differences between the SMT and MP architectures. For example, an SMT:Four Issue processor is similar to a single-threaded processor with 4 issues per cycle in terms of both the number of ports on each register file (assuming separate register files) and the amount of inter-instruction dependence checking. In each experiment we run the same version of the benchmarks for both configurations (compiled for a 4-issue, 4 functional unit processor, which most closely matches the MP configuration) on both the MP and SMT models; this typically favors the MP.

We must note that while in general we have tried to bias the tests in favor of the MP, the SMT results may be optimistic in two respects: the complexity of simultaneous multithreading itself, and the complexity of a wide superscalar processor as compared to multiple smaller processors. In this section we do not make any allowances for the complexity of simultaneous multithreading, either in the cycle time or length of the pipeline; however, in Chapter 5 we show that support for simultaneous multithreading need have little impact on cycle time, and the changes to the pipeline have less than a 2% performance effect. The second factor that these tests do not account for is the simplicity of the smaller processors in the multiprocessor. In these tests, we typically compare a wide-issue SMT processor (e.g., 8-issue) with several smaller (e.g., 4-issue or 1-issue) processors in the multiprocessor. The reduced complexity of the forwarding logic, instruction decode, and instruction scheduling logic should give a cycle time advantage to the multiprocessor; however, there are too many factors involved, and no clear case studies available, to be able to quantify that advantage. This prevents us from making raw throughput conclusions from these tests, but not from making conclusions about instruction throughput, resource utilization, and total execution resource needs (per cycle).

In these tests, the multiprocessor always has one functional unit of each type per processor. In most cases the SMT processor has the same total number of each FU type as the MP. The exception is Test D, where the SMT processor uses the same 10 functional units used for the tests in Section 4.3 (4 integer, 3 load/store, 2 floating point, 1 branch).

Figure 4.6 shows the results of our SMT/MP comparison for various configurations. In that figure, the resources listed typically represent the total for the machine (i.e., the sum for all of the processors on the multiprocessor). Tests A, B, and C compare the performance of the two schemes with an essentially unlimited number of functional units (FUs); i.e., there is a functional unit of each type available to every issue slot. The number of register sets and total issue bandwidth are constant for each experiment, e.g., in Test C, a 4 thread, 8-issue SMT and a 4-processor, 2-issue-per-processor MP both have 4 register sets and issue up to 8 instructions per cycle. In these models, the ratio of functional units (and threads) to issue bandwidth is high, so both configurations should be able to utilize most of their issue bandwidth. Simultaneous multithreading, however, does so more effectively, providing 21-29% higher throughput.

Test D repeats test A but limits the SMT processor to a more reasonable configuration, the same 10 functional unit configuration used throughout this chapter. This configuration outperforms the multiprocessor by nearly as much as test A, even though the SMT configuration has 1/3 the number of functional units.

In tests E and F, the MP is allowed a much larger total issue bandwidth. In test E, each MP processor can issue 4 instructions per cycle for a total issue bandwidth of 32 across the 8 processors; each SMT thread can also issue 4 instructions per cycle, but the 8 threads share only 8 issue slots. The results are similar despite the disparity in issue slots. In test F, the 4-thread, 8-issue SMT slightly outperforms a 4-processor, 4-issue per processor MP,

Purpose of Test	Common Elements	Specific Configuration	Throughput (instructions/cycle)
Unlimited FUs: equal total issue bandwidth, equal number of register sets (processors or threads)	**Test A**: 32 FUs Issue bw = 8 Reg sets = 8	SMT: 8 thread, 8-issue	6.64
		MP: 8 1-issue procs	5.13
	Test B: 16 FUs Issue bw = 4 Reg sets = 4	SMT: 4 thread, 4-issue	3.40
		MP: 4 1-issue procs	2.77
	Test C: 16 FUs Issue bw = 8 Reg sets = 4	SMT: 4 thread, 8-issue	4.15
		MP: 4 2-issue procs	3.44
Unlimited FUs: Test A, but limit SMT to 10 FUs	**Test D**: Issue bw = 8 Reg sets = 8	SMT: 8 threads, 10 FU	6.36
		MP: 8 1-iss procs, 32 FU	5.13
Unequal Issue BW: MP has up to four times the total issue bandwidth	**Test E**: FUs = 32 Reg sets = 8	SMT: 8 thread, 8-issue	6.64
		MP: 8 4-issue procs	6.35
	Test F: FUs = 16 Reg sets = 4	SMT: 4 thread, 8-issue	4.15
		MP: 4 4-issue procs	3.72
FU Utilization: equal FUs, equal issue bw, unequal reg sets	**Test G**: FUs = 8 Issue BW = 8	SMT: 8 thread, 8-issue	5.30
		MP: 2 4-issue procs	1.94

Figure 4.6: **Results for the various multiprocessor vs. simultaneous multithreading comparisons.**

which has twice the total issue bandwidth. Simultaneous multithreading performs well in these tests, despite its handicap, because the MP is constrained with respect to which 4 instructions a single processor can issue in a single cycle.

Test G shows the greater ability of SMT to utilize a fixed number of functional units. Here both SMT and MP have 8 functional units and 8 issues per cycle. However, while the SMT is allowed to have 8 contexts (8 register sets), the MP is limited to two processors (2 register sets) because each processor must have at least 1 of each of the 4 functional unit types. Simultaneous multithreading's ability to drive up the utilization of a fixed number of functional units through the addition of thread contexts achieves more than 2.5 times the throughput. Although this test is certainly not fair in all respects, it will in general be much easier to add contexts to a simultaneous multithreaded processor than to add processors to a multiprocessor, making this comparison reasonable.

These comparisons show that simultaneous multithreading outperforms single-chip multiprocessing in a variety of configurations because of the dynamic partitioning of functional units. More important, SMT requires many fewer resources (functional units and instruction issue slots) to achieve a given performance level. For example, a single 8-thread, 8-issue SMT processor with 10 functional units is 24% faster than the 8-processor, single-issue MP (Test D), which has identical issue bandwidth but requires 32 functional units; to equal the throughput of that 8-thread 8-issue SMT, an MP system requires eight 4-issue processors (Test E), which consume three times the number of functional units and four times the required issue bandwidth of the SMT model.

As mentioned previously, these results do not include any cycle time advantage gained by having a multiprocessor composed of simpler processors. However, there are further advantages of SMT over MP that are not shown by these experiments:

- Performance with few threads — These results show only the performance at maximum utilization. The advantage of SMT (over MP) is greater as some of the contexts (processors) become unutilized. An idle processor leaves 1/p of an MP idle, while with SMT, the other threads can expand to use the available resources. This is important when (1) we run parallel code where the degree of parallelism varies over time, (2) the performance of a small number of threads is important in the target environment, or (3) the workload is sized for the exact size of the machine (e.g., 8 threads). In the last case, a processor and all of its resources is lost when a thread experiences a latency orders of magnitude larger than what we have simulated (e.g., I/O).

- Granularity and flexibility of design — Our configuration options are much richer with SMT because the units of design have finer granularity. That is, with a multiprocessor, we would typically add hardware in units of entire processors (or add the same set of components to every processor). With simultaneous multithreading, we can benefit from the addition of a single resource, such as a functional unit, a register context, or an instruction issue slot; furthermore, all threads would be able to share in using that resource. Our comparisons did not take advantage of this flexibility. Processor designers, taking full advantage of the configurability of simultaneous multithreading, should be able to construct configurations that even further out-distance multiprocessing.

For these reasons, as well as the performance and complexity results shown, we believe that when component densities permit us to put multiple hardware contexts and wide issue bandwidth on a single chip, simultaneous multithreading will represent the most efficient organization of those resources.

4.6 Summary

This chapter evaluated the potential of simultaneous multithreading, a technique that allows independent threads to issue instructions to multiple functional units in a single cycle. Simultaneous multithreading combines facilities available in both superscalar and multithreaded architectures. We have presented several models of simultaneous multi-threading and compared them with wide superscalar, fine-grain multithreaded, and single-chip, multiple-issue multiprocessing architectures. Our evaluation used execution-driven simulation based on a model extended from the DEC Alpha 21164, running a multipro-grammed workload composed of SPEC benchmarks, compiled for our architecture with the Multiflow trace scheduling compiler.

Our results show the benefits of simultaneous multithreading when compared to the other architectures, namely:

1. Given our model, a simultaneous multithreaded architecture, properly configured, has the potential to achieve 4 times the instruction throughput of a single-threaded

wide superscalar processor with the same issue width (8 instructions per cycle, in our experiments).

2. While fine-grain multithreading (i.e., switching to a new thread every cycle) helps close the gap, the simultaneous multithreading architecture still outperforms fine-grain multithreading by a factor of 2. This is due to the inability of fine-grain multithreading to utilize issue slots lost due to horizontal waste.

3. A simultaneous multithreaded architecture provides superior throughput to a multiple-issue multiprocessor, given the same total number of register sets and functional units. Moreover, achieving a specific performance goal requires fewer hardware execution resources with simultaneous multithreading.

The advantage of simultaneous multithreading, compared to the other approaches, is its ability to boost utilization by dynamically scheduling functional units among multiple threads. SMT also increases hardware design flexibility; a simultaneous multithreaded architecture can tradeoff functional units, register sets, and issue bandwidth to achieve better performance, and can add resources in a fine-grained manner.

Simultaneous multithreading could potentially increase the complexity of instruction scheduling relative to superscalars, and causes shared resource contention, particularly in the memory subsystem. However, we have shown how simplified models of simultaneous multithreading reach nearly the performance of the most general SMT model with complexity in key areas commensurate with that of current superscalars; and we have modeled the effects of shared resource contention, showing that they do not inhibit significant performance gains.

The next chapter addresses the complexity of simultaneous multithreading in much more detail, demonstrating that even the most complex of our SMT models represents a reasonable architectural alternative for next-generation superscalar processors.

Chapter 5

Exploiting Choice: A Simultaneous Multithreading Processor Architecture

In this chapter, we show that the performance advantages of simultaneous multithreading demonstrated in the previous chapter are available without extensive changes to a conventional superscalar design. That chapter showed the potential of an SMT processor to achieve significantly higher throughput than either a wide superscalar or a traditional multithreaded processor. It also demonstrated the advantages of simultaneous multithreading over multiple processors on a single chip, due to SMT's ability to dynamically assign execution resources where needed each cycle.

Those results showed SMT's potential based on a somewhat idealized model. This chapter extends that work in four significant ways. First, we propose an architecture that is more comprehensive, realistic, and heavily leveraged off existing superscalar technology, which we use to demonstrate that the changes necessary to enable simultaneous multithreading on a conventional, wide-issue superscalar are small. Our simulations show that a minimal implementation of simultaneous multithreading achieves throughput 1.8 times that of the unmodified superscalar; small tuning of this architecture increases that gain to 2.5 (reaching throughput as high as 5.4 instructions per cycle). Second, we show that SMT need not compromise single-thread performance. Third, we use our more detailed architectural model to analyze and relieve bottlenecks that did not exist in the more idealized model. Fourth, we show how simultaneous multithreading creates an advantage previously unexploitable in other architectures: namely, the ability to choose the "best" instructions, from all threads, for both fetch and issue each cycle. By favoring the threads that will use the processor most efficiently, we can boost the throughput of our limited resources.

We present several simple heuristics for this selection process, and demonstrate how such heuristics, when applied to the fetch mechanism, can increase throughput by as much as 37%.

This chapter is organized as follows. Section 5.1 presents our baseline simultaneous multithreading architecture, comparing it with existing superscalar technology. Section 5.2 describes our simulator and our workload, and Section 5.3 shows the performance of the baseline architecture. In Section 5.4, we examine the instruction fetch process, present several heuristics for improving it based on intelligent instruction selection, and give performance results to differentiate those heuristics. Section 5.5 examines the instruction issue process in a similar way. We then use the best designs chosen from our fetch and issue studies in Section 5.6 as a basis to discover bottlenecks for further performance improvement. We summarize these results in Section 5.7.

5.1 The Architecture

Our SMT architecture is derived from a high-performance, out-of-order, superscalar architecture (Figure 5.1, without the extra program counters) which represents a projection of current superscalar design trends 3-5 years into the future. This superscalar processor fetches up to eight instructions per cycle; fetching is controlled by a conventional system of branch target buffer, branch prediction, and subroutine return stacks. Fetched instructions are then decoded and passed to the register renaming logic, which maps logical registers onto a pool of physical registers, removing false dependences. Instructions are then placed in one of two 32-entry instruction queues. Those instruction queues are similar to those used by the MIPS R10000 [Rep94a] and the HP PA-8000 [Rep94c], in this case holding instructions until they are issued. Instructions are issued to the functional units out-of-order when their operands are available. After completing execution, instructions are retired in-order, freeing physical registers that are no longer needed.

Our SMT architecture is a straightforward extension to this conventional superscalar design. We made changes only when necessary to enable simultaneous multithreading, and in general structures were not replicated or resized to support SMT or a multithreaded workload. Thus, nearly all hardware resources remain completely available even when there is only a single thread in the system. The changes necessary to support simultaneous multithreading on that architecture are:

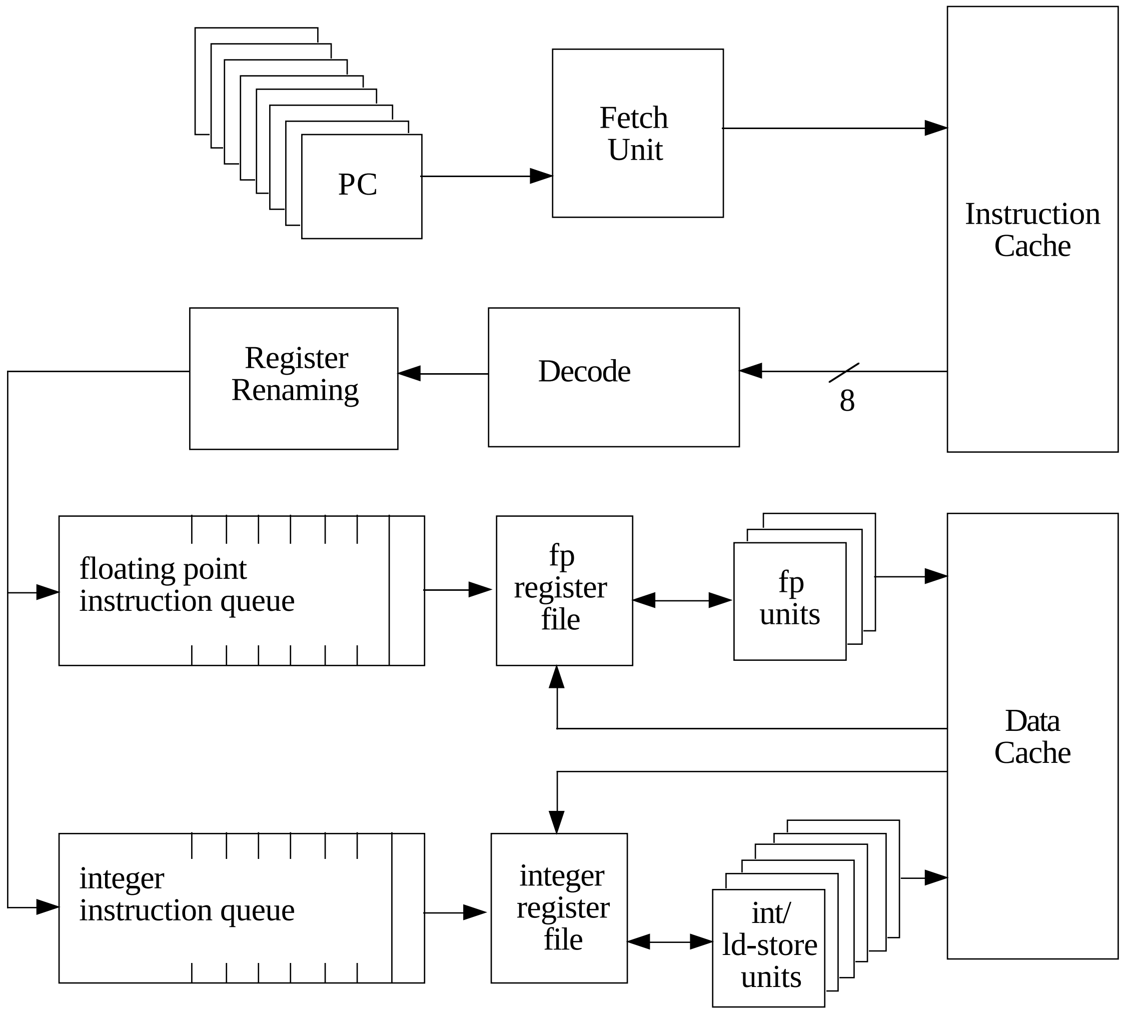


Figure 5.1: **Our base simultaneous multithreading hardware architecture.**

- multiple program counters and some mechanism by which the fetch unit selects one each cycle,
- a separate return stack for each thread for predicting subroutine return destinations,
- per-thread instruction retirement, instruction queue flush, and trap mechanisms,
- a thread id with each branch target buffer entry to avoid predicting phantom branches,
- tags on TLB entries containing a thread id or process id, if they are not already there, and
- a larger register file, to support logical registers for all threads plus additional registers for register renaming. The size of the register file affects the pipeline (we add two extra stages) and the scheduling of load-dependent instructions, which we discuss later in this section.

