SIMULTANEOUS MULTITHREADING:

A Platform for Next-Generation Processors

Susan J. Eggers
University of Washington

Joel S. Emer
Digital Equipment Corp.

Henry M. Levy
University of Washington

Jack L. Lo
University of Washington

Rebecca L. Stamm
Digital Equipment Corp.

Dean M. Tullsen
University of California, San Diego

Simultaneous multithreading exploits both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle.

As the processor community prepares for a billion transistors on a chip, researchers continue to debate the most effective way to use them. One approach is to add more memory (either cache or primary memory) to the chip, but the performance gain from memory alone is limited. Another approach is to increase the level of systems integration, bringing support functions like graphics accelerators and I/O controllers on chip. Although integration lowers system costs and communication latency, the overall performance gain to applications is again marginal.

We believe the only way to significantly improve performance is to enhance the processor's computational capabilities. In general, this means increasing parallelism—in *all* its available forms. At present only certain forms of parallelism are being exploited. Current superscalars, for example, can execute four or more instructions per cycle; in practice, however, they achieve only one or two, because current applications have low instruction-level parallelism. Placing multiple superscalar processors on a chip is also not an effective solution, because, in addition to the low instruction-level parallelism, performance suffers when there is little thread-level parallelism. A better solution is to design a processor that can exploit all types of parallelism well.

Simultaneous multithreading is a processor design that meets this goal, because it consumes both thread-level and instruction-level parallelism. In SMT processors, thread-level parallelism can come from either multithreaded, parallel programs or individual, independent programs in a multiprogramming workload. Instruction-level parallelism comes from each single program or thread. Because it successfully (and simultaneously) exploits both types of parallelism, SMT processors use resources more efficiently, and both instruction throughput and speedups are greater.

Simultaneous multithreading combines hardware features of wide-issue superscalars and multithreaded processors. From superscalars, it inherits the ability to issue multiple instructions each cycle; and like multithreaded processors it contains hardware state for several programs (or threads). The result is a processor that can issue multiple instructions from multiple threads *each cycle*, achieving better performance for a variety of workloads. For a mix of independent programs (multiprogramming), the overall throughput of the machine is improved. Similarly, programs that are parallelizable, either by a compiler or a programmer, reap the same throughput benefits, resulting in program speedup. Finally, a single-threaded program that must execute alone will have all machine resources available to it and will maintain roughly the same level of performance as when executing on a single-threaded, wide-issue processor.

Equal in importance to its performance benefits is the simplicity of SMT's design. Simultaneous multithreading adds minimal hardware complexity to, and, in fact, is a straightforward extension of, conventional dynamically scheduled superscalars. Hardware designers can focus on building a fast, single-threaded superscalar, and add SMT's multithread capability on top.

Given the enormous transistor budget in the next computer era, we believe simultaneous multithreading provides an efficient base technology that can be used in many ways to extract improved performance. For example, on a one billion transistor chip, 20

0272-1732/97/$10.00 © 1997 IEEE

to 40 SMTs could be used side-by-side to that of achieve performance comparable to a much larger number of conventional superscalars. With IRAM technology, SMT's high execution rate, which currently doubles memory bandwidth requirements, can fully exploit the increased bandwidth capability. In both billion-transistor scenarios, the SMT processor we describe here could serve as the processor building block.

How SMT works

The difference between superscalar, multithreading, and simultaneous multithreading is pictured in Figure 1, which shows sample execution sequences for the three architectures. Each row represents the issue slots for a single execution cycle: a filled box indicates that the processor found an instruction to execute in that issue slot on that cycle; an empty box denotes an unused slot. We characterize the unused slots as horizontal or vertical waste. Horizontal waste occurs when some, but not all, of the issue slots in a cycle can be used. It typically occurs because of poor instruction-level parallelism. Vertical waste occurs when a cycle goes completely unused. This can be caused by a long latency instruction (such as a memory access) that inhibits further instruction issue.

Figure 1a shows a sequence from a conventional superscalar. As in all superscalars, it is executing a single program, or thread, from which it attempts to find multiple instructions to issue each cycle. When it cannot, the issue slots go unused, and it incurs both horizontal and vertical waste.

