Simultaneous Multithreading: Maximizing On-Chip Parallelism

Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

This paper examines *simultaneous multithreading*, a technique permitting several independent threads to issue instructions to a superscalar's multiple functional units in a single cycle. We present several models of simultaneous multithreading and compare them with alternative organizations: a wide superscalar, a fine-grain multithreaded processor, and single-chip, multiple-issue multiprocessing architectures. Our results show that both (single-threaded) superscalar and fine-grain multithreaded architectures are limited in their ability to utilize the resources of a wide-issue processor. Simultaneous multithreading has the potential to achieve 4 times the throughput of a superscalar, and double that of fine-grain multithreading. We evaluate several cache configurations made possible by this type of organization and evaluate tradeoffs between them. We also show that simultaneous multithreading is an attractive alternative to single-chip multiprocessors; simultaneous multithreaded processors with a variety of organizations outperform corresponding conventional multiprocessors with similar execution resources.

While simultaneous multithreading has excellent potential to increase processor utilization, it can add substantial complexity to the design. We examine many of these complexities and evaluate alternative organizations in the design space.

1 Introduction

This paper examines *simultaneous multithreading* (SM), a technique that permits several independent threads to issue to multiple functional units each cycle. In the most general case, the binding between thread and functional unit is completely dynamic. The objective of SM is to substantially increase processor utilization in the face of both long memory latencies and limited available parallelism per thread. Simultaneous multithreading combines the multiple-issue-per-instruction features of modem superscalar processors with the latency-hiding ability of multithreaded architectures. It also inherits numerous design challenges from these architectures, e.g., achieving high register file bandwidth, supporting high memory access demands, meeting large forwarding requirements, and scheduling instructions onto functional units. In this paper, we (1) introduce several SM models, most of which limit key aspects of the complexity of such a machine, (2) evaluate the performance of those models relative to superscalar and fine-grain multithreading, (3) show how to tune the cache hierarchy for SM processors, and (4) demonstrate the potential for performance and real-estate advantages of SM 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, modem superscalars, such as the DEC Alpha 21164 [11], PowerPC 604 [9], MIPS R10000 [24], Sun UltraSparc [25], and HP PA-8000 [26] issue up to four instructions per cycle from a single thread. Multiple instruction issue has the potential to increase performance, but is ultimately limited by instruction dependencies (i.e., the available parallelism) and long-latency operations within the single executing thread. The effects of these are shown as *horizontal waste* and *vertical waste* in Figure 1. Multithreaded architectures, on the other hand, such as HEP [28], Tera [3], MASA [15] and Alewife [2] employ multiple threads with fast context switch between threads. Traditional multithreading hides memory and functional unit latencies, attacking vertical waste. In any one cycle, though, these architectures issue instructions from only one thread. The technique is thus limited by the amount of parallelism that can be found in a single thread in a single cycle. And as issue width increases, the ability of traditional multithreading to utilize processor resources will decrease. Simultaneous multithreading, in contrast, attacks both horizontal and vertical waste.

This study 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 modem superscalar processors, we simulate a base architecture derived from the 300 MHz Alpha 21164 [11], enhanced for wider superscalar execution; our SM architectures are extensions of that basic design. Since code scheduling is crucial on wide superscalars, we generate code using the state-of-the-art Multiflow trace scheduling compiler [20].

Our results show the limits of superscalar execution and traditional multithreading to increase instruction throughput in future processors. For example, we show that (1) even an 8-issue superscalar architecture fails to sustain 1.5 instructions per cycle, and (2) 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.

A more traditional means of achieving parallelism is the con-

ISCA '95, Santa Margherita Ligure Italy
© 1995 ACM 0-89791-698-0/95/0006...$3.50

This research was supported by ONR grants N00014-92-J-1395 and N00014-94-1-1136, NSF grants CCR-9200832 and CDA-9123308, NSF PYI Award MIP-9058439, the Washington Technology Center, Digital Equipment Corporation, and a Microsoft Fellowship.