Noticeably absent from this list is a mechanism to enable simultaneous multithreaded scheduling of instructions onto the functional units. Because any artificial dependences between instructions from different threads are removed by the register renaming phase, a conventional instruction queue (IQ) designed for dynamic scheduling contains all of the functionality necessary for simultaneous multithreading. These queues monitor the availability of register operands for instructions in the queue. Our shared queues similarly issue an instruction when its operands are available, without regard to which thread the instruction belongs to.

In our baseline architecture, we fetch from one program counter (PC) each cycle. The PC is chosen, in round-robin order, from among those threads not already experiencing an I cache miss. This scheme provides simultaneous multithreading at the point of issue, but only fine-grain multithreading of the fetch unit. It requires no changes to a physically-addressed I cache. We will look in Section 5.4 at ways to extend simultaneous multithreading to the fetch unit. We also investigate alternative thread priority mechanisms for fetching.

The most significant impact of multithreading on our architecture will most likely result from the size of the register file. We have a single register file, as thread-specific logical registers are mapped onto a completely shared physical register file by register renaming. To support eight threads, we need a minimum of 8*32 = 256 physical integer registers (for a 32-register instruction set architecture), plus more to enable register renaming. Access to such a large register file will be slow.

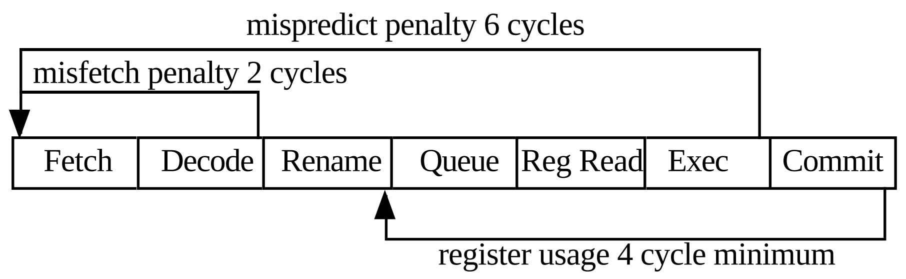
To account for the size of the register file without sacrificing processor cycle time, we modify the pipeline to allow two cycles to read registers instead of one. In the first cycle values are read into a buffer closer to the functional units. The instruction is sent to a similar buffer at the same time. The next cycle the data is sent to a functional unit for execution. Writes to the register file are treated similarly, requiring an extra *register write* stage. Figure 5.2 shows the pipeline modified for two-phase register access, compared to the pipeline of the original superscalar.

The two-stage register access has several ramifications on our architecture. First, it increases the pipeline distance between *fetch* and *exec*, increasing the branch misprediction penalty by 1 cycle. Second, it takes an extra cycle to write back results, requiring an extra level of bypass logic. Third, increasing the distance between *queue* and *exec* increases the period during which wrong-path instructions remain in the pipeline after a misprediction is discovered (the misqueue penalty in Figure 5.2). Wrong-path instructions are those instructions brought into the processor as a result of a mispredicted branch. Those instructions consume instruction queue slots, renaming registers and possibly issue slots, all of which, on an SMT processor, could be used by other threads.

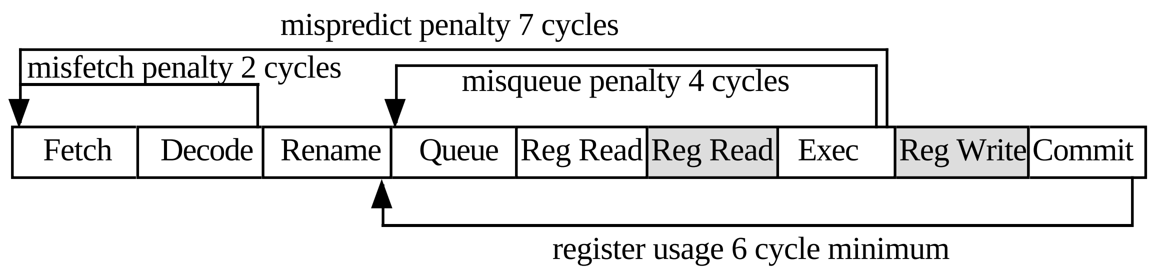
This pipeline does not increase the inter-instruction latency between most instructions. Dependent (single-cycle latency) instructions can still be issued on consecutive cycles, for example, as long as inter-instruction latencies are predetermined. That is the case for all instructions but loads. Since we are scheduling instructions a cycle earlier (relative to the *exec* cycle), cache-hit information is available a cycle later. Rather than increase load-hit latency by one cycle (to two cycles), we schedule load-dependent instructions assuming a 1-cycle data latency, but squash those instructions in the case of an L1 cache miss or a bank conflict. There are two performance costs to this solution, which we call *optimistic issue*. Optimistically issued instructions that get squashed waste issue slots, and optimistic instructions must be held in the IQ an extra cycle after they are issued, until it is known that they won't be squashed.

The last implication of the two-phase register access is that there are two more pipeline stages between *rename* and *commit*, thus increasing the minimum time that a physical register is held by an in-flight instruction. This increases the pressure on the register pool.

We assume, for each machine size, enough physical registers to support all active threads,


mispredict penalty 6 cycles
misfetch penalty 2 cycles
Fetch
Decode
Rename
Queue
Reg Read
Exec
Commit
register usage 4 cycle minimum

(a) Superscalar pipeline


mispredict penalty 7 cycles
misfetch penalty 2 cycles
misqueue penalty 4 cycles
Fetch
Decode
Rename
Queue
Reg Read
Reg Read
Exec
Reg Write
Commit
register usage 6 cycle minimum

(b) Pipeline modified for SMT's 2-cycle register access.

Figure 5.2: **The pipeline of (a) a conventional superscalar processor and (b) that pipeline modified for an SMT processor, along with some implications of those pipelines.**

plus 100 more registers to enable renaming, both for the integer file and the floating point file; i.e., for the single-thread results, we model 132 physical integer registers, and for an 8-thread machine, 356. We expect that in the 3-5 year time-frame, the scheme we have described will remove register file access from the critical path for a 4-hardware-context machine, but 8 contexts (threads) will still be a significant challenge. Nonetheless, extending our results to an 8-thread machine allows us to see trends beyond the 4-thread numbers and anticipates other solutions to this problem.

This architecture allows us to address several concerns about simultaneous multithreaded processor design. In particular, this chapter shows that:

- Instruction scheduling is no more complex on an SMT processor than on a dynamically scheduled superscalar.

- Register file data paths are no more complex than in the superscalar, and the performance implications of the register file and its extended pipeline are small.

- The required instruction fetch throughput is attainable, even without any increase in fetch bandwidth.

- Unmodified (for an SMT workload) cache and branch prediction structures do not thrash on that workload.

- Even aggressive superscalar technologies, such as dynamic scheduling and speculative execution, are not sufficient to take full advantage of a wide-issue processor without simultaneous multithreading.

We have only presented an outline of the hardware architecture to this point; the next section provides more detail.

5.1.1 Hardware Details

These simulations use a somewhat different functional unit configuration than the previous chapter. It is more consistent with the Alpha 21164 division of functional units and with the way we have divided instructions among the two instruction queues. The processor contains 3 floating point functional units and 6 integer units; four of the six integer units also execute loads and stores, and all functional units can execute branches.

The peak issue bandwidth out of the two instruction queues is therefore nine; however, the throughput of the machine is bounded by the peak fetch and decode bandwidths, which are eight instructions per cycle. For this reason, we will refer to this as an 8-issue machine. We assume that all functional units are completely pipelined. Instruction latencies, which are derived from the Alpha 21164 [ER94], are the same as used in previous chapters, given in Table 3.1.

We assume a 32-entry integer instruction queue (which handles integer instructions and all load/store operations) and a 32-entry floating point queue, not significantly larger than the HP PA-8000 [Rep94c], which has two 28-entry queues.

The caches (Table 5.1) are multi-ported by interleaving them into banks, similar to the design of Sohi and Franklin [SF91]. We model lockup-free caches and TLBs. The instruction TLB has 48 fully associative entries and the data TLB has 64 fully associative entries. TLB misses require two full memory accesses and no execution resources. To address the concern that memory throughput could be a limiting condition for simultaneous multithreading, we model the memory subsystem in great detail, simulating bandwidth limitations and access conflicts at multiple levels of the hierarchy, including buses between caches. The cache configuration is slightly different than in the previous chapters, reflecting changes in our expectations for what is reasonable in the appropriate time frame.

Table 5.1: **Details of the cache hierarchy**

	ICache	**DCache**	**L2**	**L3**
Size	32 KB	32 KB	256 KB	2 MB
Associativity	DM	DM	4-way	DM
Line Size	64	64	64	64
Banks	8	8	8	1
Transfer time	1 cycle	1	1	4
Accesses/cycle	var (1-4)	4	1	1/4
Cache fill time	2 cycles	2	2	8
Latency to next level	6	6	12	62

Each cycle, one thread is given control of the fetch unit, chosen from among those not stalled for an instruction cache miss. If we fetch from multiple threads, we never attempt to fetch from threads that conflict (on an I cache bank) with each other, although they may conflict with other I cache activity (cache fills).

Branch prediction is more aggressive than that used in the previous chapters, using a decoupled branch target buffer (BTB) and pattern history table (PHT) scheme [CG94]. We use a 256-entry BTB, organized as four-way set associative. The 2K x 2-bit PHT is accessed by the XOR of the lower bits of the address and the global history register [McF93, YP92]. Hiley and Seznec [HS96] confirm that this scheme performs well for both single-threaded and multithreaded workloads. Return destinations are predicted with a 12-entry return stack (per context).

We assume an efficient, but not perfect, implementation of dynamic memory disambiguation. This is emulated by using only part of the address (10 bits) to disambiguate memory references, so that it is occasionally over-conservative.

5.2 Methodology

The methodology in this chapter closely follows the simulation and measurement methodology of Chapter 4. Our simulator uses emulation-based, instruction-level simulation, executes unmodified Alpha object code and models the execution pipelines, memory hierarchy, TLBs, and the branch prediction logic of the processor described in Section 5.1.

In an SMT processor a branch misprediction introduces wrong-path instructions that interact with instructions from other threads. To model this behavior, we fetch down wrong paths, introduce those instructions into the instruction queues, track their dependences, and issue them. We eventually squash all wrong-path instructions a cycle after a branch misprediction is discovered in the *exec* stage. Our throughput results only count useful instructions.

Our workload comes primarily from the SPEC92 benchmark suite [Dix92]. We use five floating point programs (alvinn, doduc, fpppp, ora, and tomcatv) and two integer programs (espresso and xlisp) from that suite, and the document typesetting program TeX. We added the TeX application to the working set for this chapter to place more stress on the caches and branch prediction mechanism. We assign a distinct program to each thread in the processor. This multiprogrammed workload stresses our architecture more than a parallel program. Because there is no overlap in data or instruction working sets with this workload, we maximize contention for cache, TLB, and branch prediction structures. To eliminate the effects of benchmark differences, a single data point is composed of 8 runs, each T * 300 million instructions in length, where T is the number of threads. Each of the 8 runs uses a different combination of the benchmarks, similar to the methodology of Chapter 4.

We compile each program with the Multiflow trace scheduling compiler [LFK$^+$93], modified to produce Alpha code. In contrast to previous chapters, we here turn off trace scheduling in the compiler for this study, for two reasons. In our measurements, we want to differentiate between useful and useless speculative instructions, which is easy with hardware speculation, but not possible for software speculation with our system. Also, software speculation is not as beneficial on an architecture which features hardware speculation, and in some cases is harmful. However, the Multiflow compiler is still a good choice for our compilation engine, because of the high quality of the instruction scheduling and other optimizations, as well as the ease with which the machine model can be changed. The benchmarks are compiled to optimize single-thread performance on the base hardware.

5.3 Performance of the Base Hardware Design

In this section we examine the performance of the base architecture and identify opportunities for improvement. Figure 5.3 shows that with only a single thread running on our SMT architecture (the leftmost point), the throughput is less than 2% below a superscalar without SMT support. The drop in throughput is due to the longer pipeline (described in Section 5.1) used by the SMT processor. The peak throughput is 84% higher than the superscalar. This gain is achieved with virtually no tuning of the base architecture for simultaneous multithreading. This design combines low single-thread impact with high speedup for even a few threads, enabling simultaneous multithreading to reap benefits even in an environment where multiple processes are running only a small fraction of the time. We also note, however, that the throughput peaks before 8 threads, and the processor utilization is less than 50% of the 8-issue processor. Both of these characteristics fall short of the potential shown in Chapter 4, where a somewhat more idealized model is used.

We make several conclusions about the potential bottlenecks of this system as we approach 8 threads, aided by Figure 5.3 and Table 5.2. Issue bandwidth is clearly not a bottleneck, as the throughput represents a fraction of available issue bandwidth, and our data shows that no functional unit type is being overloaded (the fp units are 34% utilized, the integer units are 52% utilized, and the load-store units are 37% utilized with 8 threads).