Figure 1b shows a sequence from a multithreaded architecture, such as the Tera.[1] Multithreaded processors contain hardware state (a program counter and registers) for several threads. On any given cycle a processor executes instructions from one of the threads. On the next cycle, it switches to a different thread context and executes instructions from the new thread. As the figure shows, the primary advantage of multithreaded processors is that they better tolerate long-latency operations, effectively eliminating vertical waste. However, they cannot remove horizontal waste. Consequently, as instruction issue width continues to increase, multithreaded architectures will ultimately suffer the same fate as superscalars: they will be limited by the instruction-level parallelism in a single thread.

Figure 1c shows how *each cycle* an SMT processor selects instructions for execution from all threads. It exploits instruction-level parallelism by selecting instructions from any thread that can (potentially) issue. The processor then dynamically schedules machine resources among the instructions, providing the greatest chance for the highest hardware utilization. If one thread has high instruction-level parallelism, that parallelism can be satisfied; if multiple threads each have low instruction-level parallelism, they can be executed together to compensate. In this way, SMT can recover issue slots lost to both horizontal and vertical waste.

SMT model

We derived our SMT model from a high-performance, out-of-order, superscalar architecture whose dynamic scheduling core is similar to that of the Mips R10000. In each cycle

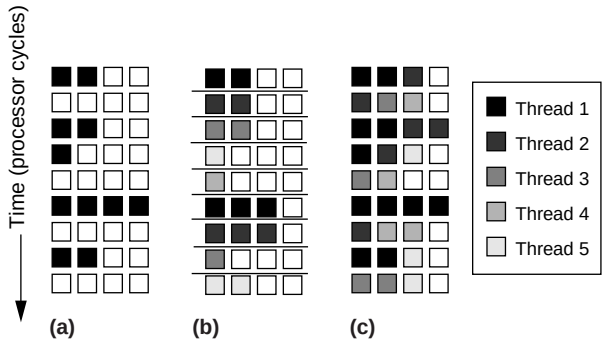


Figure 1. How architectures partition issue slots (functional units): a superscalar (a), a fine-grained multithreaded superscalar (b), and a simultaneous multithreaded processor (c). The rows of squares represent issue slots. The processor either finds an instruction to execute (filled box) or the slot goes unused (empty box).

the processor fetches eight instructions from the instruction cache. After instruction decoding, the register-renaming logic maps the architectural registers to the hardware renaming registers to remove false dependencies. Instructions are then fed to either the integer or floating-point dispatch queues. When their operands become available, instructions are issued from these queues to their corresponding functional units. To support out-of-order execution, the processor tracks instruction and operand dependencies so that it can determine which instructions it can issue and which must wait for previously issued instructions to finish. After instructions complete execution, the processor retires them in order and frees hardware registers that are no longer needed.

Our SMT model, which can simultaneously execute threads from up to eight hardware contexts, is a straightforward extension of this conventional superscalar. We replicated some superscalar resources to support simultaneous multithreading: state for the hardware contexts (registers and program counters) and per-thread mechanisms for pipeline flushing, instruction retirement, trapping, precise interrupts, and subroutine return. We also added per-thread (address-space) identifiers to the branch target buffer and translation look-aside buffer. Only two components, the instruction fetch unit and the processor pipeline, were redesigned to benefit from SMT's multithread instruction issue.

Simultaneous multithreading needs no special hardware to schedule instructions from the different threads onto the functional units. Dynamic scheduling hardware in current out-of-order superscalars is already functionally capable of simultaneous multithreaded scheduling. Register renaming eliminates register name conflicts both within and between threads by mapping thread-specific architectural registers onto the hardware registers; the processor then issues instructions (after their operands have been calculated or loaded from memory) without regard to thread.

This minimal redesign has two important consequences. First, since most hardware resources are still available to a single thread executing alone, SMT provides good perfor-

Because it can fetch instructions from more than one thread, an SMT processor can be selective about which threads it fetches.

mance for programs that cannot be parallelized. Second, because the changes to enable simultaneous multithreading are minimal, the commercial transition from current superscalars to SMT processors should be fairly smooth.

However, should an SMT implementation negatively impact either the targeted processor cycle time or the time to design completion, designers could take several approaches to simplify it. Because most of SMT's implementation complexity stems from its wide-issue superscalar underpinnings, many of the alternatives involve altering the superscalar. One solution is to partition the functional units across a duplicated register file (as in the Alpha 21264) to reduce ports on the register file and dispatch queues.[2] Another is to build interleaved caches augmented with multiple, independently addressed banks, or phase-pipeline cache accesses to increase the number of simultaneous cache accesses. A third alternative would subdivide the dispatch queue to reduce instruction issue delays.[3] As a last resort, a designer could reduce the number of register and cache ports by using a narrower issue width. This alternative would have the greatest impact on performance.