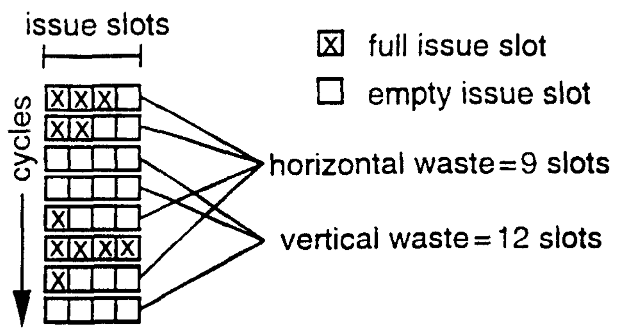


Figure 1: **Empty issue slots can be defined as either vertical waste or horizontal waste. Vertical waste is introduced when the processor issues no instructions in a cycle, horizontal waste when not all issue slots can be filled in a cycle. Superscalar execution (as opposed to single-issue execution) both introduces horizontal waste and increases the amount of vertical waste.**

ventional multiprocessor. As chip densities increase, single-chip multiprocessors will become a viable design option [7]. The simultaneous multithreaded processor and the single-chip multiprocessor are two close organizational alternatives for increasing on-chip execution resources. 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.

For this study we have speculated on the pipeline structure for a simultaneous multithreaded processor, since an implementation does not yet exist. Our architecture may therefore be optimistic in two respects: first, in the number of pipeline stages required for instruction issue; second, in the data cache access time (or load delay cycles) for a shared cache, which affects our comparisons with single-chip multiprocessors. The likely magnitude of these effects is discussed in Sections 2.1 and 6, respectively. Our results thus serve, at the least, as an upper bound to simultaneous multithreading performance, given the other constraints of our architecture. Real implementations may see reduced performance due to various design tradeoffs; we intend to explore these implementation issues in future work.

Previous studies have examined architectures that exhibit simultaneous multithreading through various combinations of VLIW, superscalar, and multithreading features, both analytically [34] and through simulation [16, 17, 6, 23]; we discuss these in detail in Section 7. Our work differs and extends from that work in multiple respects: (1) the methodology, including the accuracy and detail of our simulations, the base architecture we use for comparison, the workload, and the wide-issue compiler optimization and scheduling technology; (2) the variety of SM models we simulate; (3) our analysis of cache interactions with simultaneous multithreading; and finally, (4) in our comparison and evaluation of multiprocessing and simultaneous multithreading.

This paper is organized as follows. Section 2 defines in detail our basic machine model, the workloads that we measure, and the simulation environment that we constructed. Section 3 evaluates the performance of a single-threaded superscalar architecture; it provides motivation for the simultaneous multithreaded approach. Section 4 presents the performance of a range of SM architectures and compares them to the superscalar architecture, as well as a fine-grain multithreaded processor. Section 5 explores the effect of cache design alternatives on the performance of simultaneous multithreading. Section 6 compares the SM approach with conventional multiprocessor architectures. We discuss related work in Section 7, and summarize our results in Section 8.

2 Methodology

Our goal is to evaluate several architectural alternatives as defined in the previous section: 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 execution on our target machine.

2.1 Simulation Environment

Our simulator uses emulation-based instruction-level simulation, similar to Tango [8] and g88 [4]. Like g88, it features caching of partially decoded instructions for fast emulated execution.

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, augmented first for wider superscalar execution and then for multithreaded execution. The model 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.

The typical 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 1 shows the instruction latencies used in the simulations, which are derived from the Alpha 21164.

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

Table 1: **Simulated instruction latencies**

We assume first- and second-level on-chip caches considerably larger than on the Alpha, for two reasons. First, multithreading puts a larger strain on the cache subsystem, and second, we expect larger on-chip caches to be common in the same time-frame that simultaneous multithreading becomes viable. We also ran simulations with caches closer to current processors—we discuss these experiments as appropriate, but do not show any results. The caches (Table 2) are multi-ported by interleaving them into banks, similar to the design of Sohi and Franklin [30]. 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.

	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

Table 2: **Details of the cache hierarchy**

We support limited dynamic execution. Dependence-free instructions are issued in-order to an eight-instruction-per-thread scheduling window; from there, instructions can be scheduled onto functional units out of order, depending on functional unit availability. 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 [20]. This reduces the benefits that might be gained by full dynamic execution, thus eliminating a great deal of complexity (e.g., we don't need register renaming unless we need precise exceptions, and we can use a simple 1-bit-per-register scoreboarding scheme) in the replicated register sets and fetch/decode pipes.

A 2048-entry, direct-mapped, 2-bit branch prediction history table [29] supports branch prediction; the table improves coverage of branch addresses relative to the Alpha (with an 8 KB I cache), which only stores prediction information for branches that remain in the I cache. 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.