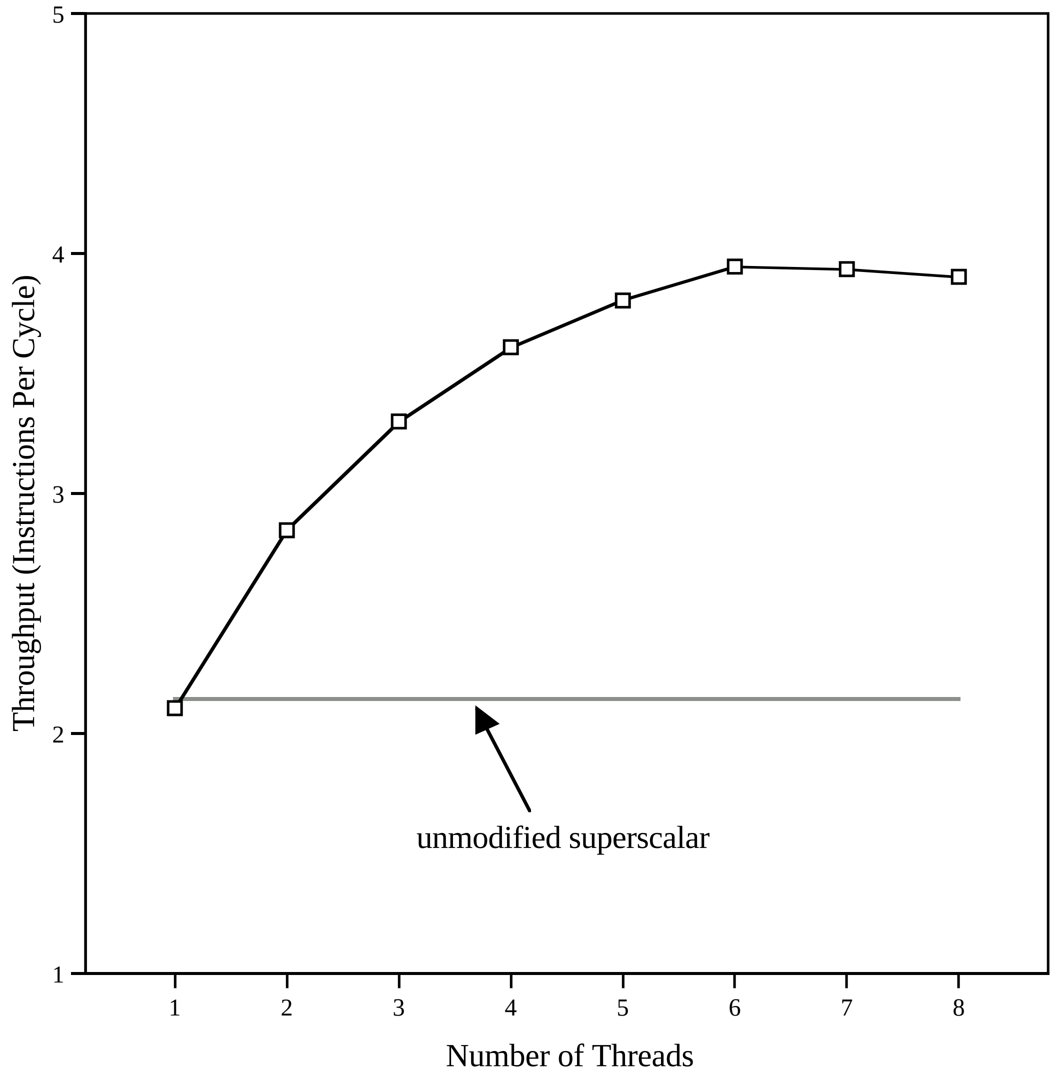


Figure 5.3: **Instruction throughput for the base hardware architecture.**

Table 5.2: **The result of increased multithreading on some low-level metrics for the base architecture.**

	Number of Threads		
Metric	1	4	8
out-of-registers (% of cycles)	3%	7%	3%
I cache miss rate	2.5%	7.8%	14.1%
-misses per thousand instructions	6	17	29
D cache miss rate	3.1%	6.5%	11.3%
-misses per thousand instructions	12	25	43
L2 cache miss rate	17.6%	15.0%	12.5%
-misses per thousand instructions	3	5	9
L3 cache miss rate	55.1%	33.6%	45.4%
-misses per thousand instructions	1	3	4
branch misprediction rate	5.0%	7.4%	9.1%
jump misprediction rate	2.2%	6.4%	12.9%
integer IQ-full (% of cycles)	7%	10%	9%
fp IQ-full (% of cycles)	14%	9%	3%
avg (combined) queue population	25	25	27
wrong-path instructions fetched	24%	7%	7%
wrong-path instructions issued	9%	4%	3%

We appear to have enough physical registers. The caches and branch prediction logic are being stressed more heavily at 8 threads, but we expect the latency-hiding potential of the additional threads to make up for those drops. The culprit appears to be one or more of the following three problems: (1) IQ size — IQ-full conditions are common, 12 to 21% of cycles total for the two queues; (2) fetch throughput — even in those cycles where we don't experience an IQ-full condition, our data shows that we are sustaining only 4.2 useful instructions fetched per cycle (4.5 including wrong-path) with 8 threads; and (3) lack of parallelism — although the queues are reasonably full, we find fewer than four out of, on average, 27 instructions (the average queue population with 8 threads) per cycle to issue. We expect eight threads to provide more parallelism, so perhaps we are not effectively exposing inter-thread parallelism in the queues.

The rest of this chapter focuses on improving this base architecture. The next section addresses each of the problems identified here with different fetch policies and IQ configurations. Section 5.5 examines ways to prevent issue waste, and Section 5.6 re-examines the improved architecture for new bottlenecks, identifying directions for further improvement.

5.4 The Fetch Unit — In Search of Useful Instructions

In this section we examine ways to improve fetch throughput without increasing the fetch bandwidth. Our SMT architecture shares a single fetch unit among eight threads, making the fetch unit a likely bottleneck in this processor, resulting in high contention for this resource. However, we can exploit the high level of competition for the fetch unit in two ways not possible with single-threaded processors: (1) the fetch unit can fetch from multiple threads at once, increasing our utilization of the fetch bandwidth, and (2) it can be selective about which thread or threads to fetch. Because not all paths provide equally useful instructions in a particular cycle, an SMT processor may benefit by fetching from the thread(s) that will provide the best instructions.

We examine a variety of fetch architectures and fetch policies that exploit those advantages. Specifically, they attempt to improve fetch throughput by addressing three factors: fetch efficiency, by partitioning the fetch unit among threads (Section 5.4.1); fetch effectiveness, by improving the quality of the instructions fetched (Section 5.4.2); and fetch availability, by eliminating conditions that block the fetch unit (Section 5.4.3).

5.4.1 Partitioning the Fetch Unit

Recall that our baseline architecture fetches up to eight instructions from one thread each cycle. The frequency of branches in typical instruction streams and the misalignment of branch destinations make it difficult to fill the entire fetch bandwidth from one thread, even for smaller block sizes [CMMP95, SJH89]. In this processor, we can spread the burden of filling the fetch bandwidth among multiple threads. For example, the probability of finding four instructions from each of two threads should be greater than that of finding eight from one thread.

In this section, we attempt to reduce *fetch block fragmentation* (our term for the various factors that prevent us from fetching the maximum number of instructions) by fetching from multiple threads each cycle, while keeping the maximum fetch bandwidth (but not necessarily the I cache bandwidth) constant. We evaluate several fetching schemes, which are labeled *alg.num1.num2,* where *alg* is the fetch selection method (in this section threads are always selected using a round-robin priority scheme), *num1* is the number of threads that can fetch in 1 cycle, and *num2* is the maximum number of instructions fetched per thread in 1 cycle. The maximum number of total instructions fetched is always limited to eight. For each of the fetch partitioning policies, the cache is always 32 kilobytes organized into 8 banks; a given bank can do just one access per cycle.

RR.1.8 — This is the baseline scheme from Section 5.3. Each cycle one thread fetches as many as eight instructions. The thread is determined by a round-robin priority scheme from among those not currently suffering an I cache miss. In this scheme the I cache is indistinguishable from that on a single-threaded superscalar. Each cache bank has its own address decoder and output drivers; each cycle, only one of the banks drives the cache output bus, which is 8 instructions (32 bytes) wide.

RR.2.4, RR.4.2 — These schemes fetch fewer instructions per thread from more threads (four each from two threads, or two each from four threads). If we try to partition the fetch bandwidth too finely, however, we may suffer *thread shortage,* where fewer threads are available than are required to fill the fetch bandwidth.

For these schemes, multiple cache addresses are driven to each cache data bank, each of which now has a multiplexer before its address decoder, to select one cache index per cycle. Since the cache banks are single-ported, bank-conflict logic is needed to ensure that each address targets a separate bank. RR.2.4 has two cache output buses, each four instructions wide, while RR.4.2 has four output buses, each two instructions wide. For both schemes, the total width of the output buses is 8 instructions (identical to that in RR.1.8), but additional circuitry is needed so that each bank is capable of driving any of the multiple (now smaller) output buses, and is able to select one or none to drive in a given cycle. Also, the cache tag store logic must be replicated or multiple-ported in order to calculate hit/miss for each address looked up per cycle.

Thus, the hardware additions are: the address multiplexor, multiple address buses, selection logic on the output drivers, the bank conflict logic, and multiple hit/miss calculations. The changes required for RR.2.4 would have a negligible impact on area and cache access time. The changes for RR.4.2 are more extensive, and would be more difficult to do without affecting area or access time. These schemes actually reduce the latency in the decode and rename stages, as the maximum length of dependency chains among fetched instructions is reduced by a factor of 2 and 4, respectively.

RR.2.8 — This scheme attacks fetch block fragmentation without suffering from thread shortage by fetching eight instructions more flexibly from two threads. This can be implemented by reading an eight-instruction block for each thread (16 instructions total), then combining them. We take as many instructions as possible from the first thread, then fill in with instructions from the second, up to eight total. Like RR.2.4, two addresses must be routed to each cache bank, then multiplexed before the decoder; bank-conflict logic and two hit/miss calculations per cycle are necessary; and each bank drives one of the two output buses. Now, however, each output bus is eight instructions wide, which doubles the bandwidth out of the cache compared to the previous schemes. This could be done without greatly affecting area or cycle time, as the additional busing could probably be done without expanding the cache layout. In addition, logic to select and combine the instructions is necessary, which might or might not require an additional pipe stage. Our simulations assume it does not.

Figure 5.4 shows that we can get higher maximum throughput by splitting the fetch over multiple threads. For example, the RR.2.4 scheme outperforms RR.1.8 at 8 threads by 9%. However, this comes at the cost of a 12% single-thread penalty; in fact, RR.2.4 does not surpass RR.1.8 until 4 threads. The RR.4.2 scheme needs 6 threads to surpass RR.1.8 and never catches the 2-thread schemes, suffering from thread shortage.

The RR.2.8 scheme provides the best of both worlds: few-threads performance like RR.1.8 and many-threads performance like RR.2.4. However, the higher throughput of this scheme puts more pressure on the instruction queues, causing IQ-full conditions at a rate of 18% (integer) and 8% (fp) with 8 threads.

With the RR.2.8 scheme we have improved the maximum throughput by 10% without compromising single-thread performance. This was achieved by a combination of (1) partitioning the fetch bandwidth over multiple threads, and (2) making that partition flexible. This is the same approach (although in a more limited fashion here) that simultaneous multithreading uses to improve the throughput of the functional units.

5.4.2 *Exploiting Thread Choice in the Fetch Unit*

The efficiency of the entire processor is affected by the quality of instructions fetched. A multithreaded processor has a unique ability to control that factor. In this section, we

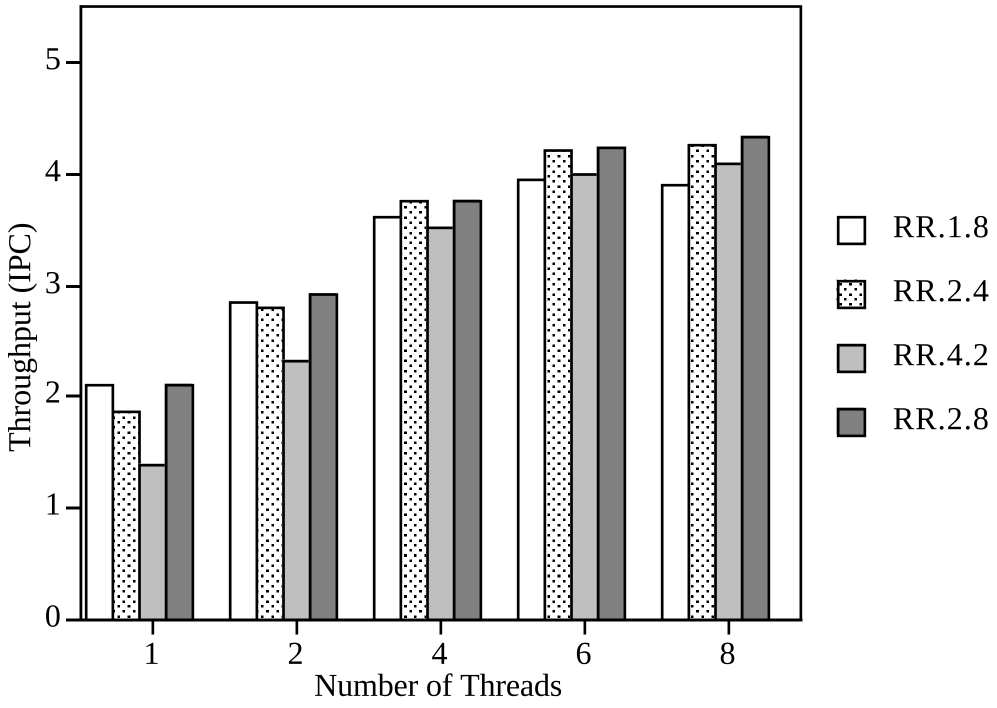


Figure 5.4: **Instruction throughput for the different instruction cache interfaces with round-robin instruction scheduling.**

examine fetching policies aimed at identifying the "best" thread or threads available to fetch each cycle. Two factors make one thread less desirable than another. The first is the probability that a thread is following a wrong path as a result of an earlier branch misprediction. Wrong-path instructions consume not only fetch bandwidth, but also registers, IQ space, and possibly issue bandwidth. The second factor is the length of time the fetched instructions will be in the queue before becoming issuable. We maximize the throughput of a queue of bounded size by feeding it instructions that will spend the least time in the queue. If we fetch too many instructions that block for a long time, we eventually fill the IQ with unissuable instructions, a condition we call *IQ clog*. This restricts both fetch and issue throughput, causing the fetch unit to go idle and preventing issuable instructions from getting into the IQ. Both of these factors (wrong-path probability and expected queue time) improve over time, so a thread becomes more desirable as we delay fetching it.

We define several fetch policies, each of which attempts to improve on the round-robin priority policy using feedback from other parts of the processor. The first attacks wrong-path fetching, the others attack IQ clog. They are:

BRCOUNT — Here we attempt to give highest priority to those threads that are least likely to be on a wrong path. We do this by counting branch instructions that are in the decode stage, the rename stage, and the instruction queues, favoring those with the fewest unresolved branches.