Instruction fetching. In a conventional processor, the instruction unit fetches instructions from a single thread into the execution unit. Performance issues revolve around maximizing the number of useful instructions that can be fetched (by minimizing branch mispredictions, for example) and fetching independent instructions quickly enough to keep functional units busy. An SMT processor places additional stress on the fetch unit. The fetch unit must now fetch instructions more quickly to satisfy SMT's more efficient dynamic scheduler, which issues more instructions each cycle (because it takes them from multiple threads). Thus, the fetch unit becomes SMT's performance bottleneck.

On the other hand, an SMT fetch unit can take advantage of the interthread competition for instruction bandwidth to enhance performance. First, it can partition this bandwidth among the threads. This is an advantage, because branch instructions and cache-line boundaries often make it difficult to fill issue slots if the fetch unit can access only one thread at a time. We fetch from two threads each cycle to increase the probability of fetching only useful (nonspeculative) instructions. In addition, the fetch unit can be smart about which threads it fetches, fetching those that will provide the most immediate performance benefit.

2.8 fetching. The fetch unit we propose is the 2.8 scheme.[4] The unit has eight program counters, one for each thread context. On each cycle, it selects *two* different threads (from those not already incurring instruction cache misses) and fetches *eight* instructions from each thread. To match the issue hardware's lower instruction width, it then chooses a subset of these instructions for decoding. It takes instructions from the first thread until it encounters a branch instruction or the end of a cache line and then takes the remaining instructions from the second thread.

Because the instructions come from two different threads, there is a greater likelihood of fetching useful instructions—indeed, the 2.8 scheme performed 10% better than fetching from one thread at a time. Its hardware cost is only an additional port on the instruction cache and logic to locate the branch instruction—nothing extraordinary for near-future processors. A less hardware intensive alternative is to fetch from multiple threads while limiting the fetch bandwidth to eight instructions. However, the performance gain here is lower: for example, the 2.8 scheme performed 5% better than fetching four instructions from each of two threads.

Icount feedback. Because it can fetch instructions from more than one thread, an SMT processor can be selective about which threads it fetches. Not all threads provide equally useful instructions in a particular cycle. If the processor can predict which threads will produce the fewest delays, performance should improve. Our thread selection hardware uses the Icount feedback technique,[4] which gives highest priority to the threads with the fewest instructions in the decode, renaming, and queue pipeline stages.

Icount increases performance in several ways. First, it replenishes the dispatch queues with instructions from the fast-moving threads, avoiding those that will fill the queues with instructions that depend on and are consequently blocked behind long-latency instructions. Second and most important, it maintains in the queues a fairly even distribution of instructions among these fast-moving threads, thereby increasing interthread parallelism (and the ability to hide more latencies). Finally, it avoids thread starvation, because threads whose instructions are not executing will eventually have few instructions in the pipeline and will be chosen for fetching. Thus, even though eight threads are sharing and competing for slots in the dispatch queues, the percentage of cycles in which the queues are full is actually *less* than on a single-threaded superscalar (8 % of cycles versus 21%).

This performance also comes with a very low hardware cost. Icount requires only a small amount of additional logic to increment (decrement) per-thread counters when instructions enter the decode stage (exit the dispatch queues) and to pick the two smallest counter values.

Icount feedback works because it addresses all causes of dispatch queue inefficiency. In our experiments,[4] it outperformed alternative schemes that addressed a particular cause of dispatch queue inefficiency, such as those that minimize branch mispredictions (by giving priority to threads with the fewest outstanding branches) or minimize load delays (by giving priority to threads with the fewest outstanding on-chip cache misses).

Register file and pipeline. In simultaneous multithreading (as in a superscalar processor), each thread can address 32 architectural integer (and floating-point) registers. The register-renaming mechanism maps these archi-

tectural registers onto a hardware register file whose size is determined by the number of architectural registers in all thread contexts, plus a set of additional renaming registers. The larger SMT register file requires a longer access time; to avoid increasing the processor cycle time, we extended the SMT pipeline two stages to allow two-cycle register reads and two-cycle writes.