For our multithreaded experiments, we assume support is added for up to eight hardware contexts. We support several models of simultaneous multithreaded execution, as discussed in Section 4. 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, and the performance of the fair scheme can be extrapolated from the priority scheme results.

We do not assume any changes to the basic pipeline to accommodate simultaneous multithreading. The Alpha devotes a full pipeline stage to arrange instructions for issue and another to issue. If simultaneous multithreading requires more than two pipeline stages for instruction scheduling, the primary effect would be an increase in the misprediction penalty. We have run experiments that show that a one-cycle increase in the misprediction penalty would have less than a 1% impact on instruction throughput in single-threaded mode. With 8 threads, where throughput is more tolerant of misprediction delays, the impact was less than .5%.

2.2 Workload

Our workload is the SPEC92 benchmark suite [10]. 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 parallel 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.

In the single-thread experiments, 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).

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. By maximizing single-thread parallelism through our compilation system, we avoid overstating the increases in parallelism achieved with simultaneous multithreading.

3 Superscalar Bottlenecks: Where Have All the Cycles Gone?

This section provides motivation for 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.

Using the base single-hardware-context machine, we measured the issue utilization, i.e., the percentage of issue slots that are filled each cycle, for most of the SPEC benchmarks. We also recorded 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,

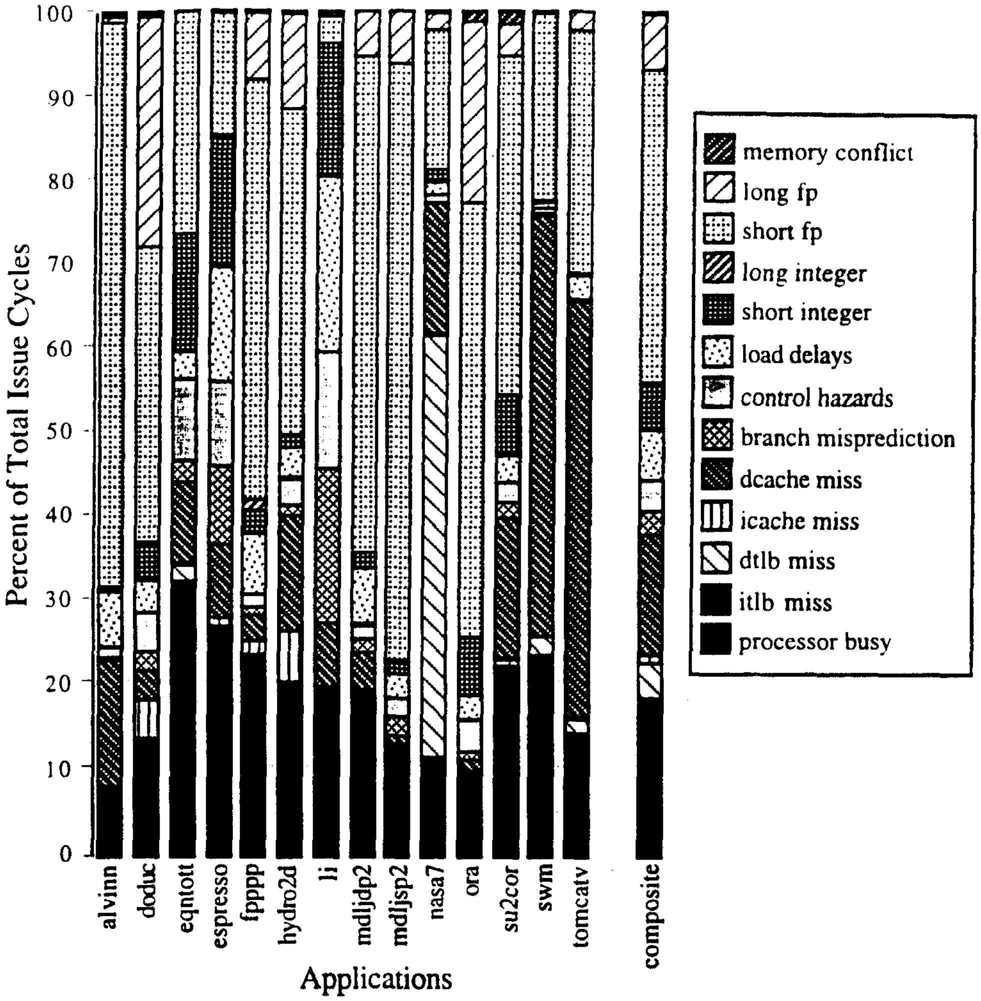


Figure 2: **Sources of all unused issue cycles in an 8-issue superscalar processor.** ***Processor busy*** **represents the utilized issue slots; all others represent wasted issue slots.**

such as an I tlb miss and an I cache miss, the wasted cycles are divided up appropriately. Table 3 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 [32, 5, 18], in contrast, quantified some of these same effects by removing barriers to parallelism and measuring the resulting increases in performance.