MISSCOUNT — This policy detects an important cause of IQ clog. A long memory latency can cause dependent instructions to back up in the IQ waiting for the load to complete, eventually filling the queue with blocked instructions from one thread. This policy prevents that by giving priority to those threads that have the fewest outstanding D cache misses.

ICOUNT — This is a more general solution to IQ clog. Here priority is given to threads with the *fewest* instructions in decode, rename, and the instruction queues. This achieves three purposes: (1) it prevents any one thread from filling the IQ, (2) it gives highest priority to threads that are moving instructions through the IQ most efficiently, and (3) it provides the most even mix of instructions from the available threads, maximizing the amount of inter-thread parallelism in the queues. If cache misses are the dominant cause of IQ clog, MISSCOUNT may perform better, since it gets cache miss feedback to the fetch unit more quickly. If the causes are more varied, ICOUNT should perform better.

IQPOSN — Like ICOUNT, IQPOSN strives to minimize IQ clog and bias toward efficient threads. It gives lowest priority to those threads with instructions closest to the head of either the integer or floating point instruction queues (the oldest instruction is at the head of the queue). Threads with the oldest instructions will be most prone to IQ clog, and those making the best progress will have instructions farthest from the head of the queue. This policy does not require a counter for each thread, as do the previous three policies. All of these policies require some kind of mechanism to select threads for fetch once the priorities have been computed.

Like any control system, the efficiency of these mechanisms is limited by the feedback latency resulting, in this case, from feeding data from later pipeline stages back to the fetch stage. For example, by the time instructions enter the *queue* stage or the *exec* stage, the information used to fetch them is three or (at least) six cycles old, respectively.

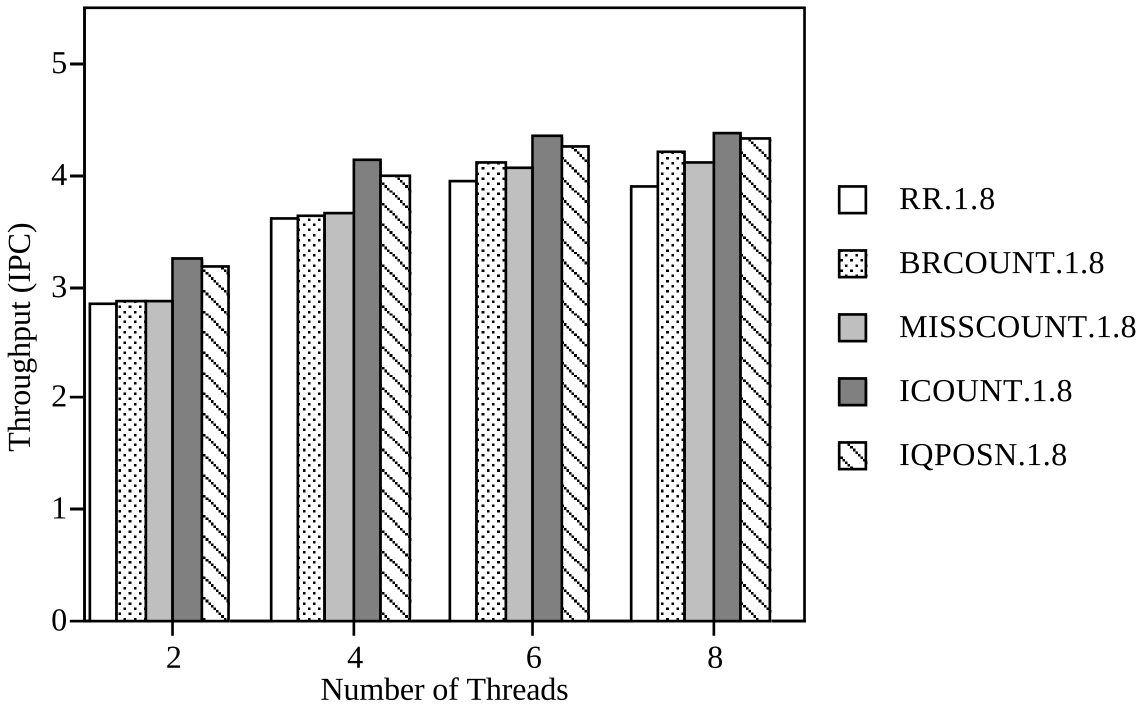
Both the branch-counting and the miss-counting policies tend to produce frequent ties. In those cases, the tie-breaker is round-robin priority.

Figure 5.5 shows that all of the fetch heuristics provide speedup over round-robin. Branch counting and cache-miss counting provide moderate speedups, but only when the processor is saturated with many threads. Instruction counting, in contrast, produces more significant improvements regardless of the number of threads. IQPOSN provides similar results to ICOUNT, being within 4% at all times, but never exceeding it.

The branch-counting heuristic does indeed reduce wrong-path instructions, from 8.2% of fetched instructions to 3.6%, and from 3.6% of issued instructions to 0.8% (RR.1.8 vs. BRCOUNT.1.8 with eight threads). And it improves throughput by as much as 8%. Its weakness is that the wrong-path problem it solves is not large on this processor, which has already attacked the problem with simultaneous multithreading. Even with the RR scheme, simultaneous multithreading reduces fetched wrong-path instructions from 16% with one thread to 8% with 8 threads.

Cache miss counting also achieves throughput gains as high as 8% over RR, but in general the gains are lower. It is not particularly effective at reducing IQ clog, as we get IQ-full conditions 12% of the time on the integer queue and 14% on the floating point queue (for MISSCOUNT.2.8 with 8 threads). These results indicate that IQ clog is more than simply the result of long memory latencies.

The instruction-counting heuristic provides instruction throughput as high as 5.3 instructions per cycle, a throughput gain over the unmodified superscalar of 2.5. It outperforms the best round-robin result by 23%. Instruction counting is as effective at 2 and 4 threads



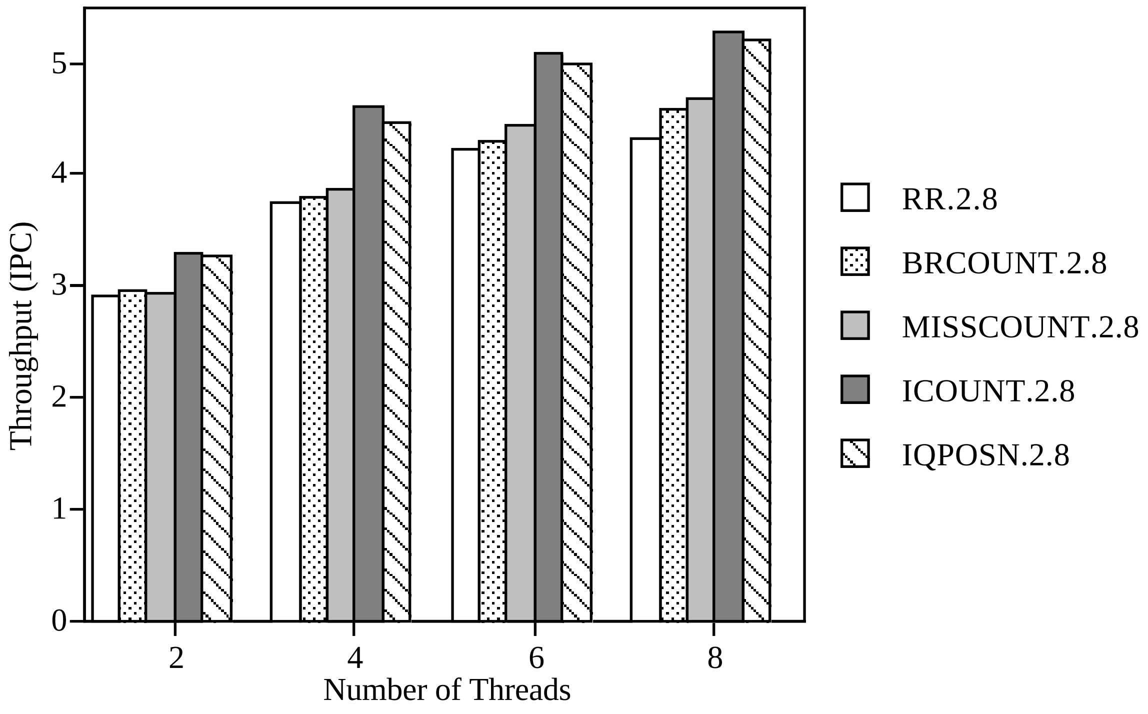


Figure 5.5: **Instruction throughput for fetching based on several priority heuristics, all compared to the baseline round-robin scheme. The results for 1 thread are the same for all schemes, and thus not shown.**

Table 5.3: **Some low-level metrics for the round-robin and instruction-counting priority policies (and the 2.8 fetch partitioning scheme).**

	1	8 Threads	
Metric	**Thread**	**RR**	**ICOUNT**
integer IQ-full (% of cycles)	7%	18%	6%
fp IQ-full (% of cycles)	14%	8%	1%
avg queue population	25	38	30
out-of-registers (% of cycles)	3%	8%	5%

(in benefit over round-robin) as it is at 8 threads. It nearly eliminates IQ clog (see IQ-full results in Table 5.3) and greatly improves the mix of instructions in the queues (we are finding more issuable instructions despite having fewer instructions in the two queues). Intelligent fetching with this heuristic is of greater benefit than partitioning the fetch unit, as the ICOUNT.1.8 scheme consistently outperforms RR.2.8.

Table 5.3 points to a surprising result. As a result of simultaneous multithreaded instruction issue and the ICOUNT fetch heuristics, we actually put *less* pressure on the same instruction queue with eight threads than with one, having sharply reduced IQ-full conditions. It also reduces pressure on the register file (vs. RR) by keeping fewer instructions in the queue.

BRCOUNT and ICOUNT each solve different problems, and perhaps the best performance could be achieved from a weighted combination of them; however, the complexity of the feedback mechanism increases as a result. By itself, instruction counting clearly provides the best gains.

Given our measurement methodology, it is possible that the throughput increases could be overstated if a fetch policy simply favors those threads with the most inherent instruction-level parallelism or the best cache behavior (increasing the sampling of instructions from high-throughput threads in the results), achieving improvements that would not be seen in practice. However, with the ICOUNT policy the opposite happens. Our results show that this scheme favors threads with lower single-thread ILP, thus its results include a higher sample of instructions from the slow threads than either the superscalar results or the RR results. For example, the four slowest threads represent 41% of the sampled instructions with the RR.2.8 scheme, but those same four threads represent 45% of the sampled instructions for the ICOUNT.2.8 results. If anything, then, the ICOUNT.2.8 improvements are understated.

In summary, we have identified a simple heuristic that successfully identifies the best threads to fetch. Instruction counting dynamically biases toward threads that will use processor resources most efficiently, thereby improving processor throughput as well as relieving pressure on scarce processor resources: the instruction queues, the registers, and the fetch bandwidth.

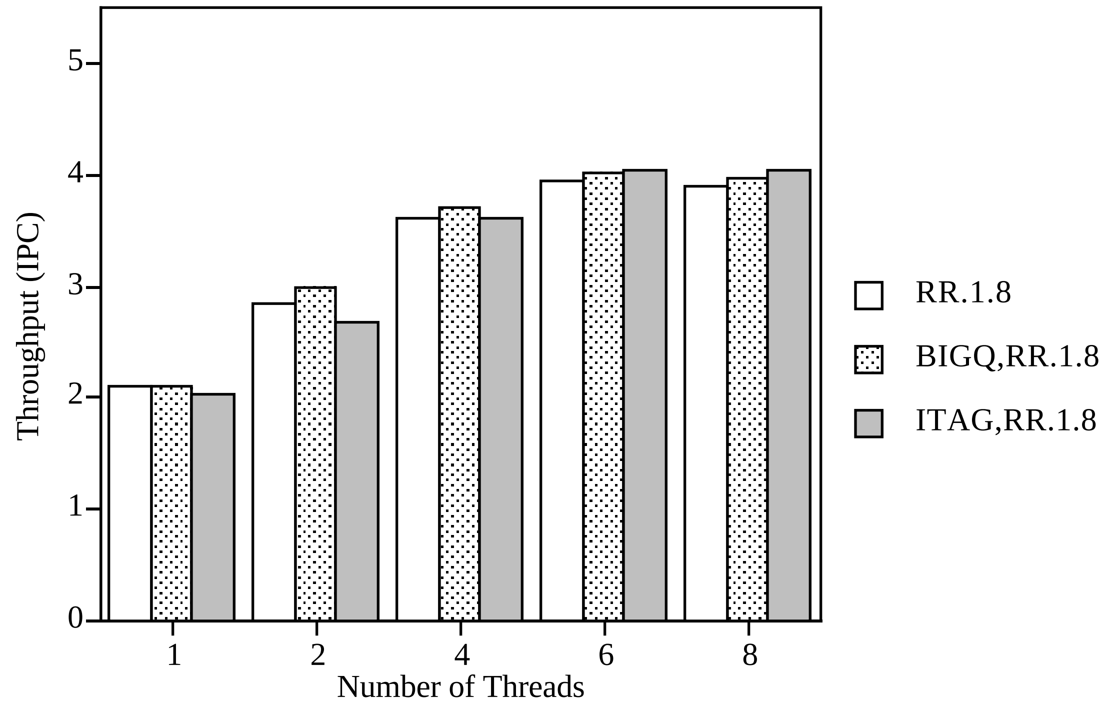
5.4.3 Unblocking the Fetch Unit

By fetching from multiple threads and using intelligent fetch heuristics, we have significantly increased fetch throughput and efficiency. The more efficiently we are using the fetch unit, the more we stand to lose when it becomes blocked. In this section we examine schemes that prevent two conditions that cause the fetch unit to miss fetch opportunities, specifically IQ-full conditions (discussed already in the previous section) and I cache misses. The two schemes are:

BIGQ — The primary restriction on IQ size is not the chip area, but the time to search it; therefore we can increase its size as long as we don't increase the search space. In this scheme, we double the sizes of the instruction queues, but only search the first 32 entries for issue. This scheme allows the queues to buffer instructions from the fetch unit when the IQ overflows, preventing queue-full events.

ITAG — When a thread is selected for fetching but experiences a cache miss, we lose the opportunity to fetch that cycle. If we do the I cache tag lookups a cycle early, we can fetch around cache misses: cache miss accesses are still started immediately, but only non-missing threads are chosen for fetch. Because we need to have the fetch address a cycle early, we essentially add a stage to the front of the pipeline, increasing the misfetch and mispredict penalties. This scheme requires one or more additional ports on the I cache tags, so that potential replacement threads can be looked up at the same time.