The two-stage register access has several ramifications on the architecture. For example, the extra stage between instruction fetch and execute increases the branch misprediction penalty by one cycle. The two extra stages between register renaming and instruction commit increase the minimum time that an executing instruction can hold a hardware register; this, in turn, increases the pressure on the renaming registers. Finally, the extra stage needed to write back results to the register file requires an extra level of bypass logic.

Implementation parameters. SMT's implementation parameters will, of course, change as technologies shrink. In our simulations, they targeted an implementation that should be realized approximately three years from now.

For the CPU, the parameters are

- an eight-instruction fetch/decode width;
- six integer units, four of which can load (store) from (to) memory;
- four floating-point units;
- 32-entry integer and floating-point dispatch queues;
- hardware contexts for eight threads;
- 100 additional integer renaming registers;
- 100 additional floating-point renaming registers; and
- retirement of up to 12 instructions per cycle.

For the memory subsystem, the parameters are

- 128-Kbyte, two-way set associative, L1 instruction and data caches; the D-cache has four dual-ported banks; the I-cache has eight single-ported banks; the access time per bank is two cycles;
- a 16-Mbyte, direct-mapped, unified L2 cache; the single bank has a transfer time of 12 cycles on a 256-bit bus;
- an 80-cycle memory latency on a 128-bit bus;
- 64-byte blocks on all caches;
- 16 outstanding cache misses;
- data and instruction TLBs that contain 128 entries each; and
- McFarling-style branch prediction hardware:[5] a 256-entry, four-way set-associative branch target buffer with an additional thread identifier field, and a hybrid branch predictor that selects between global and local predictors. The global predictor has 13 history bits; the local predictor has a 2,048-entry local history table that indexes into a 4,096-entry prediction table.

Simulation environment

We compared SMT with its two ancestral processor architectures, wide-issue superscalars and fine-grained, multithreaded superscalars. Both are single-processor architectures, designed to improve instruction throughput. Superscalars do so by issuing and executing multiple instructions from a single thread, exploiting instruction-level parallelism. Multithreaded superscalars, in addition to heightening instruction-level parallelism, hide latencies of one thread by switching to and executing instructions from another thread, thereby exploiting thread-level parallelism.

To gauge SMT's potential for executing parallel workloads, we also compared it to a third alternative for improving instruction throughput: small-scale, single-chip shared-memory multiprocessors, whose processors are also superscalars. We examined both two- and four-processor multiprocessors, partitioning their scheduling unit resources (the functional units—and therefore the issue width—dispatch queues, and renaming registers) differently for each case. In the two-processor multiprocessor (MP2), each processor received half of SMT's execution resources, so that the total resources of the two architectures were comparable. Each processor of the four-processor multiprocessor (MP4) contains approximately one-fourth of SMT's chip resources. For some experiments we increased these base configurations until each MP processor had the resource capability of a single SMT.

MP2 and MP4 represent an interesting trade-off between thread-level and instruction-level parallelism. MP2 can exploit more instruction-level parallelism, because each processor has more functional units than its MP4 counterpart. On the other hand, MP4 has two more processors to take advantage of more thread-level parallelism.

All processor simulators are execution-driven, cycle-level simulators; they model the processor pipelines and memory subsystems (including interthread contention for all structures in the memory hierarchy and the buses between them) in great detail. The simulators for the three alternative architectures reflect the SMT implementation model, but without the simultaneous multithreading extensions. Instead they use single-threaded fetching (per processor) and the shorter pipeline, without simultaneous multithreaded issue. The fine-grained multithreaded processor simulator context switches between threads each cycle in a round-robin fashion for instruction fetch, issue, and retirement. Table 1 (next page) summarizes the characteristics of each architecture.

We evaluated simultaneous multithreading on a multiprogramming workload consisting of several single-threaded programs and a group of parallel (multithreaded) applications. We used both types of workloads, because each exercises different parts of an SMT processor.

The larger (workload-wide) working set of the multiprogramming workload should stress the shared structures in an SMT processor (for example, the caches, TLB, and branch prediction hardware) more than the largely identical threads of the parallel programs, which share both instructions and data. We chose the programs in the multiprogramming workload from the Spec95 and Splash2 benchmark suites. Each SMT program executed as a separate thread. To eliminate the effects of benchmark differences when simulating fewer than eight threads, each data point in Table 2 and Figure 2 comprises at least four simulation runs, where each run used a different combination of the benchmarks. The data represent continuous execution with a particular number of threads; that is, we stopped simulating when one of the threads completed.