Our results, shown in Figure 2, demonstrate that the functional units of our wide superscalar processor are highly underutilized. From the composite results bar on the far right, we see a utilization of only 19% (the "processor busy" component of the composite bar of Figure 2), which represents an average execution of less than 1.5 instructions per cycle on our 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 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 are limited, then any dramatic increase in parallelism needs to come from a general latency-hiding solution, of which multithreading is an example. The different types of multithreading have the potential to hide all sources of latency, but to different degrees.

This becomes clearer if we classify wasted cycles as either vertical

Source of Wasted Issue Slots	Possible Latency-Hiding or Latency-Reducing Technique
instruction tlb miss, data tlb miss	decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses
I cache miss	larger, more associative, or faster instruction cache hierarchy; hardware instruction prefetching
D cache miss	larger, more associative, or faster data cache hierarchy; hardware or software prefetching; improved instruction scheduling; more sophisticated dynamic execution
branch misprediction	improved branch prediction scheme; lower branch misprediction penalty
control hazard	speculative execution; more aggressive if-conversion
load delays (first-level cache hits)	shorter load latency; improved instruction scheduling; dynamic scheduling
short integer delay	improved instruction scheduling
long integer, short fp, long fp delays	(multiply is the only long integer operation, divide is the only long floating point operation) shorter latencies; improved instruction scheduling
memory conflict	(accesses to the same memory location in a single cycle) improved instruction scheduling

Table 3: **All possible causes of wasted issue slots, and latency-hiding or latency-reducing techniques that can reduce the number of cycles wasted by each cause.**

waste (completely idle cycles) or horizontal waste (unused issue slots in a non-idle cycle), as shown previously in Figure 1. 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. The next section provides details on how effectively it does so.

4 Simultaneous Multithreading

This section presents performance results for simultaneous multithreaded processors. We begin by defining several machine models for simultaneous multithreading, spanning a range of hardware complexities. We then show that simultaneous multithreading provides significant performance improvement over both single-thread superscalar and fine-grain multithreaded processors, both in the limit, and also under less ambitious hardware assumptions.

4.1 The Machine Models

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 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 [3], which issues one 3-operation LIW instruction per cycle.

- **SM:Full Simultaneous Issue.** This is a completely flexible simultaneous multithreaded superscalar: all eight threads compete for each of the issue slots each cycle. This is the least realistic model in terms of hardware complexity, but provides insight into the potential for simultaneous multithreading. The following models each represent restrictions to this scheme that decrease hardware complexity.

- **SM:Single Issue, SM:Dual Issue, and SM: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 SM: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.

- **SM:Limited Connection.** 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.

Some important differences in hardware implementation complexity are summarized in Table 4. Notice that the fine-grain model may not necessarily represent the cheapest implementation. Many of these complexity issues are inherited from our wide superscalar design rather than from multithreading, per se. Even in the SM: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

Model	Register Ports	Inter-inst Dependence Checking	Forwarding Logic	Instruction Scheduling onto FUs	Notes
Fine-Grain	H	H	H/L*	L	Scheduling independent of other threads.
SM:Single Issue	L	None	H	H	
SM:Dual Issue	M	L	H	H	
SM:Four Issue	M	M	H	H	
SM:Limited Connection	M	M	M	M	No forwarding between FUs of same type; scheduling is independent of other FUs
SM:Full Simultaneous Issue	H	H	H	H	Most complex, highest performance

* We have modeled this scheme with all forwarding intact, but forwarding could be eliminated, requiring more threads for maximum performance

Table 4: **A comparison of key hardware complexity features of the various models (H=high complexity). We consider the number of ports needed for each register file, the dependence checking for a single thread to issue multiple instructions, the amount of forwarding logic, and the difficulty of scheduling issued instructions onto functional units.**

the number of threads. Our choice of ten functional units seems reasonable for an 8-issue processor. Current 4-issue processors have between 4 and 9 functional units. The number of ports per register file and the logic to select instructions for issue in the four-issue and limited connection models are comparable to current four-issue superscalars; the single-issue and dual-issue are less. The scheduling of instructions onto functional units is more complex on all types of simultaneous multithreaded processors. The Hirata, *et al.*, design [16] is closest to the single-issue model, although they simulate a small number of configurations where the per-thread issue bandwidth is increased. Others [34, 17, 23, 6] implement models that are more similar to full simultaneous issue, but the issue width of the architectures, and thus the complexity of the schemes, vary considerably.