Although the BIGQ scheme improves the performance of the round-robin scheme (Figure 5.6) 1.5-2% across the board, Figure 5.7 shows that the bigger queues add no significant improvement to the ICOUNT policy. In fact, it is actually detrimental for several thread configurations. This is because the buffering effect of the big queue scheme brings instructions into the issuable part of the instruction queue that may have been fetched



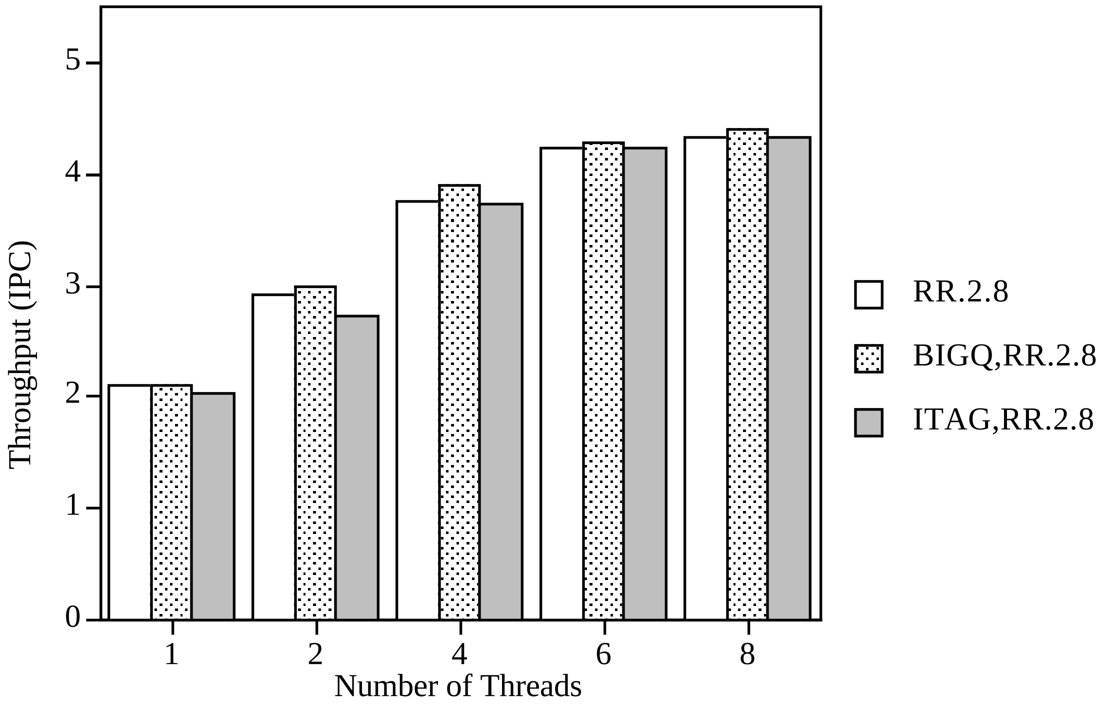
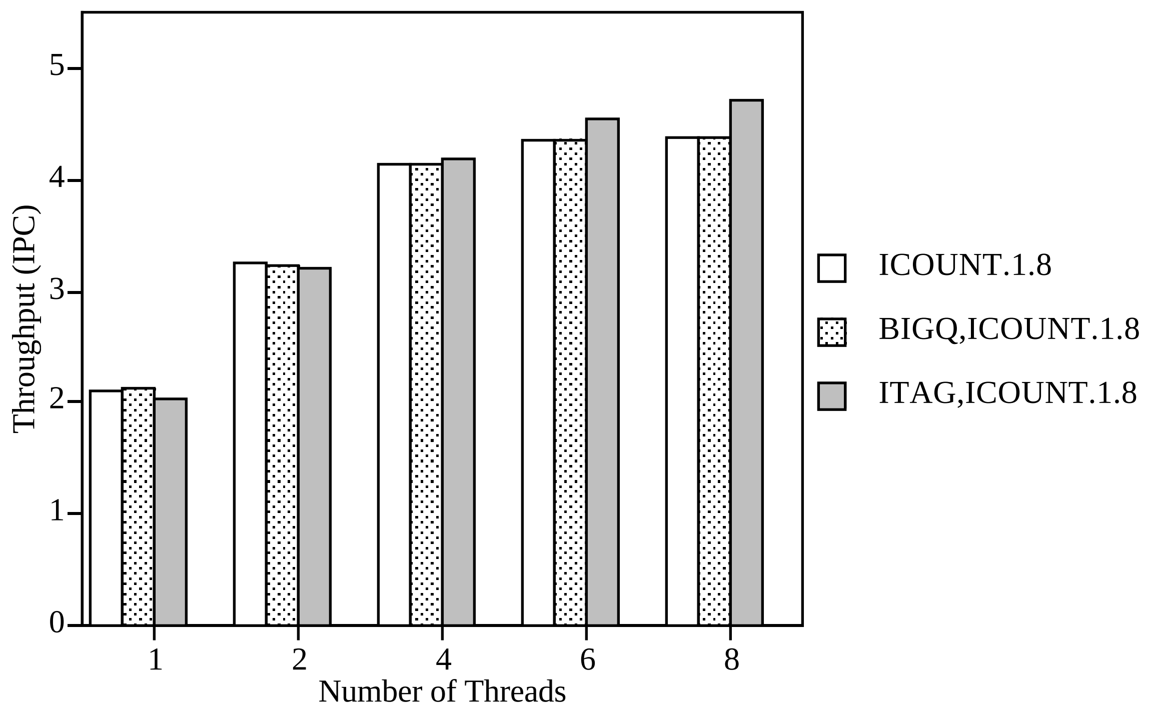


Figure 5.6: **Instruction throughput for the 64-entry queue and early I cache tag lookup, when coupled with the RR fetch policy.**



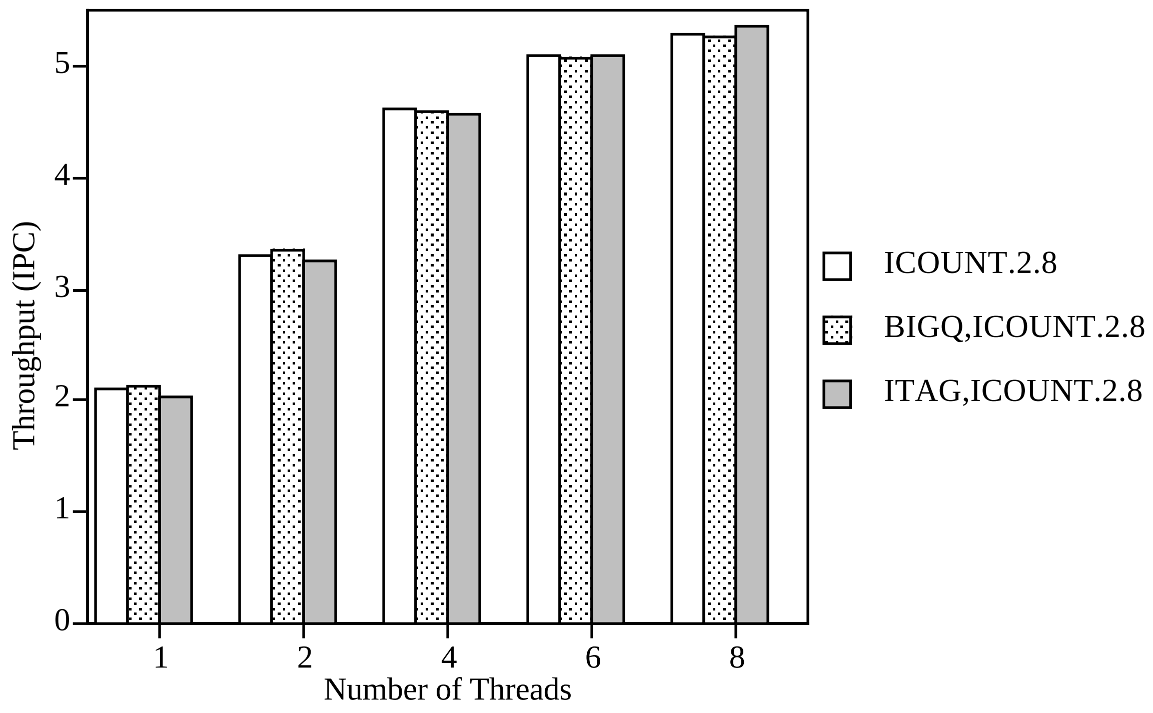


Figure 5.7: **Instruction throughput for the 64-entry queue and early I cache tag lookup, when coupled with the ICOUNT fetch policy.**

many cycles earlier, using priority information that is now out-of-date. The results indicate that using up-to-date priority information is more important than buffering.

These results show that intelligent fetch heuristics have made the extra instruction queue hardware unnecessary. The bigger queue by itself is actually *less* effective at reducing IQ clog than the ICOUNT scheme. With 8 threads, the bigger queues alone (BIGQ,RR.2.8) reduce IQ-full conditions to 11% (integer) and 0% (fp), while instruction counting alone (ICOUNT.2.8) reduces them to 6% and 1%. Combining BIGQ and ICOUNT drops them to 3% and 0%.

Early I cache tag lookup boosts throughput as much as 8% over ICOUNT. It is most effective when fetching one thread (1.8 rather than 2.8 partitioning, where the cost of a lost fetch slot is greater), and when the demand on the fetch unit is highest (ICOUNT, rather than RR). It improves the ICOUNT.2.8 results no more than 2%, as the flexibility of the 2.8 scheme already hides some of the lost fetch bandwidth. In addition, ITAG lowers throughput with few threads, where competition for the fetch slots is low and the cost of the longer misprediction penalty is highest.

Using a combination of partitioning the fetch unit (2.8), intelligent fetching (ICOUNT), and early I cache tag lookups (ITAG), we have raised the peak performance of the base SMT architecture by 37% (5.4 instructions per cycle vs. 3.9). Our maximum speedup relative to a conventional superscalar has gone up proportionately, from 1.8 to 2.5 times the throughput. That gain comes from exploiting characteristics of a simultaneous multithreading processor not available to a single-threaded machine.

High fetch throughput makes issue bandwidth a more critical resource. We focus on this factor in the next section.

5.5 Choosing Instructions For Issue

Much as the fetch unit in a simultaneous multithreading processor can take advantage of the ability to choose which threads to fetch, the issue logic has the ability to choose instructions for issue. While a dynamically scheduled, single-threaded processor also must choose between ready instructions for issue, with an SMT processor the options are more diverse. Also, because we have higher throughput than a single-threaded superscalar processor, the issue bandwidth is potentially a more critical resource; therefore, avoiding issue slot waste may be more beneficial.

In this section, we examine issue priority policies aimed at preventing waste of the issue bandwidth. Issue slot waste comes from two sources, wrong-path instructions (resulting from mispredicted branches) and optimistically issued instructions. Recall (from Section 5.1) that we optimistically issue load-dependent instructions a cycle before we have D cache hit information. In the case of a cache miss or bank conflict, we have to squash the optimistically issued instruction, wasting that issue slot.

In a single-threaded processor, choosing instructions least likely to be on a wrong path is always achieved by selecting the oldest instructions (those deepest into the instruction queue), because newer instructions will always have more unresolved branches in front of them than will older instructions. In a simultaneous multithreading processor, the position of an instruction in the queue is no longer the best indicator of the level of speculation of that instruction, as right-path instructions are intermingled in the queues with wrong-path.

We examine the following policies:

OLDEST_FIRST is our default algorithm up to this point. It always assigns highest priority to the oldest instructions in the machine. In a single-threaded machine, the oldest instruction is the least speculative, and most likely to be along some critical execution path. In a multithreaded machine with shared instruction queues, that is not necessarily the case. The following policies feature oldest-first scheduling except in the noted instances.

OPT_LAST only issues optimistic (load-dependent instructions) after all other instructions have been issued.

SPEC_LAST, similarly, only issues speculative instructions after all others have been issued. In this case a speculative instruction is defined as any instruction behind a branch from the same thread in the instruction queue. This will not completely eliminate wrong-path instructions because instructions can still be issued after a mispredicted branch, but before the misprediction is discovered.

BRANCH_FIRST issues branches as early as possible in order to identify mispredicted branches quickly.

The default fetch algorithm for each of these schemes is ICOUNT.2.8.

The strong message of Table 5.4 is that issue bandwidth is not yet a bottleneck. Even when it does become a critical resource, the amount of improvement we get from not wasting it is likely to be bounded by the percentage of our issue bandwidth given to useless instructions, which currently stands at 7% (4% wrong-path instructions, 3% squashed optimistic instructions). Because we don't often have more issuable instructions than functional units, we aren't able to and don't need to reduce that significantly. The SPEC_LAST scheme

Table 5.4: **Instruction throughput (instructions per cycle) for the issue priority schemes, and the percentage of useless instructions issued when running with 8 threads.**

Issue	**Number of Threads**					**Useless Instructions**	
Method	1	2	4	6	8	wrong-path	optimistic
OLDEST_FIRST	2.10	3.30	4.62	5.09	5.29	4%	3%
OPT_LAST	2.07	3.30	4.59	5.09	5.29	4%	2%
SPEC_LAST	2.10	3.31	4.59	5.09	5.29	4%	3%
BRANCH_FIRST	2.07	3.29	4.58	5.08	5.28	4%	6%

is unable to reduce the number of useless instructions at all, while OPT_LAST brings it down to 6%. BRANCH_FIRST actually increases it to 10%, as branch instructions are often load-dependent; therefore, issuing them as early as possible often means issuing them optimistically. A combined scheme (OPT_LAST and BRANCH_FIRST) might reduce that side effect, but is unlikely to have much effect on throughput.

Since each of the alternate schemes potentially introduces multiple passes to the IQ search, it is convenient that the simplest mechanism still works well.

The fact that the issue bandwidth failed to become a bottleneck, even with the improvements in the fetch throughput, raises the question of what the bottlenecks really are in our improved architecture. The next section looks for answers to that question.

5.6 Where Are the Bottlenecks Now?

We have shown that proposed changes to the instruction queues and the issue logic are unnecessary to achieve the best performance with this architecture, but that significant gains can be produced by moderate changes to the instruction fetch mechanisms. Here we examine that architecture more closely (using ICOUNT.2.8 as our new baseline), identifying likely directions for further improvements.

In this section we present results of experiments intended to identify bottlenecks in the new design. For components that are potential bottlenecks, we quantify the size of the bottleneck by measuring the impact of relieving it. For some of the components that are not bottlenecks, we examine whether it is possible to simplify those components without creating a bottleneck. Because we are identifying bottlenecks rather than proposing architectures, we are no longer bound by implementation practicalities in these experiments, in many cases modeling ideal implementations.

The Issue Bandwidth — The experiments in Section 5.5 indicate that issue bandwidth is not a bottleneck. In fact, we found that even an infinite number of functional units increases throughput by only 0.5% at 8 threads. Issue bandwidth is, indeed, not a bottleneck.

Instruction Queue Size — Results in Section 5.4 would, similarly, seem to imply that the size of the instruction queues was not a bottleneck, particularly with instruction counting; however, the schemes we examined are not the same as larger, searchable queues, which would also increase available parallelism. Nonetheless, the experiment with larger (64-entry) queues increased throughput by less than 1%, despite reducing IQ-full conditions to 0%. The size of the instruction queue is not a bottleneck.