Table 1. Comparison of processor architectures.*

Features	SS	MP2	MP4	FGMT	SMT
CPUs	1	2	4	1	1
Functional units/CPU	10	5	3	10	10
Architectural registers/CPU (integer or floating point)	32	32	32	256 (8 contexts)	256 (8 contexts)
Dispatch queue size (integer or floating point)	32	16	8	32	32
Renaming registers/CPU (integer or floating point)	100	50	25	100	100
Pipe stages	7	7	7	9	9
Threads fetched/cycle	1	1	1	1	2
Multithread fetch algorithm	n/a	n/a	n/a	Round-robin	Icount

*SS represents a wide-issue superscalar, MP2 and MP4 represent small-scale, single-chip, shared-memory multiprocessors, whose processors (two and four, respectively) are also superscalars, and FGMT represents a fine-grained multithreaded superscalar.

The parallel workload consists of coarse-grained (parallel threads) and medium-grained (parallel loop iterations) parallel programs targeted for shared-memory multiprocessors. As such, it was a fair basis for evaluating MP2 and MP4. It also presents a different, but equally challenging, test of SMT. Unlike the multiprogramming workload, all threads in a parallel application execute the same code, and therefore, have similar execution resource requirements (they may need the same functional units at the same time, for example). Consequently, there is potentially more contention for these resources.

We also selected the parallel applications from the Spec95 and Splash2 suites. Where feasible, we executed the entire parallel portions of the programs; for the long-running Spec95 programs, we simulated several iterations of the main loops, using their reference data sets. The Spec95 programs were parallelized with the SUIF compiler,[6] using policies developed for shared-memory machines.

We compiled the multiprogramming workload with cc and the parallel benchmarks with a version of the Multiflow compiler[7] that produces Alpha executables. For all programs we set compiler optimizations to maximize each program's performance on the superscalar; however, we disabled trace scheduling, so that speculation could be guided by the branch prediction and out-of-order execution hardware.

Simulation results

As Table 2 shows, simultaneous multithreading achieved much higher throughput than the one to two instructions per cycle normally reported for current wide-issue superscalars. Throughput rose consistently with the number of threads; at eight threads, it reached 6.2 for the multiprogramming workload and 6.1 for the parallel applications. As Figure 2b shows, speedups for parallel applications at eight threads averaged 1.9 over the same processor with one thread, demonstrating SMT's parallel processing capability.

As we explained earlier, SMT's pipeline is two cycles longer than that of the superscalar and the processors of the single-chip multiprocessor. Therefore, a single thread executing on an SMT processor has additional latencies and should have lower performance. However, as Figure 2 shows, SMT's single-thread performance was only one percent (parallel workload) and 1.5% (multiprogramming workload) worse than that of the single-threaded superscalar. Accurate branch prediction hardware and the shared pool of renaming registers prevented additional penalties from occurring frequently.

Resource contention on SMT is also a potential problem. Many of SMT's hardware structures, such as the caches, TLBs, and branch prediction tables, are shared by all threads. The unified organization allows a more flexible, and therefore higher, utilization of these structures, as executing threads place different usage demands on them. It also makes the entire structures available when fewer than eight threads—most importantly, a single thread—are executing. On the downside, interthread use leads to competition for the shared resources, potentially driving up cache misses, TLB misses, and branch mispredictions.

We found that interthread interference was significant only for the L1 data cache and the branch prediction tables; the data sets of our workload fit comfortably into the off-chip L2 cache, and conflicts in the TLBs and the L1 instruction cache were minimal. L1 data cache misses rose by 68% (parallel workload) and 66% (multiprogramming workload) and branch mispredictions by 50% and 27%, as the number of threads went from one to eight.

SMT was able to absorb the additional conflicts. Many of the L1 misses were covered by the fully pipelined 16-Mbyte L2 cache, whose latency was only 10 cycles longer than that of the L1 cache. Consequently, L1 interthread conflict misses degraded performance by less than one percent.[8] The most important factor, however, was SMT's ability to simultaneously issue instructions from multiple threads. Thus, although simultaneous multithreading introduces additional conflicts for the shared hardware structures, it has a greater ability to hide them.

SMT vs. the superscalar. The single-threaded superscalar fell far short of SMT's performance. As Table 2 shows, the superscalar's instruction throughput averaged 2.7 instructions per cycle, out of a potential of eight, for the multiprogramming workload; the parallel workload had a slightly higher average throughput of 3.3.