4.2 The Performance of Simultaneous Multithreading

Figure 3 shows the performance of the various models as a function of the number of threads. The segments of each bar indicate the throughput component contributed by each thread. The bar-graphs show three interesting points in the multithreaded design space: fine-grained multithreading (only one thread per cycle, but that thread can use all issue slots), SM: Single Issue (many threads per cycle, but each can use only one issue slot), and SM: Full Simultaneous Issue (many threads per cycle, any thread can potentially use any issue slot).

The fine-grain multithreaded architecture (Figure 3(a)) provides a maximum speedup (increase in instruction throughput) of only 2.1 over single-thread execution (from 1.5 IPC to 3.2). The graph shows that there is little advantage to adding more than four threads in this 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 [2, 1, 19, 14, 33, 31] for both coarse-grain and fine-grain multithreading on *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.

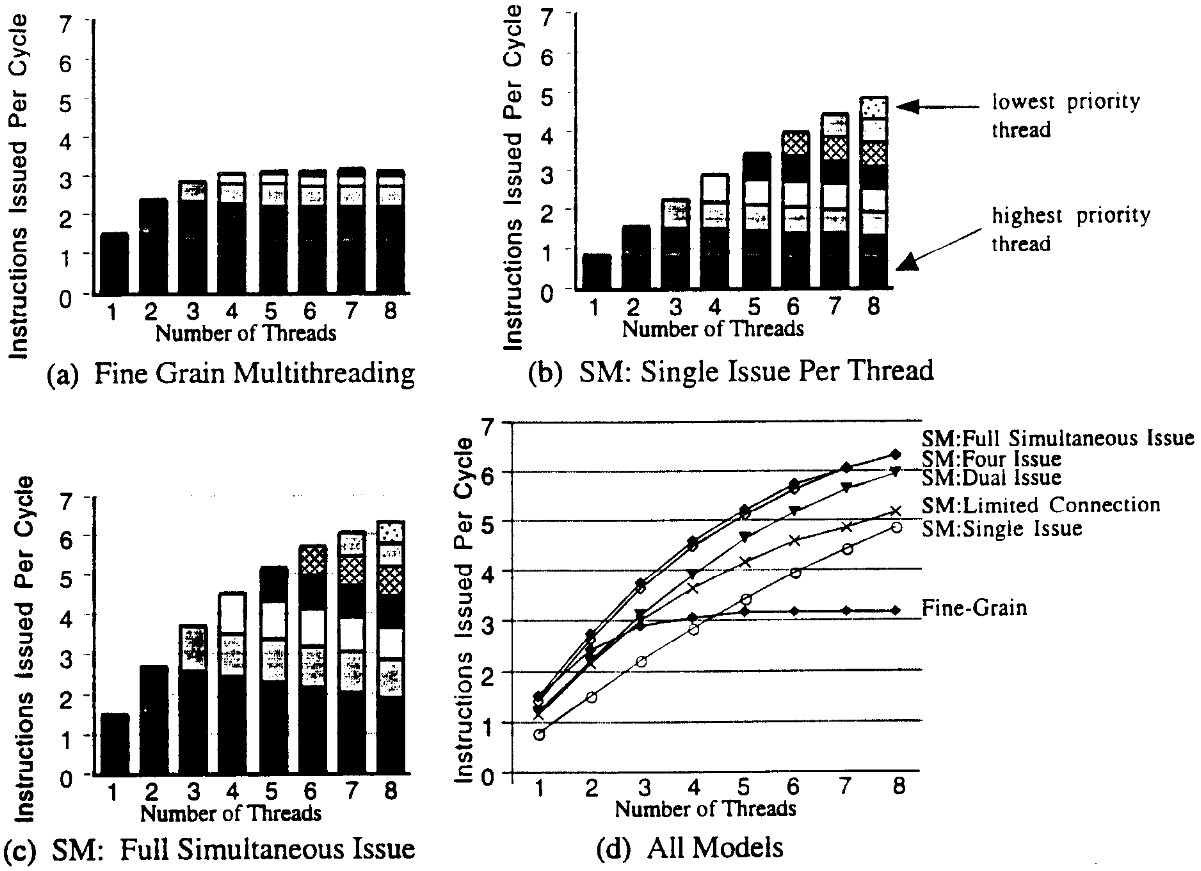
Figures 3(b,c,d) show the advantage of the simultaneous multithreading models, which achieve maximum speedups over single-thread superscalar execution ranging from 3.2 to 4.2, with an issue rate as high as 6.3 IPC. The speedups are calculated using the full simultaneous issue, 1-thread result to represent the single-thread superscalar.

