Simultaneous Multithreading: A Platform for Next-generation Processors

Susan J. Eggers, Joel Emer☸, Henry M. Levy, Jack L. Lo, Rebecca Stamm☸ and Dean M. Tullsen§

Dept. of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350
{eggers,levy,jlo}@cs.washington.edu

☸Digital Equipment Corporation
HLO2-3/J3
77 Reed Road
Hudson, MA 01749
{emer,stamm}@vssad.enet.dec.com

§Dept. of Computer Science and Engineering
9500 Gilman Drive
University of California, San Diego
LaJolla, CA 92093-0114
tullsen@cs.ucsd.edu

1 Introduction

With the dizzying pace of semiconductor technology development, CPU designers are squeezing previously unimaginable amounts of hardware onto a single chip. Over the next 15 years we can expect the number of transistors on a chip to increase by two orders of magnitude, to a billion transistors. The obvious question, then, is how to use these transistors. One possibility is to add more memory (either cache or "primary memory") to the chip; however, there is a limit to the performance that can be gained by the addition of memory alone. Another approach is to increase the level of "systems integration," bringing onto the chip all of the support functions that we now find off chip (for example, graphics accelerators, I/O controllers and interfacing), thereby decreasing communication costs. This decreases system cost, but also has only limited impact on performance.

Monumental performance improvements can only be achieved by increasing the computational capabilities of the processor. In general, this means increasing *parallelism*, in perhaps several (or *all*) of its available forms. Current superscalar processors, for example, can execute four or more instructions per cycle; in practice, however, they sustain much less than that -- perhaps closer to one or two instructions per cycle -- because applications have insufficient parallelism (due to inter-instruction dependences and long-latency instructions) to fill the CPU's resources. Placing multiple superscalar processors on a chip, another design alternative, will suffer a similar fate, and as we will show, suffers from other problems as well. Ultimately, then, we must produce both hardware able to exploit high degrees of parallelism, and an execution workload capable of feeding it.

In this article we describe *simultaneous multithreading (SMT)*, a processor design that can consume parallelism of any type -- thread-level parallelism (from either multi-threaded parallel programs or individual programs in a multiprogrammed workload) and instruction-level parallelism (from a single program or thread) -- to maintain high processor utilization and increase

workload performance. Equally important, simultaneous multithreading adds minimal hardware complexity to, and is a straightforward extension of, today's advanced dynamically-scheduled microprocessors. Our experiments show that with the addition of SMT, an eight-wide superscalar executing a multiprogrammed workload can double its throughput; similarly, a parallelized program can execute in half the time. And yet the performance of a single application running in single-threaded mode is degraded by less than 2%. In the near future, an SMT processor on a chip will be achievable, and looking further ahead, multiple SMT processors per chip could be envisioned.

Simultaneous multithreading combines hardware features seen in two other types of processors: wide-issue superscalars and multithreaded processors. From superscalars it inherits the ability to issue multiple instructions each cycle; and like multithreaded processors it can execute several programs (or threads) at once. The result is a processor that can issue multiple instructions from multiple threads each cycle.

The difference between superscalar, multithreading, and simultaneous multithreading is pictured in Figure 1, which shows example execution sequences for the three types of architectures. In these figures, each row represents a single cycle of execution; the four boxes show four potential instructions that the processor could issue each cycle. A filled box indicates that the processor was able to find an instruction to execute in that issue slot on that cycle, while an empty box shows an unused or wasted instruction issue slot. We characterize the wasted issue slots as being of two types. *Horizontal waste* occurs when some, but not all, of the issue slots in a cycle can be used. This is typically due to a lack of instruction-level parallelism in the program. *Vertical waste* occurs when a cycle goes completely unused. This can be caused by a long latency instruction (such as a memory read) that is holding back further instruction issue.

A standard superscalar, such as the DEC Alpha 21164 or 21264, HP PA-8000, Intel Pentium Pro, MIPS R10000, PowerPC 604 or UltraSPARC 1 [1] appears in Figure 1a. As in all superscalars, it is executing a *single* program, or thread, from which it attempts to find multiple instructions per cycle to issue. When it cannot, the issue slots go unused, and it incurs both horizontal and vertical waste. Multithreaded architectures, for example, the Tera [2], contain hardware state -- a program counter and registers -- for several threads. On any given cycle the processor executes instructions from one of the threads. On the next cycle, it switches to a different thread context, so that it can execute instructions from the new thread. Context switching between these threads can be done in a single cycle, because the multiple hardware contexts obviate the need for saving and restoring processor