Consequently, SMT executed the multiprogramming workload 2.3 times faster and the parallel workload 1.9 times faster (at eight threads). The superscalar's inability to exploit more instruction-level parallelism and any thread-

level parallelism (and consequently hide horizontal and vertical waste) contributed to its lower performance.

SMT vs. fine-grained multithreading. By eliminating vertical waste, the fine-grained multithreaded architecture provided speedups over the superscalar as high as 1.3 on both workloads. However, this maximum speedup occurred at only four threads; with additional threads, performance fell. Two factors contribute. First, fine-grained multithreading eliminates only vertical waste; given the latency-hiding capability of its out-of-order processor and lockup-free caches, four threads were sufficient to do that. Second, fine-grained multithreading cannot hide the additional conflicts from interthread competition for shared resources, because it can issue instructions from only one thread each cycle. Neither limitation applies to simultaneous multithreading. Consequently, SMT was able to get higher instruction throughput and greater program speedups than the fine-grained multithreaded processor.

Table 2. Instruction throughput executing a multiprogramming workload and a parallel workload.

	Multiprogramming workload			Parallel workload				
Threads	SS	FGMT	SMT	SS	MP2	MP4	FGMT	SMT
1	2.7	2.6	3.1	3.3	2.4	1.5	3.3	3.3
2	—	3.3	3.5	—	4.3	2.6	4.1	4.7
4	—	3.6	5.7	—	—	4.2	4.2	5.6
8	—	2.8	6.2	—	—	—	3.5	6.1

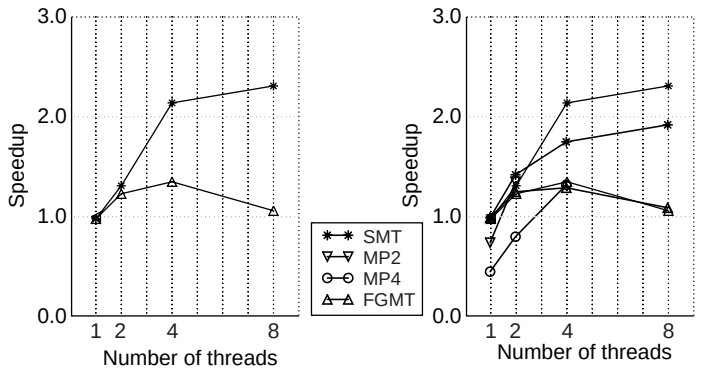


Figure 2. Speedups with the multiprogramming workload (a) and parallel workload (b).

SMT vs. the multiprocessors. SMT obtained better speedups than the multiprocessors (MP2 and MP4), not only when simulating the machines at their maximum thread capability (eight for SMT, four for MP4, and two for MP2), but also for a given number of threads. At maximum thread capability, SMT's throughput reached 6.1 instructions per cycle, compared with 4.3 for MP2 and 4.2 for MP4.

Speedups on the multiprocessors were hindered by the fixed partitioning of their hardware resources across processors, which prevents them from responding well to changes in instruction- and thread-level parallelism. Processors were idle when thread-level parallelism was insufficient; and the multiprocessor's narrower processors had trouble exploiting large amounts of instruction-level parallelism in the unrolled loops of individual threads. An SMT processor, on the other hand, dynamically partitions its resources among threads, and therefore can respond well to variations in both types of parallelism, exploiting them interchangeably. When only one thread is executing, (almost) all machine resources can be dedicated to it; and additional threads (more thread-level parallelism) can compensate for a lack of instruction-level parallelism in any single thread.

To understand how fixed partitioning can hurt multiprocessor performance, we measured the number of cycles in which one processor needed an additional hardware resource and the resource was idle in another processor. (In SMT, the idle resource would have been used.) Fixed partitioning of the integer units, for both arithmetic and memory operations, was responsible for most of MP2's and MP4's inefficient resource use. The floating-point units were also a bottleneck for MP4 on this largely floating-point-intensive workload. Selectively increasing the hardware resources of MP2 and MP4 to match those of SMT eliminated a particular bottleneck. However, it did not improve speedups, because the bottleneck simply shifted to a different resource. Only when we gave *each* processor within MP2 and MP4 *all* the hardware resources of SMT did the multiprocessors obtain greater speedups. However, this occurred only when the architecture executed the same number of threads; at maximum thread capability, SMT still did better.