Fetch Bandwidth — Although we have significantly improved fetch throughput, it is still a prime candidate for bottleneck status. Branch frequency and program counter alignment problems still prevent us from fully utilizing the fetch bandwidth. A scheme that allows us to fetch as many as 16 instructions (up to eight each from two threads), increases throughput 8% to 5.7 instructions per cycle. At that point, however, the IQ size and the number of physical registers each become more of a restriction. Increasing the instruction queues to 64 entries and the excess registers to 140 increases performance another 7% to 6.1 IPC. These results indicate that we have not yet completely removed fetch throughput as a performance bottleneck.

Branch Prediction — Simultaneous multithreading has a dual effect on branch prediction, much as it has on caches. While it puts more pressure on the branch prediction hardware (lowering branch prediction accuracy in a multiprogrammed environment, see Table 5.2), it is more tolerant of branch mispredictions. This tolerance arises because SMT is less dependent on techniques that expose single-thread parallelism (e.g., speculative fetching and speculative execution based on branch prediction) due to its ability to exploit inter-thread parallelism. With one thread running, on average 16% of the instructions we fetch and 10% of the instructions we execute are down a wrong path. With eight threads running and the ICOUNT fetch scheme, only 9% of the instructions we fetch and 4% of the instructions we execute are wrong-path.

Perfect branch prediction boosts throughput by 25% at 1 thread, 15% at 4 threads, and

9% at 8 threads. So despite the significantly decreased efficiency of the branch prediction hardware in multithreaded execution, simultaneous multithreading is much less sensitive to the quality of the branch prediction than a single-threaded processor. Still, better branch prediction is beneficial for both architectures. Significant improvements come at a cost, however; a better scheme than our baseline (doubling the size of both the BTB and PHT) yields only a 2% throughput gain at 8 threads.

Speculative Execution — The ability to do speculative execution on this machine is not a bottleneck, but we would like to know whether eliminating it would create one. The cost of speculative execution (in performance) is not particularly high (again, 4% of issued instructions are wrong-path), but the benefits may not be either.

Speculative execution can mean two different things in an SMT processor, (1) the ability to issue wrong-path instructions that can interfere with others, and (2) the ability to allow instructions to issue before preceding branches from the same thread. In order to guarantee that no wrong-path instructions are issued, we need to delay instructions 4 cycles after the preceding branch is issued. This reduces throughput by 7% at 8 threads and 38% at 1 thread. Simply preventing instructions from passing branches only lowers throughput 1.5% (vs. 12% for 1 thread). Simultaneous multithreading (with many threads) benefits much less from speculative execution than a single-threaded processor; it benefits more from the ability to issue wrong-path instructions than from allowing instructions to pass branches.

The common theme for branch prediction hardware and speculative execution is that these are both techniques for exposing single-stream instruction-level parallelism, thus both help single-threaded and SMT superscalars. However, simultaneous multithreading performance is much less dependent on single-stream parallelism, because it can also take advantage inter-thread parallelism.

Memory Throughput — While simultaneous multithreading hides memory latencies effectively, it does not address the problem of memory bandwidth if some level of the memory/cache hierarchy has become saturated. For that reason, our simulator models memory throughput limitations at multiple levels of the cache hierarchy, as well as the buses between them. With our workload, we never saturate any single cache or bus, but in some cases there are significant queueing delays for certain levels of the cache. If we had infinite bandwidth caches (i.e., the same cache latencies, but no cache bank or bus conflicts), the throughput would only increase by 3%. With our workload, memory throughput is not a limiting factor; however, close attention to this issue is warranted for other workloads.

Register File Size — The number of registers required by this machine is a very significant issue. While we have modeled the effects of register renaming, we have not set the number of physical registers low enough that it is a significant bottleneck. In fact, setting the number of excess registers to infinite instead of 100 only improves 8-thread performance by 2%. Lowering it to 90 reduces performance by 1%, and to 80 by 3%, and 70 by 6%, so there is no sharp drop-off point over this range. The ICOUNT fetch scheme is probably a factor in this, as we've shown that it creates more parallelism with fewer instructions in the machine. With four threads, the reductions due to reducing the number of excess registers were nearly identical.

However, this does not completely address the total size of the register file, particularly when comparing different numbers of threads. An alternate approach is to examine the maximum performance achieved with a given set of physical registers. For example, if we identify the largest register file that could support the scheme outlined in Section 5.1, then we can investigate how many threads to support for the best performance. This more closely models the design process, where the size of the register file is likely to be determined by the desired cycle time rather than vice versa. The tradeoff arises because supporting more hardware contexts leaves fewer (excess) registers available for renaming. The number of renaming registers, however, determines the total number of instructions the processor can have in-flight. It is difficult to predict the right register file size that far into the future, but in Figure 5.8 we illustrate this type of analysis by finding the performance achieved with 200 physical registers. That equates to a 1-thread machine with 168 excess registers or a 4-thread machine with 72 excess registers, for example. In this case there is a clear maximum point at 4 threads.

In summary, fetch throughput is still a bottleneck in our proposed architecture. It may no longer be appropriate to keep fetch and issue bandwidth in balance, given the much greater difficulty of filling the fetch bandwidth. Also, register file access time will likely be a limiting factor in the number of threads an architecture can support.

5.7 Summary

This chapter presents a simultaneous multithreading architecture that:

- borrows heavily from conventional superscalar design, thus limiting the required additional hardware support,

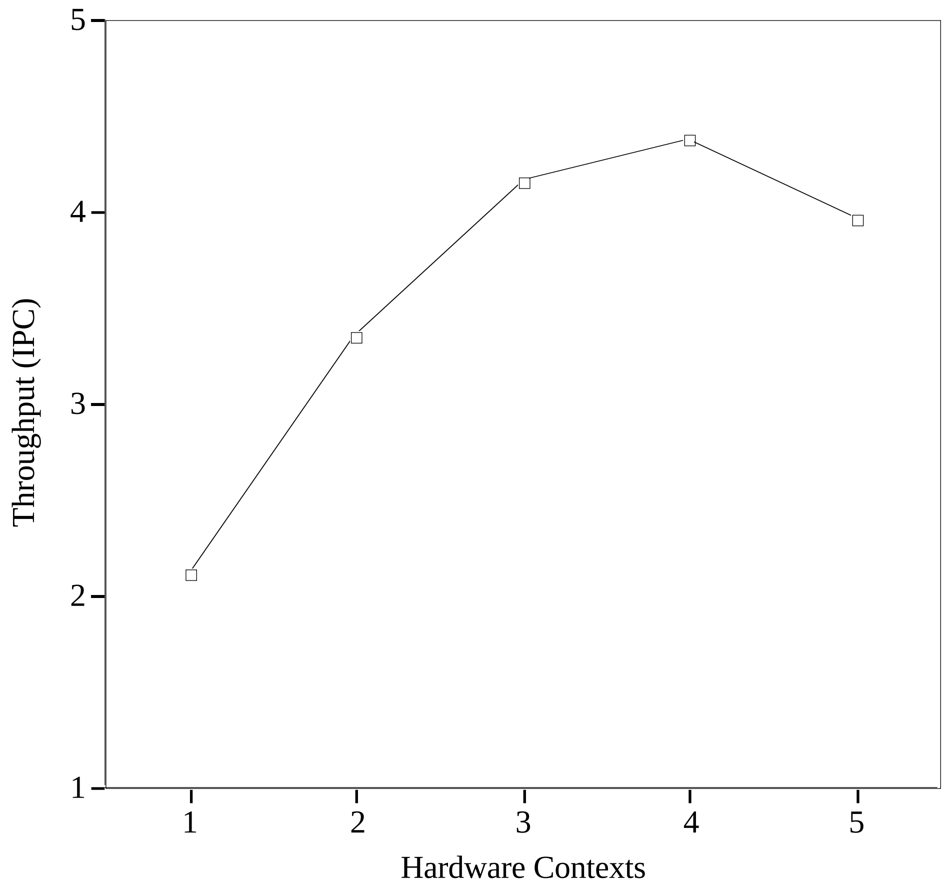


Figure 5.8: **Instruction throughput for machines with 200 physical registers and from 1 to 5 hardware contexts.**

- minimizes the impact on single-thread performance, running only 2% slower in that scenario, and

- achieves significant throughput improvements over the superscalar when many threads are running: a 2.5 throughput gain at 8 threads, achieving 5.4 IPC.

The fetch improvements result from two advantages of simultaneous multithreading unavailable to conventional processors: the ability to partition the fetch bandwidth over multiple threads, and the ability to dynamically select for fetch those threads that will use processor resources most efficiently.

Simultaneous multithreading allows us to achieve multiprocessor-type speedups without multiprocessor-type hardware explosion. This architecture achieves significant throughput gains over a superscalar using the same cache sizes, fetch bandwidth, branch prediction hardware, functional units, instruction queues, and TLBs. The SMT processor is actually less sensitive to instruction queue and branch prediction table sizes than the single-thread superscalar, even with a multiprogrammed workload.

Chapter 6

A Response Time Model of a Simultaneous Multithreading Processor

In Chapter 4 we saw that a single process running alone on a system runs faster than the same process in the presence of other threads on a simultaneous multithreading system. This might lead to the conclusion that simultaneous multithreading increases system throughput at the expense of the performance (or response time) of individual applications. The exact opposite is true, however; in most cases every application will benefit from the improved throughput of the system, resulting in decreased response times. In a typical system a large contributor to response time is queueing time, waiting for other jobs in the processor. Simultaneous multithreading greatly reduces queueing time by (1) increasing the throughput of the processor and (2) allowing much more of the queue (waiting jobs) into the processor at once. In this chapter we show that simultaneous multithreading improves the response time of individual jobs in the system except in the case where the CPU is nearly idle, assuming the same multiprogrammed workload used in Chapter 5. We show this to be true for both interactive and batch workload characteristics that have a single job class (i.e., all jobs have similar priority).

This chapter presents analytic response time models for a simultaneous multithreading processor, using results from the previous chapter on the throughput of an SMT processor. Section 6.1 presents our model of a single CPU handling jobs with a constant arrival rate, and Section 6.2 presents a queueing network model of an interactive system.

6.1 A Simultaneous Multithreading CPU Model

In this section, we model the processor using a birth-death Markov model. This allows us to use the same methods to model both the superscalar processor, which is easily modeled as a simple service queue, and the SMT processor, which is best modeled as a load-dependent service center. A Markov model allows us to solve for system performance metrics derived from a probability distribution (produced by the model) on the queue population. This provides more accurate results for a load-dependent system such as the SMT processor than simpler queueing models that depend only on the average population in the queue.

Our model for the superscalar processor (in this chapter, *superscalar* represents the superscalar processor unmodified for SMT execution) is the M/M/1 queue [Kle75], whose Markov model is shown in Figure 6.1. The arrival rates for the superscalar (λ_i) are all equal to a constant arrival rate, A. Because the throughput of the system is independent of the number of jobs in the system, the completion rates (μ_i) are all equal to a constant completion rate, C. The completion rate is assumed to be that of the unmodified superscalar processor from Chapter 5. A Markov system can be solved for the equilibrium condition by finding population probabilities that equalize the flow in and out of every population state.

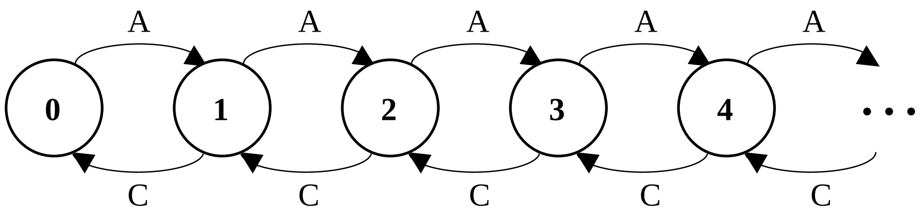


Figure 6.1: **The Markov state diagram for a single processor. Both the arrival rate and the completion rate are constant. The numbers in the circles are the population in the queue, the arcs flowing right are the rate at which jobs enter the system, and the arcs flowing left are the rate at which jobs exit the system.**

If we know A and C, or for most of the quantities just the ratio $\rho = A/C$, we can use the well-known solution for the M/M/1 system [Kle75].

The probability (p_k) that k jobs are in the queue:

$$p_k = p_0(\frac{A}{C})^k = p_0\rho^k$$

allows us to solve for the probability that zero jobs are in the queue:

$$p_0 = 1/[1 + \sum_{k=1}^{\infty} \rho^k] = 1 - \rho \; .$$

Thus the utilization (U), the probability that there is more than one job in the system, is simply:

$$U = 1 - p_0 = \rho = \frac{A}{C} \; .$$

The average population ($\overline{N}$) of the queue is:

$$\overline{N} = \sum_{k=0}^{\infty} k p_k = \frac{\rho}{1 - \rho}$$

and the response time (R), from Little's Law [Lit61]:

$$R = \frac{\overline{N}}{A} = \frac{1/C}{1 - \rho} \; .$$

The more complex Markov model for the SMT processor, shown in Figure 6.2, models a limited load-dependent server. It is similar to the M/M/m queue (a queue with multiple equivalent servers); however, in the M/M/m queue the completion rates vary linearly with the queue population when the population is less than m, which is not true here. In this system, the completion rates are the throughput rates, relative to the superscalar completion rate, C, for the ICOUNT.2.8 architecture from Chapter 5. For example, when there is only one job in the system, the SMT processor runs at .98 times the speed of the superscalar, but when five jobs are in the system, it completes jobs at 2.29 times the speed of the superscalar. That is, while the job at the head of the queue may be running more slowly, because all five jobs are actually executing the completion rate is higher, proportional to the throughput increase over the superscalar.

We assume that the rate of completion is constant once there are more jobs than thread slots in a system. This assumes that the cost of occasionally swapping a thread out of the running set (an operating system context switch) is negligible. This is the same assumption we make to allow us to use a single completion rate for the superscalar model. This assumption favors the superscalar, because it incurs the cost much earlier (when there are two jobs in the running set).

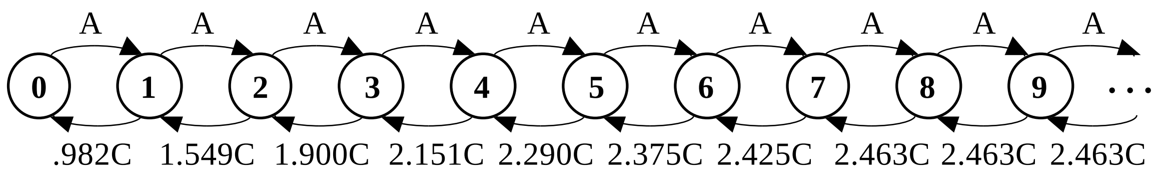


Figure 6.2: **The Markov state diagram for a simultaneous multithreading processor. The arrival rate is constant, but the completion rate varies with the number of jobs in the system.**

Because the arrival rate is still constant, but the completion rates, μ_i are not, the formula for the probability of population k is:

$$p_k = p_0 \prod_{i=1}^{k} \frac{A}{\mu_i}$$

where the μ_i values are as shown in Table 6.1.