With SM, 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 issue slots increases.

With the results shown in Figure 3(d), 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. Tera [3] 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 that would otherwise remain idle much of the time; however, sharing also has negative effects. We see (in Figure 3(c)) the effect of competition for issue slots and functional units in the full simultaneous issue model, where the lowest priority thread (at 8 threads) runs at 55% of the speed of the highest priority thread. We can 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 [21, 33, 1, 31]. Our data (not shown) pinpoints the ex-



(a) Fine Grain Multithreading

(b) SM: Single Issue Per Thread

(c) SM: Full Simultaneous Issue

(d) All Models

Figure 3: Instruction throughput as a function of the number of threads. (a)-(c) show the throughput by thread priority for particular models, and (d) shows the total throughput for all threads for each of the six machine models. The lowest segment of each bar is the contribution of the highest priority thread to the total throughput.

act areas where sharing degrades performance. 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 less than 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.

It is not necessary to have extremely large caches to achieve the speedups shown in this section. Our experiments with significantly smaller caches (not shown here) reveal that the size of the caches affects 1-thread and 8-thread results equally, making the total speedups relatively constant across a wide range of cache sizes. That is, while 8-thread execution results in lower hit rates than 1-thread execution, the relative effect of changing the cache size is the same for each.

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 SM 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 SM 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.

5 Cache Design for a Simultaneous Multithreaded Processor

Our measurements show a performance degradation due to cache sharing in simultaneous multithreaded processors. In this section, we explore the cache problem further. Our study 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.) All experiments use the 4-issue model with up to 8 threads.

The caches are specified as [total I cache size in KB][*p*rivate or *s*hared].[D cache size][*p*rivate or *s*hared] in Figure 4. For instance.

64p.64s has eight private 8 KB I caches and a shared 64 KB data cache. Not all of the private caches will be utilized when fewer than eight threads are running.

Figure 4 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 64s.64s cache ranks first among all models at 1 thread and last at 8 threads, while the 64p.64p 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 64p.64s with 64p.64p), 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.

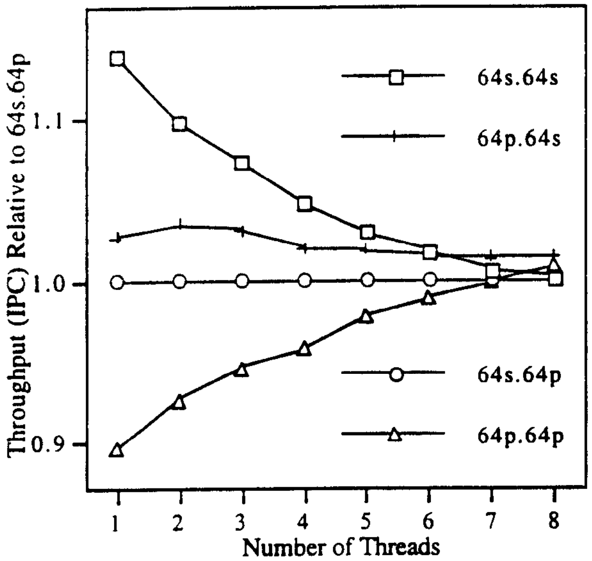


Figure 4: **Results for the simulated cache configurations, shown relative to the throughput (instructions per cycle) of the 64s.64p cache results.**

There are two configurations that appear to be good choices. Because there is little performance difference at 8 threads, the cost of optimizing for a small number of threads is small, making 64s.64s an attractive option. However, if we expect to typically operate with all or most thread slots full, the 64p.64s gives the best performance in that region and is never worse 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 make each thread less sensitive to the presence of other threads. Shared data caches also have a significant advantage in a data-sharing environment by allowing sharing at the lowest level of the data cache hierarchy without any special hardware for cache coherence.