The speedup results also affect the implementation of these machines. Because of their narrower issue width, the multiprocessors could very well be built with a shorter cycle time. The speedups indicate that a multiprocessor's cycle time must be less than 70% of SMT's before its performance is comparable.

SIMULTANEOUS MULTITHREADING is an evolutionary design that attacks multiple sources of waste in wide-issue processors. Without sacrificing single-thread performance, SMT uses instruction-level and thread-level parallelism to substantially increase effective processor utilization and to accelerate both multiprogramming and parallel workloads. Our measurements show that an SMT

Alternative approaches to exploiting parallelism

Other researchers have studied simultaneous multithreading designs, as well as several architectures that represent alternative approaches to exploiting parallelism. Hirata and colleagues,[1] presented an architecture for an SMT processor and simulated its performance on a ray-tracing application. Gulati and Bagherzadeh[2] proposed an SMT processor with four-way issue. Yamamoto and Nemirovsky[3] evaluated an SMT architecture with separate dispatch queues for up to four threads.

Thread-level parallelism is also essential to other next-generation architectures. Olukotun and colleagues[4] investigated design trade-offs for a single-chip multiprocessor and compared the performance and estimated area of this architecture with that of superscalars. Rather than building wider superscalar processors, they advocate the use of multiple, simpler superscalars on the same chip.

Multithreaded architectures have also been widely investigated. The Tera[5] is a fine-grained multithreaded processor that issues up to three operations each cycle. Keckler and Dally[6] describe an architecture that dynamically interleaves operations from LIW instructions onto individual functional units. Their M-Machine can be viewed as a coarser grained, compiler-driven SMT processor.

Some architectures use threads in a speculative manner to exploit both thread-level and instruction-level parallelism. Multiscalar[7] speculatively executes threads using dynamic branch prediction techniques and squashes threads if control (branches) or data (memory) speculation is incorrect. The superthreaded architecture[8] also executes multiple threads concurrently, but does not speculate on data dependencies.

Although all of these architectures exploit multiple forms of parallelism, only simultaneous multithreading has the ability to dynamically share execution resources among all threads. In contrast, the others partition resources either in space or in time, thereby limiting their flexibility to adapt to available parallelism.

References

1. H. Hirata et al., "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," *Proc. Int'l Symp. Computer Architecture,* Assoc. of Computing Machinery, N.Y., 1992, pp. 136-145.
2. M. Gulati and N. Bagherzadeh, "Performance Study of a Multithreaded Superscalar Microprocessor," *Proc. Int'l Symp. High-Performance Computer Architecture,* ACM, 1996, pp.291-301.
3. W. Yamamoto and M. Nemirovsky, "Increasing Superscalar Performance through Multistreaming," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* IFIP, Laxenburg, Austria, 1995, pp. 49-58.
4. K. Olukotun et al., "The Case for a Single-Chip Multiprocessor," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems, ACM,* 1996, pp. 2-11.
5. R. Alverson et al., "The Tera Computer System," *Proc. Int'l Conf. Supercomputing, ACM,* 1990, pp. 1-6.
6. S.W. Keckler and W.J. Dally, "Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism," *Proc. Int'l Symp. Computer Architecture,* ACM, 1992, pp. 202-213.
7. G.S. Sohi, S.E. Breach, and T. Vijaykumar, "Multiscalar Processors," *Proc. Int'l Symp. Computer Architecture,* ACM, 1995, pp. 414-425.
8. J.-Y. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 49-58.

processor achieves performance superior to that of several competing designs, such as superscalar, traditional multithreaded, and on-chip multiprocessor architectures. We believe that the efficiency of a simultaneous multithreaded processor makes it an excellent building block for future technologies and machine designs.

Several issues remain whose resolution should improve SMT's performance even more. Our future work includes compiler and operating systems support for optimizing programs targeted for SMT and combining SMT's multithreading capability with multiprocessing architectures.

Acknowledgments

We thank John O'Donnell of Equator Technologies, Inc. and Tryggve Fossum of Digital Equipment Corp. for the source to the Alpha AXP version of the Multiflow compiler. We also thank Jennifer Anderson of DEC Western Research Laboratory for copies of the SpecFP95 benchmarks, parallelized by the most recent version of the SUIF compiler, and Sujay Parekh for comments on an earlier draft.