Table 6.1: **Completion rates for SMT Markov model.**

i	μ_i
1	.982C
2	1.549C
3	1.900C
4	2.151C
5	2.290C
6	2.375C
7	2.425C
8 - ∞	2.463C

This series becomes manageable because the tail of the Markov state diagram is once again an M/M/1 queue. So

$$p_k = \begin{cases} p_0 \prod_{i=0}^{k} \frac{A}{\mu_i} & k <= 8 \\ p_8(\frac{A}{2.463C})^{k-8} = p_0(\frac{A}{C})^8 \frac{1}{M}(\frac{A}{2.463C})^{k-8} & k > 8 \end{cases}$$

$$\text{where } M = (.982)(1.549)...(2.425)(2.463)$$

As with the M/M/1 queue, because all probabilities sum to one and we can express all probabilities in terms of p_0, we can solve for p_0, and thus the utilization. We can solve it because the sum of all probabilities for $k >= 8$ is a series that reduces to a single term (as long as A/2.463C $<$ 1), as does the M/M/1 queue when A/C $<$ 1. Thus,

$$p_0 = \frac{1}{1 + \frac{A}{C}\frac{1}{.982} + (\frac{A}{C})^2 \frac{1}{.982 * 1.549} + ... + (\frac{A}{C})^8 \frac{1}{M} \frac{1}{1 - \frac{A}{2.463C}}} .$$

In order to calculate response times, we need to know the average population in the queue, so:

$$\overline{N} = \sum_{k=0}^{\infty} kp_k = \sum_{k=0}^{7} kp_k + \sum_{k=0}^{\infty} (k+8)p_8 {\rho'}^k = \sum_{k=0}^{7} kp_k + p_8 \frac{\rho'}{(1-\rho')^2} + 8p_8 \frac{1}{1-\rho'}$$

$$\text{where } \rho' = \frac{A}{2.463C} .$$

Response time can then be computed from the average population and the throughput (A) using Little's Law ($R = \overline{N}/A$).

The utilization and response time results for both the superscalar and the simultaneous multithreading processors are shown in Figure 6.3. Utilization depends only on the ratio $\rho = A/C$, which is the X axis (labeled "Load") in the figure. Response time, however, depends also on the value of A. For this figure, we assume C = 1 (e.g., 1 completion per second for the superscalar), and thus $A = \rho$. In that case, Little's Law reduces to $R = \overline{N}/\rho$. We calculate the utilization of the SMT processor as with the superscalar ($U = 1 - p_0$). That is, the processor is busy if one or more jobs are in the CPU, for the purpose of calculating the utilization for Figures 6.3 and 6.4. Notice that this is a very different definition of

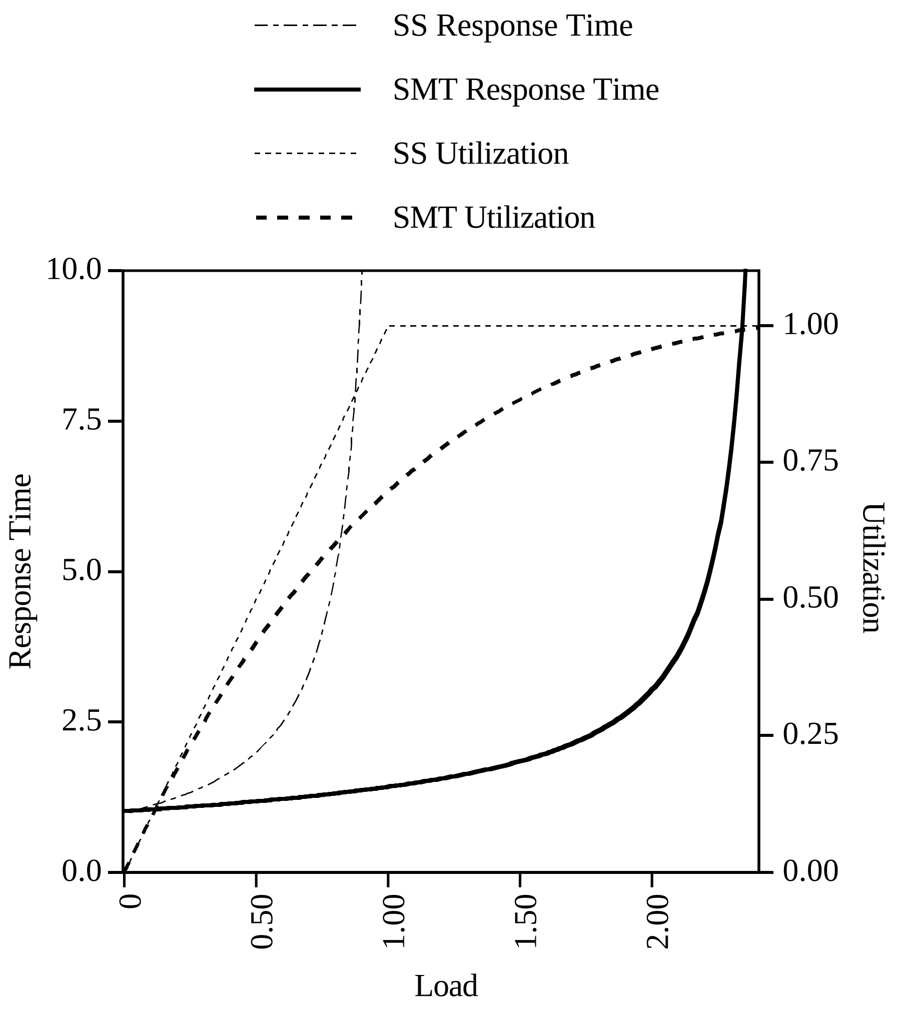


Figure 6.3: **The response times and utilization for both the superscalar CPU and the simultaneous multithreading CPU. Load is the ratio A/C. The response time curves assume a completion rate (C) of one job per second.**

utilization than was used in previous chapters, which were concerned with the utilization of individual resources within the CPU.

The response times of the two processors (the superscalar and the SMT processor) are similar when utilization is very low — when there is little or no queueing in the system. Once the utilization is high enough to cause some queueing, the SMT processor shows significant response time improvements over the superscalar. As the throughput approaches the capacity of the superscalar, that processor's response time approaches infinity, while the SMT response time is still low. The only region where the superscalar has a response time advantage over simultaneous multithreading is when the processor is less than 3% utilized; that is, when the machines are virtually idle.

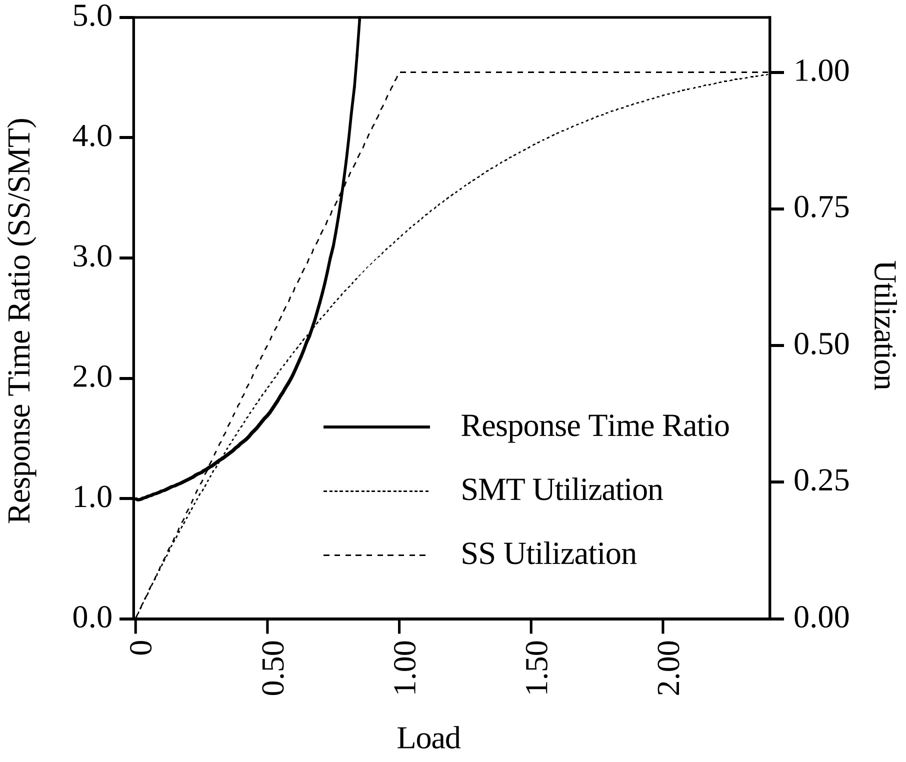


Figure 6.4: **The ratio of the unmodified superscalar's response time to the simultaneous multithreading processor's response time with increasing load on the system.**

The ratio of response times is shown in Figure 6.4. The ratio is infinite for over half of the region for which the SMT processor is short of saturation, due to the superscalar's inability to meet the demand over that range. If we define a normal operating region of a processor as between 40% and 90% utilization, the SMT processor reduces response time by a factor of 1.5 to 7.3 over the normal operating region of the superscalar, and by a factor of 1.6 to infinite over the normal operating region of the simultaneous multithreading processor.

The SMT utilization curve in Figure 6.3 has the unusual characteristic that utilization does not go up linearly with increased demand, because the simultaneous multithreading processor becomes faster with increased load.

In this section, we have modeled response times as observed by individual applications running on an SMT machine and have shown that there is no significant degradation in single application response time with simultaneous multithreading, and that speedups can be quite significant.

The results in this section are for a single queue with a constant arrival stream. This represents a reasonable model for a large server where the arrival rate of jobs is relatively independent of the response time seen by the users. In an interactive system feedback prevents the processor from becoming oversaturated. as a slow response time prevents new jobs from entering the machine. The next section models such a system.

6.2 A Queueing Network Model of a Simultaneous Multithreading System

This section presents a queueing network model for the interactive system shown in Figure 6.5. We will solve for the response time of this system with both a single superscalar CPU, and the simultaneous multithreading CPU from Chapter 5. This queueing network models an interactive system with a number of users submitting jobs with an average CPU service time of 5 seconds (on the unmodified superscalar CPU). The job completion rate at the CPU again varies with the population in the CPU queue for the simultaneous multithreading system. Each job also visits the disk an average of 50 times, each disk access requiring 30 ms. The users' think time (time between receiving the previous response and submitting a new job) is 30 seconds. This models an interactive time-sharing system that is slightly CPU-bound when the SMT processor is heavily utilized, but will be qualitatively similar to a number of interactive scenarios.

We have kept this model intentionally simple because the size of Markov models

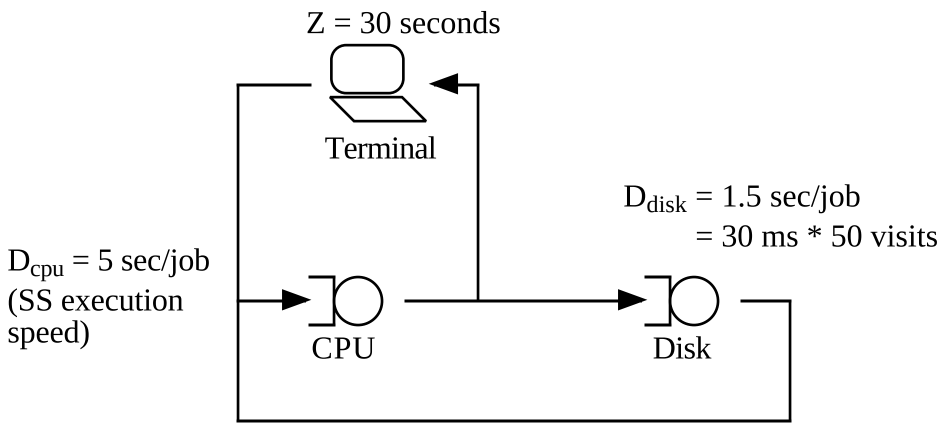


Figure 6.5: **A queueing model for an interactive system with a single CPU and a single disk.**

explode as the number of queues (and the population) increases. For example, the queue shown above with a population of 40 users has 861 states. With another disk, the model would require 12,341 states. The Markov diagram for this queueing network is shown in Figure 6.6, with the parameters shown in Table 6.2.

The states in Figure 6.6 are labeled (C:c,D:d), where c is the number of jobs at the CPU queue, d is the number of jobs at the disk queue, and the rest of the users, $N - c - d$, are thinking.

The results from solving this model are shown in Figure 6.7. The ratio of the response times varies from near unity when the machine has very few users to 2.463 (the ratio of the 8-thread SMT throughput to the superscalar) at the limit when both systems are saturated. That ratio rises quickly, going beyond 2.0 with only nine users. Thus, in a purely interactive system, the response time improvement with simultaneous multithreading is bounded by the improvement in throughput; however, the improvement approaches that limit with just a moderate amount of queueing activity.

The number of users a system can support is typically determined by setting a response time cutoff point. The SMT system, in the limit, supports 2.463 times as many users at the same response time cutoff. Even for much lower response time cutoffs, the SMT system

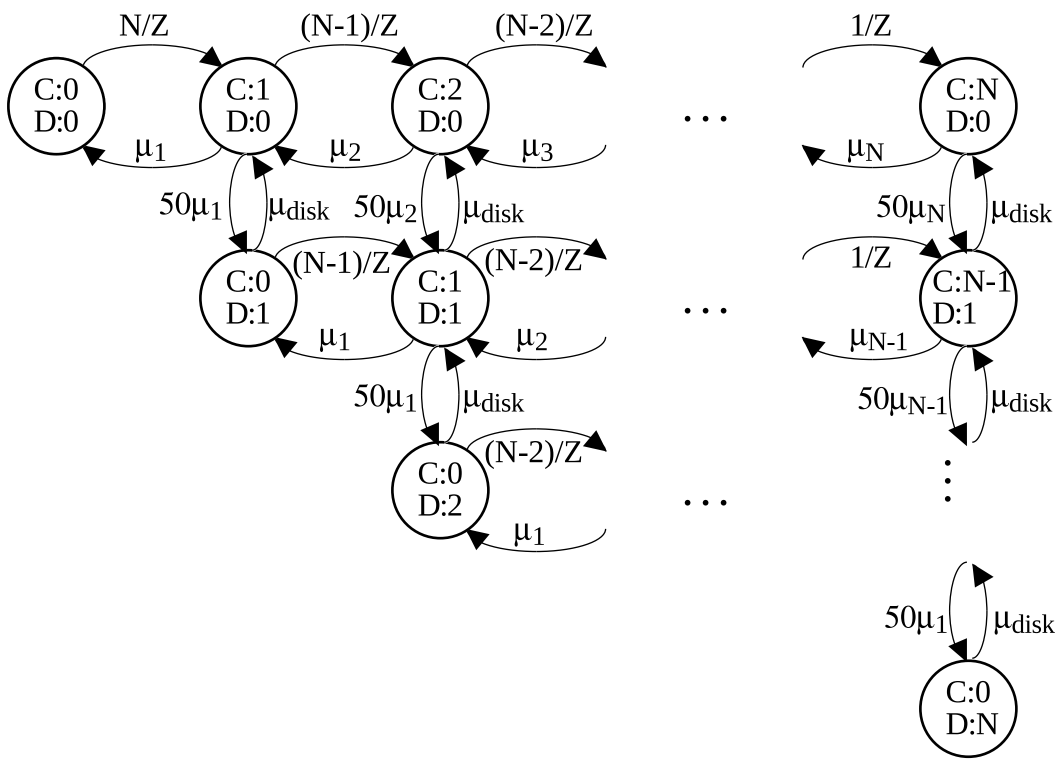


Figure 6.6: **The Markov state diagram for the interactive system**

.