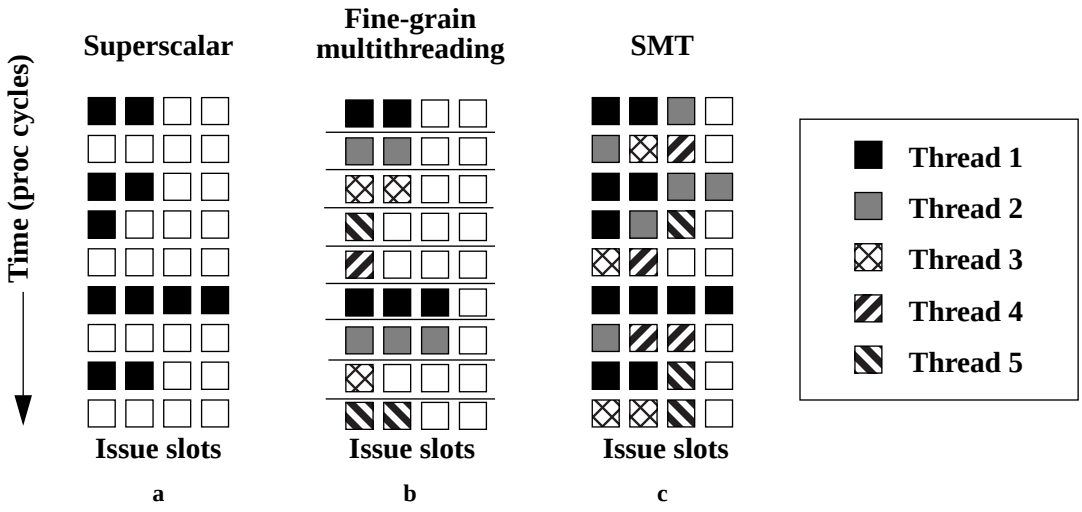

Figure 1: Comparison of issue slot (functional unit) partitioning in various architectures.

state. Thread context switching and the resulting execution pattern are shown in Figure 1b. The different stipple patterns represent different threads, which issue instructions in different cycles. The figure illustrates the primary advantage of multithreading, namely, its better tolerance of long-latency operations, because it can schedule another thread to run while one is stalled. Note, however, that while the multithreaded architecture has effectively eliminated vertical waste, it has increased horizontal waste, by converting some of the vertical waste into horizontal waste. Consequently, as instruction issue width continues to increase, multithreaded architectures will ultimately suffer the same fate as superscalars, i.e., they will be unable to find enough instruction-level parallelism in a single thread to effectively utilize the processor.

Simultaneous multithreading, shown in Figure 1c, combines the best features of multithreading and superscalar architectures. Like a superscalar, SMT can exploit instruction-level parallelism in one thread by issuing multiple instructions each cycle; like a multithreaded processor, it can hide long latency operations by executing instructions from different threads. The difference is that it can do both at the same time, that is, in the same cycle. Each cycle an SMT processor selects instructions for execution from *all* threads. The processor dynamically schedules all machine resources among the threads, providing the greatest chance for the highest hardware utilization. If one thread has a high level of instruction-level parallelism, that parallelism can be satisfied; if multiple threads each have low levels of instruction-level parallelism, they can be executed together to compensate for the low ILP in each. Consequently, simultaneous multithreading has the potential to recover issue slots lost to *both* horizontal and vertical waste.

The result is better performance for a variety of workloads. For a mix of independent programs (multiprogramming), the overall throughput of the machine is improved. When one program has no instructions that are ready to issue, instructions can be found from one of the others. Similarly, programs that are parallelizable, either by a compiler or a programmer, reap the same throughput benefits; but here the outcome is a *decrease* in execution time for the application. Finally, programs that must execute as a single thread, i.e., that cannot be parallelized, have all machine resources available to them and maintain roughly the same level of performance as when executing on a single-threaded processor.

The following sections describe simultaneous multithreading and present an implementation model for a simultaneous multithreaded processor. We show that implementing simultaneous multithreading is surprisingly straightforward, given today's advanced superscalar processors -- hardware designers can focus on building a fast single-thread superscalar, and add SMT's multi-thread capability on top. We also present simulation-based performance results that demonstrate the benefits of simultaneous multithreading when compared with superscalar, multithreaded, and on-chip multiprocessor architectures.

Given the enormous transistor budget in the next computer era, we believe that SMT provides a base technology that can be used in many ways to extract improved performance. For example, for wider superscalars, SMT provides a multithreaded workload to guarantee high utilization of functional units. For multiple CPUs on a chip, a small number of SMTs can be used side-by-side to achieve identical performance to a larger number of conventional superscalars. For increased on-chip cache, SMT can take great advantage of that memory due to its high execution rate. Overall, SMT benefits from all types of parallelism, both instruction-level parallelism and thread-level parallelism. We believe that simultaneous multithreading will provide a significant boost in processor performance for next-generation processors, solving many of the performance challenges that current design technologies present.

2 An Implementation Model for Simultaneous Multithreading

At first glance, simultaneous multithreading may sound complex: the processor must support multiple hardware contexts (threads) and be capable of fetching and issuing instructions from multiple contexts in a single cycle. In fact, we will show that necessary modifications should be straightforward. In this section we present a high-level design for an SMT implementation, effectively showing how a superscalar can be converted into an SMT engine.