6 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 multiprocessing (MP). On the organizational level, the two approaches are extremely 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 SM processor allows the partitioning to change every cycle. Clearly, scheduling is more complex for an SM processor; however, we will show that in other areas the SM model requires fewer resources, relative to multiprocessing, in order to achieve a desired level of performance.

For these experiments, we tried to choose SM and MP configurations that are reasonably equivalent, although in several cases we biased in favor of the MP. For most of the comparisons we keep all or most of the following equal: the number of register sets (i.e, the number of threads for SM 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 SM, per processor for MP), a 256 KB 4-way set-associative shared second-level cache, and a 2 MB direct-mapped third-level cache. We want to keep the caches constant in our comparisons, and this (private I and D caches) is the most natural configuration for the multiprocessor.

We evaluate MPs with 1, 2, and 4 issues per cycle on each processor. We evaluate SM processors with 4 and 8 issues per cycle; however we use the SM:Four Issue model (defined in Section 4.1) for all of our SM measurements (i.e., each thread is limited to four issues per cycle). Using this model minimizes some of the inherent complexity differences between the SM and MP architectures. For example, an SM: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 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 SM 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 SM results may be optimistic in two respects—the amount of time required to schedule instructions onto functional units, and the shared cache access time. The impact of the former, discussed in Section 2.1, is small. The distance between the load/store units and the data cache can have a large impact on cache access time. The multiprocessor, with private caches and private load/store units, can minimize the distances between them. Our SM processor cannot do so, even with private caches, because the load/store units are shared. However, two alternate configurations could eliminate this difference. Having eight load/store units (one private unit per thread, associated with a private cache) would still allow us to match MP performance with fewer than half the total number of MP functional units (32 vs. 15). Or with 4 load/store

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**: FUs = 32 Issue bw = 8 Reg sets = 8	SM: 8 thread, 8-issue	6.64
		MP: 8 1-issue procs	5.13
	Test B: FUs = 16 Issue bw = 4 Reg sets = 4	SM: 4 thread, 4-issue	3.40
		MP: 4 1-issue procs	2.77
	Test C: FUs = 16 Issue bw = 8 Reg sets = 4	SM: 4 thread, 8-issue	4.15
		MP: 4 2-issue procs	3.44
Unlimited FUs: Test A, but limit SM to 10 FUs	**Test D**: Issue bw = 8 Reg sets = 8	SM: 8 thread, 8 issue, 10 FU	6.36
		MP: 8 1-issue 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	SM: 8 thread, 8-issue	6.64
		MP: 8 4-issue procs	6.35
	Test F: FUs = 16 Reg sets = 4	SM: 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	SM: 8 thread, 8-issue	5.30
		MP: 2 4-issue procs	1.94

Figure 5: **Results for the various multiprocessor vs. simultaneous multithreading comparisons. The multiprocessor always has one functional unit of each type per processor. In most cases the SM processor has the same total number of each FU type as the MP.**

units and 8 threads, we could statically share a single cache/load-store combination among each set of 2 threads. Threads 0 and 1 might share one load/store unit, and all accesses through that load/store unit would go to the same cache, thus allowing us to minimize the distance between cache and load/store unit, while still allowing resource sharing.

Figure 5 shows the results of our SM/MP comparison for various configurations. 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 SM 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.

Test D repeats test A but limits the SM processor to a more reasonable configuration (the same 10 functional unit configuration used throughout this paper). This configuration outperforms the multiprocessor by nearly as much as test A, even though the SM configuration has 22 fewer functional units and requires fewer forwarding connections.

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 SM 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 SM slightly outperforms a 4-processor, 4-issue per processor MP, 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 SM to utilize a fixed number of functional units. Here both SM and MP have 8 functional units and 8 issues per cycle. However, while the SM 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 $2\frac{1}{2}$ times the throughput.

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, SM requires many fewer resources (functional units and instruction issue slots) to achieve a given performance level. For example, a single 8-thread, 8-issue SM 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 SM, an MP system requires eight 4-issue processors (Test E), which consume 32 functional units and 32 issue slots per cycle.

Finally, there are further advantages of SM over MP that are not shown by the experiments:

- Performance with few threads — These results show only the performance at maximum utilization. The advantage of SM (over MP) is greater as some of the contexts (processors) become unutilized. An idle processor leaves $1/p$ of an MP idle, while with SM, 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., IO).