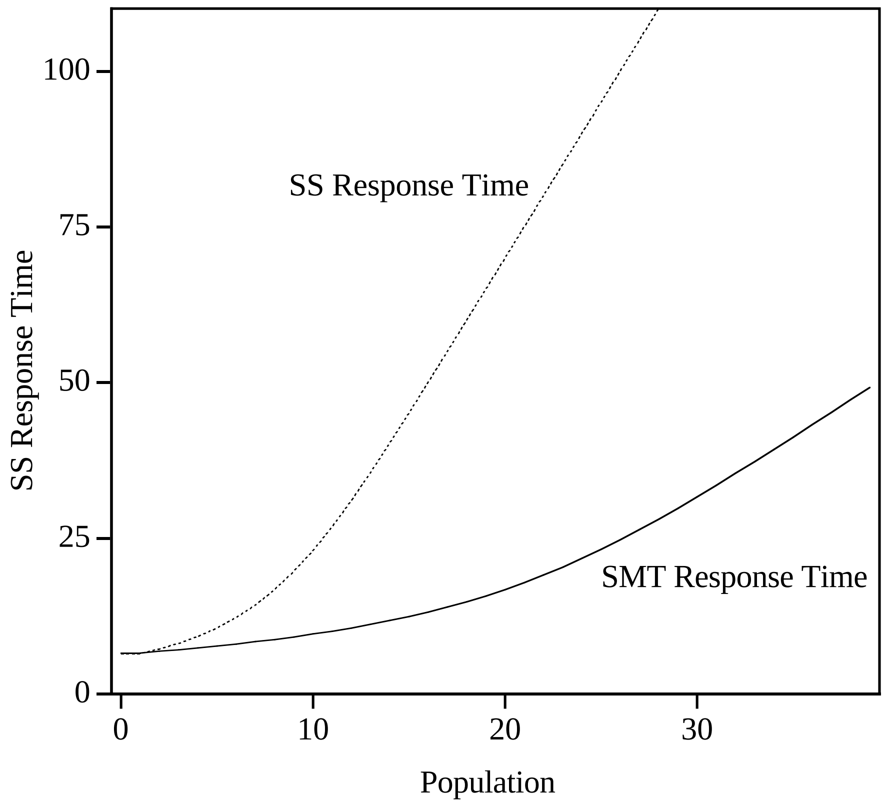


Figure 6.7: **The response times for the queueing network with an unmodified superscalar CPU, and with a simultaneous multithreading CPU.**

Table 6.2: **Model parameters for the interactive system.**

	Superscalar Model	Simultaneous Multithreading Model
μ_1	1/(5 sec)	.982/(5 sec)
μ_2	1/(5 sec)	1.546/(5 sec)
...		
$\mu_i, (i \geq 8)$	1/(5 sec)	2.463/(5 sec)
μ_{disk}	1/(30 ms)	1/(30 ms)
Z	30 sec	30 sec

generally supports about 2.5 times as many users as the superscalar.

6.3 Summary

We have shown that individual process response times improve with a simultaneous multithreading system and a single job class, for both constant arrival rate and interactive workloads, over all interesting configurations (i.e., when the system is not nearly idle). This is the result in the worst case, a workload where no single application takes advantage of simultaneous multithreading's ability to run parallel code. Simultaneous multithreading does not trade off throughput for response time; it provides significant increases in both metrics. It is even possible for the response time improvement to exceed the throughput improvements by a wide margin.

The contribution of this chapter is not the quantitative results, but the qualitative results. The actual response time improvements will vary as the throughput improvements vary by workload. The important result is that the throughput increases provided by simultaneous multithreading translate directly into improved response times for CPU-bound workloads with even a moderate amount of queueing.

Chapter 7

Summary and Conclusions

This dissertation examines *simultaneous multithreading*, a technique that allows multiple independent threads to issue instructions to a superscalar processor's functional units in a single cycle. Simultaneous multithreading combines a superscalar processor's ability to exploit high degrees of instruction-level parallelism with a multithreaded processor's ability to expose more instruction-level parallelism to the hardware by via inter-thread parallelism. Existing multithreading architectures provide the ability to hide processor latencies; simultaneous multithreading adds to that the ability to tolerate low levels of single-stream instruction parallelism.

High performance superscalar processors, both current and next-generation, will see low levels of processor utilization for many applications and workloads. We have shown that an 8-issue processor will see very low utilization (less than 20%) even running CPU-intensive applications with high cache hit rates and branch prediction accuracy. There is little benefit, then, to producing ever wider superscalar processors without also introducing techniques that significantly increase the amount of parallelism exposed to the processor.

We have shown that simultaneous multithreading can significantly increase the instruction throughput of a superscalar processor, exploiting execution resources that would otherwise be idle. A simultaneous multithreading processor has the potential to provide four times the throughput of a single-threaded, 8-issue superscalar processor, and twice the throughput of a fine-grain multithreaded superscalar. We presented several models of simultaneous multithreading and showed that even limited implementations of simultaneous multithreading easily outdistanced these alternative architectures.

Relative to a single-chip multiprocessor, we have shown that a simultaneous multithreading processor has the potential for significantly higher instruction throughput with similar execution resources (instruction issue bandwidth, functional units), or to achieve similar throughput with far fewer execution resources.

The potential performance gains of simultaneous multithreading, however, are only interesting if the complexity of SMT is manageable and doesn't eclipse the performance gains. In this dissertation we have presented an architecture for simultaneous multithreading that minimizes the complexity costs of simultaneous multithreading, and accounts for those costs that affect performance in the simulations.

This architecture demonstrates that high single-thread throughput and high multithread throughput are not mutual exclusive goals. The architecture minimizes the impact on single-thread performance when only one thread is running. When there are multiple threads to run, the architecture achieves significant throughput improvements over the superscalar. One of the keys to achieving this performance improvement without excessive complexity is the ability to (1) share the instruction queues without any loss in performance, and (2) share a single fetch unit. We minimize the performance costs of the latter by exploiting two advantages of simultaneous multithreading unavailable to conventional processors: the ability to partition the fetch bandwidth over multiple threads, and the ability to dynamically select for fetch those threads that are using processor resources most efficiently.

A simultaneous multithreading processor can exploit parallelism at all levels. It can exploit instruction-level parallelism in the same manner as today's aggressive superscalars. It can also exploit process-level parallelism available in a multiprogrammed workload, explicit thread-based parallelism in a parallel workload, and implicit thread-based parallelism that can be extracted from a single program via the compiler. What makes simultaneous multithreading unique is its ability to exploit all of these types of parallelism at once, and in different combinations each cycle.

Simultaneous multithreading represents an attractive option for mainstream processors because it provides a smooth migration path from today's computer usage patterns. Because the impact on single-thread execution is low, the negative impact on performance in the worst case will be nearly unnoticeable, and a user need only have multiple threads or applications running a small fraction of the time to overcome that cost. In fact, simply threading the operating system may be enough to produce noticeable improvements in that scenario.

We have shown that significant performance improvement is gained without any recompilation or reprogramming of applications. That is, many existing workloads will benefit immediately from the introduction of this technology. In fact, simultaneous multithreading will in general make the performance of a workload much more resistant to changes in processor technology, as it is no longer necessary that individual programs be compiled to take full advantage of the underlying hardware to get good processor utilization. Operating systems will need to be modified to take advantage of simultaneous multithreading, but the necessary changes are modest, and operating systems typically react to new processor introductions much faster than applications.

Once simultaneous multithreading is available in the hardware, then compiler, operating system, and runtime support for software multithreading will follow, allowing individual applications to take even greater advantage of the technology. When multithreaded applications are more pervasive, it will facilitate migration to other possible technologies, such as single-chip multiprocessors. This is important for two reasons. First, single-chip multiprocessors alone do not provide a smooth migration from current technology, due to their inability to run a single-thread application as fast as a superscalar with all of its resources devoted to a single execution stream. Second, increasingly wide superscalars will likely not be viable into the distant future. The complexity of certain components of superscalar processors goes up more than linearly with the issue width; for example, forwarding logic, instruction queue to functional unit interconnects, and dependence-checking in the renaming stage. Eventually we will get to the point where multiprocessing is the only way to manage the complexity. If we are already multithreading, the migration from multithreading to multiprocessing will be easy. And the individual processors of the multiprocessor will likely still feature simultaneous multithreading to get maximum utilization out of their resources.

Bibliography

[ACC$^+$90] 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.

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

[ALKK90] A. Agarwal, B.H. 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.

[Bed90] R. Bedichek. Some efficient architecture simulation techniques. In *Winter 1990 Usenix Conference*, pages 53–63, January 1990.

[BP92] C.J. Beckmann and C.D. Polychronopoulos. Microarchitecture support for dynamic scheduling of acyclic task graphs. In *25th Annual International Symposium on Microarchitecture*, pages 140–148, December 1992.

[BYP$^+$91] M. Butler, T.Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction steam parallelism is greater than two. In *18th Annual International Symposium on Computer Architecture*, pages 276–286, May 1991.

[CG94] 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.

[CMMP95] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *22nd Annual International Symposium on Computer Architecture*, pages 333–344, June 1995.

[Den94] M. Denman. PowerPC 604. In *Hot Chips VI*, pages 193–200, August 1994.

[DGH91] H. Davis, S.R. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using Tango. In *International Conference on Parallel Processing*, pages II:99–107, August 1991.

[Dix92] K.M. Dixit. New CPU benchmark suites from SPEC. In *COMPCON, Spring 1992*, pages 305–310, 1992.

[DKF94] P.K. Dubey, A. Krishna, and M. Flynn. Analytic modeling of multithreaded pipeline performance. In *Twenty-Seventh Hawaii International Conference on System Sciences*, pages I:361–367, January 1994.

[DKS96] P.K. Dubey, A. Krishna, and M.S. Squillante. Performance modeling of a multithreaded processor spectrum. In K. Bagchi, J. Walrand, and G. Zobrist, editors, *The State-of-the-Art in Performance Modeling and Simulation of Advanced Computer Systems*, chapter 1, pages 1–25. Gordon and Breach, 1996.

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

[EJK$^+$96] R.J. Eickmeyer, R.E. Johnson, S.R. Kunkel, S. Liu, and M.S. Squillante. Evaluation of multithreaded uniprocessors for commercial application environments. In *23nd Annual International Symposium on Computer Architecture*, pages 203–212, May 1996.

[ER94] J. Edmondson and P. Rubinfield. An overview of the 21164 AXP microprocessor. In *Hot Chips VI*, pages 1–8, August 1994.

[FKD$^+$95] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee. The M-Machine multicomputer. In *28th Annual International Symposium on Microarchitecture,* November 1995.

[FP91] M.K. Farrens and A.R. Pleszkun. Strategies for achieving improved processor throughput. In *18th Annual International Symposium on Computer Architecture*, pages 362–369, 1991.

[Fra93] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin, Madison, 1993.

[FS92] 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.

[GB96] 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.

[GHG$^+$91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[GNL95] R. Govindarajan, S.S. Nemawarkar, and P. LeNir. Design and peformance evaluation of a multithreaded architecture. In *First IEEE Symposium on High-Performance Computer Architecture*, pages 298–307, January 1995.

[Gun93] B.K. Gunther. *Superscalar performance in a multithreaded microprocessor*. PhD thesis, University of Tasmania, December 1993.

[HF88] R.H. 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.

[HKN+92] 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.

[HS96] S. Hiley and A. Seznec. Branch prediction and simultaneous multithreading. In *Conference on Parallel Architectures and Compilation Techniques*, page to appear, October 1996.

[KD79] W.J. Kaminsky and E.S. Davidson. Developing a multiple-instruction-stream single-chip processor. *Computer*, 12(12):66–76, December 1979.

[KD92] 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.

[Kle75] L. Kleinrock. *Queueing Systems Volume 1: Theory*. John Wiley & Sons, 1975.

[LB96] 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.

[LC95] 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.

[LEL+96] J.L. Lo, J.S. Emer, H.M. Levy, R.L. Stamm, and D.M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *In submission*, 1996.

[LFK+93] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. ODonnell, and J.C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[LGH94] 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.

[Lit61] J.D.C. Little. A simple proof of the queueing formula L = λ W. *Operations Research*, 9:383–387, 1961.

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

[McC93] D.C. McCrackin. The synergistic effect of thread scheduling and caching in multithreaded computers. In *COMPCON, Spring 1993*, pages 157–164, 1993.

[McF93] S. McFarling. Combining branch predictors. Technical Report TN-36, DEC-WRL, June 1993.

[NA89] R.S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *16th Annual International Symposium on Computer Architecture*, pages 262–272, June 1989.

[NGGA93] S.S. Nemawarkar, R. Govindarajan, G.R. Gao, and V.K. Ararwal. Analysis of multithreaded architectures with distributed shared memory. In *Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 114–121, December 1993.

[PW91] R.G. 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.

[Rep94a] Microprocessor Report, October 24 1994.

[Rep94b] Microprocessor Report, October 3 1994.

[Rep94c] Microprocessor Report, November 14 1994.

[Rep95] Microprocessor Report, February 16 1995.

[SBCvE90] 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.

[SBV95] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[SF91] G.S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[Sir93] E.G. Sirer. Measuring limits of fine-grained parallelism. Senior Independent Work, Princeton University, June 1993.

[SJH89] M.D. Smith, M. Johnson, and M.A. Horowitz. Limits on multiple instruction issue. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, 1989.

[Smi81a] B.J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Real Time Signal Processing IV*, pages 241–248, 1981.

[Smi81b] J. Smith. A study of branch prediction strategies. In *8th Annual International Symposium on Computer Architecture*, pages 135–148, May 1981.

[TE94] 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.

[TEE$^+$96] 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 *23nd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[TEL95] 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.

[TGH92] K.B. Theobald, G.R. Gao, and L.J. Hendren. On the limits of program parallelism and its smoothability. In *25th Annual International Symposium on Microarchitecture*, pages 10–19, December 1992.

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

[Wal91] D.W. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.

[WG89] W.D. 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.

[YN95] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

[YP92] T.-Y. Yeh and Y. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*, pages 124–134, May 1992.

[YST$^+$94] W. Yamamoto, M.J. Serrano, A.R. Talcott, R.C. Wood, and M. Nemirosky. Performance estimation of multistreamed, superscalar processors. In *Twenty-Seventh Hawaii International Conference on System Sciences*, pages I:195–204, January 1994.

Vita

Dean M. Tullsen was born in Redding, CA, February 21, 1961. He received the B.S. degree (Summa Cum Laude) in Computer Engineering from U.C.L.A. in 1984, and the M.S. degree in Computer Science, also from U.C.L.A. in 1986. He worked in computer architecture for AT&T Bell Laboratories from 1986 to 1990. From 1990 to 1991, he taught in the Computer Science department of Shantou University in Shantou, China. In 1991, he became a graduate student in the Computer Science and Engineering department at the University of Washington, receiving his Ph.D. degree in 1996. In August of 1996, he joined the faculty of the University of California, San Diego.