This research was supported by NSF grants MIP-9632977, CCR-9200832, and CCR-9632769, NSF PYI Award MIP-9058439, DEC Western Research Laboratory, and several fellowships (Intel, Microsoft, and the Computer Measurement Group).

References

1. R. Alverson et al., "The Tera Computer System," *Proc. Int'l Conf. Supercomputing*, Assoc. of Computing Machinery, N.Y., 1990, pp. 1-6.
2. K. Farkas et al., "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," to appear in *Proc. 30th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE Computer Society Press, Los Alamitos, Calif., Dec. 1997.
3. S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. Int'l Symp. Computer Architecture*, ACM, 1997, pp. 206-218.
4. D. M. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading

Processor," *Proc. Int'l Symp. Computer Architecture*, ACM, 1996, pp. 191-202.
5. S. McFarling, "Combining Branch Predictors," Tech. Report TN-36, Western Research Laboratory, Digital Equipment Corp., Palo Alto, Calif., June 1993.
6. M.W. Hall et al., "Maximizing Multiprocessor Performance with the SUIF Compiler," *Computer*, Dec. 1996, pp. 84-89.
7. P.G. Lowney et al., "The Multiflow Trace Scheduling Compiler," *J. Supercomputing*, May 1993, pp. 51-142.
8. J.L. Lo et al., "Converting Thread-level Parallelism to Instruction-level Parallelism via Simultaneous Multithreading," *ACM Trans. Computer Systems*, ACM, Aug. 1997.

Susan J. Eggers is an associate professor of computer science and engineering at the University of Washington. Her research interests are computer architecture and back-end compilation, with an emphasis on experimental performance analysis. Her current focus is on issues in compiler optimization (dynamic compilation and compiling for multithreaded machines) and processor design (multithreaded architectures).

Eggers received a PhD in computer science from the University of California at Berkeley. She is a recipient of an NSF Presidential Young Investigator award and a member of the IEEE and ACM.

Joel S. Emer is a senior consulting engineer at Digital Equipment Corp., where he has worked on processor performance analysis and performance modeling methodologies for a number of VAX and Alpha CPUs. His current research interests include multithreaded processor organizations, techniques for increased instruction-level parallelism, instruction and data cache organizations, branch-prediction schemes, and data-prefetch strategies for future Alpha processors.

Emer received a PhD in electrical engineering from the University of Illinois. He is a member of the IEEE, ACM, Tau Beta Pi, and Eta Kappa Nu.

Henry M. Levy is professor of computer science and engineering at the University of Washington, where his research involves operating systems, distributed and parallel systems, and computer architecture. His current research interests include operating system support for simultaneous multithreaded processors and distributed resource management in local-area and wide-area networks.

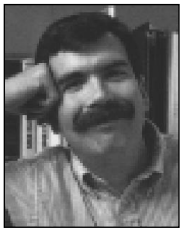
Levy received an MS in computer science from the University of Washington. He is a senior member of IEEE, a fellow of the ACM, and a recipient of a Fulbright Research Scholar award.

Jack L. Lo is currently a PhD candidate in computer science at the University of Washington. His research interests include architectural and compiler issues for simultaneous multithreading, processor microarchitecture, instruction-level parallelism, and code scheduling.

Lo received a BS and an MS in computer science from Stanford University. He is a member of the ACM.

Rebecca L. Stamm is a principal hardware engineer in Digital Semiconductor's Advanced Development Group at Digital Equipment Corp. She has worked on the architecture, logic and circuit design, performance analysis, verification, and debugging of four generations of VAX and Alpha microprocessors and is currently doing research in hardware multithreading. She holds 10 US patents in computer architecture.

Stamm received a BSEE from the Massachusetts Institute of Technology and a BA in history from Swarthmore College. She is a member of the IEEE and the ACM.



Dean M. Tullsen is a faculty member in computer science at the University of California, San Diego. His research interests include simultaneous multithreading, novel branch architectures, compiling for multithreaded and other high-performance processor architectures, and memory subsystem design. He received a 1997 Career SA ard from the National Science Foundation.

Tullsen received a PhD in computer science from the University of Washington, with emphasis on simultaneous multithreading. He is a member of the ACM.

Direct questions concerning this article to Eggers at Dept. of Computer Science and Engineering, Box 352350, University of Washington, Seattle WA 98115; eggers@cs.washington.edu.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 150 Medium 151 High 152