- Granularity and flexibility of design — Our configuration options are much richer with SM, because the units of design have finer granularity. That is, with a multiprocessor, we would typically add computing in units of entire processors. 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 represents the most efficient organization of those resources.

7 Related Work

We have built on work from a large number of sources in this paper. In this section, we note 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 have multiple contexts active simultaneously, but do not have simultaneous multithreading. We also discuss previous studies of architectures that exhibit simultaneous multithreading and contrast our work with these in particular.

The data presented in Section 3 provides a different perspective from previous studies on ILP, which remove barriers to parallelism (i.e. apply real or ideal latency-hiding techniques) and measure the resulting performance. Smith, *et al.*, [28] focus on the effects of fetch, decoding, dependence-checking, and branch prediction limitations on ILP; Butler, *et al.*, [5] examine these limitations plus scheduling window size, scheduling policy, and functional unit configuration; Lam and Wilson [18] focus on the interaction of branches and ILP; and Wall [32] examines scheduling window size, branch prediction, register renaming, and aliasing.

Previous work on coarse-grain [2, 27, 31] and fine-grain [28, 3, 15, 22, 19] multithreading provides the foundation for our work on simultaneous multithreading, but none features simultaneous issuing of instructions from different threads during the same cycle. In fact, most of these architectures are single-issue, rather than superscalar, although Tera has LIW (3-wide) instructions. In Section 4, we extended these results by showing how fine-grain multithreading runs on a multiple-issue processor.

In the M-Machine [7] each processor cluster schedules LIW instructions onto execution units on a cycle-by-cycle basis similar to the Tera scheme. There is no simultaneous issue of instructions from multiple threads to functional units in the same cycle on individual clusters. Franklin's Multiscalar architecture [13, 12] 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.

Hirata, *et al.*, [16] 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. Yamamoto, *et al.*, [34] 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 the same trace). They report increases in instruction throughput of 1.3 to 3 with four threads.

Keckler and Dally [17] and Prasadh and Wu [23] describe architectures that dynamically interleave operations from VLIW instructions onto individual functional units. Keckler and Dally report speedups as high as 3.12 for some highly parallel applications. Prasadh and Wu also examine the register file bandwidth requirements for 4 threads scheduled in this manner. They use infinite caches and show a maximum speedup above 3 over single-thread execution for parallel applications.

Daddis and Torng [6] plot increases in instruction throughput as a function of the fetch bandwidth and the size of the dispatch stack. The dispatch stack is the global instruction window that issues all fetched instructions. Their system has two threads, unlimited functional units, and unlimited issue bandwidth (but limited fetch bandwidth). They report a near doubling of throughput.

In contrast to these studies of multithreaded, superscalar architectures, we use a heterogeneous, multiprogrammed workload based on the SPEC benchmarks; we model all sources of latency (cache, memory, TLB, branching, real instruction latencies) in detail. We also extend the previous work in evaluating a variety of models of SM execution. We look more closely at the reasons for the resulting performance and address the shared cache issue specifically. We go beyond comparisons with single-thread processors and compare simultaneous multithreading with other relevant architectures: fine-grain, superscalar multithreaded architectures and single-chip multiprocessors.

8 Summary

This paper examined *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 multithreading 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 multiprogrammed 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, can achieve 4 times the instruction

throughput of a single-threaded wide superscalar 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 is superior in performance 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. SM 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 increases 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 SM model with complexity in key areas commensurate with that of current superscalars; we also show how properly tuning the cache organization can both increase performance and make individual threads less sensitive to multi-thread contention.

Acknowledgments

We would like to thank John O'Donnell from Equator Technologies, Inc. and Tryggve Fossum of Digital Equipment Corporation for access to the source for the Alpha AXP version of the Multiflow compiler. We would also like to thank Burton Smith, Norm Jouppi, and the reviewers for helpful comments and suggestions on the paper and the research.

References

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

[2] A. Agarwal, B.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.

[3] 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.

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

[5] 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.

[6] 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.

[7] W.J. Dally, S.W. Keckler, N. Carter, A. Chang, M. Fillo, and W.S. Lee. M-Machine architecture v1.0. Technical Report – MIT Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, March 1994.

[8] 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.

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] Microprocessor Report, October 24 1994.

[25] Microprocessor Report, October 3 1994.

[26] Microprocessor Report, November 14 1994.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.