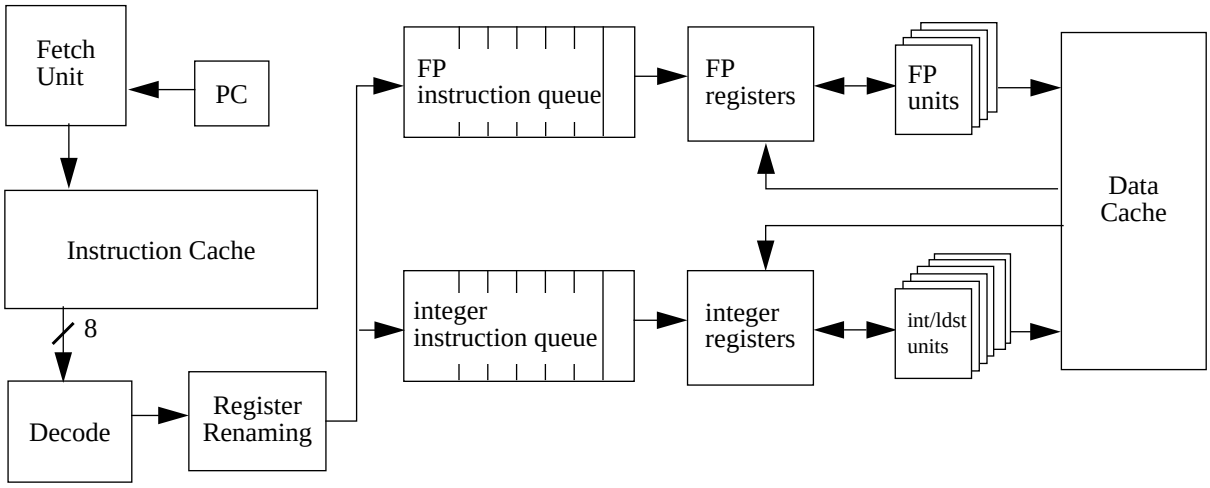

Figure 2: Organization of the dynamically-scheduled superscalar processor on which SMT is based.

2.1 A Straightforward Extension of Conventional Superscalars

Our primary design goal for a simultaneous multithreaded processor was to achieve good instruction throughput, but at the same time have minimal impact on the microarchitecture of a conventional, wide-issue superscalar. Our current SMT model is derived from a high-performance, out-of-order, superscalar architecture, whose dynamic scheduling core is similar to the MIPS R10000 [1]. (The organization of the superscalar core is illustrated in Figure 2; a glossary of terms appears in Table 1.) On each cycle, the superscalar fetches eight instructions from the instruction cache. After the instructions are decoded, the register renaming logic maps the architectural registers to the hardware renaming registers to remove false dependences. Instructions are then fed to either the integer or floating point instruction queues. When their operands become available, they are issued from these queues to their corresponding functional units. To support out-of-order execution, the processor must track instruction and operand dependences, in order to determine which instructions can issue (those without dependences), and which must wait for previously-issued instructions to finish (those with dependences). After completing execution, the instructions are retired in-order, and hardware registers that are no longer needed are freed.

Our SMT architecture, which can simultaneously execute threads from up to eight hardware contexts, is a straightforward extension of this conventional superscalar processor. Some resources were replicated to support SMT, namely, state for the hardware contexts (registers, program counters, subroutine stacks) and per-thread mechanisms for pipeline flushing, instruction retirement, and trapping. All are resources that currently exist on today's processors; in an SMT they are simply replicated for each thread. We also widened the instruction issue width of SMT beyond the current superscalar capability of 4-6 instructions per cycle. The additional width enables SMT to issue more instructions from the different threads and gives a single thread with large amounts of instruction-level parallelism an opportunity to exploit that parallelism. Finally, per thread identifiers were added to the branch target buffer and TLB. Only one component, instruction fetching, was redesigned to be compatible with SMT's multi-thread instruction issue.

Significantly, no special hardware is needed for scheduling instructions from the different threads onto the functional units. Conventional dynamic scheduling hardware in today's out-of-order superscalars can perform simultaneous multithreaded scheduling. The register renaming hardware removes inter-thread register name conflicts, by mapping the thread-specific architectural registers onto the hardware registers. Instructions from all threads are placed together in the instruction queues.

When their operands become available, they are issued to functional units, without regard to which thread they came from.

This minimal redesign has two important consequences. First, since most hardware resources are still available to a single thread executing alone, SMT provides good single-thread performance, in addition to its benefits for multiple threads. Second, because the changes to enable SMT are minimal, the commercial transition from today's superscalars to SMT should be fairly smooth.

Term	Definition
software threads	Independent instruction streams.
hardware threads	Processor hardware that contains the state of executing threads, namely the PC and the register file; sometimes called hardware contexts.
in-order execution	Instructions are issued to functional units and executed in the order the compiler has generated them and the fetch hardware fetches them. This is also known as the program order.
out-of-order execution (also called dynamic scheduling)	Instructions are issued to functional units and executed as soon as their operands have been calculated or loaded from memory (assuming the appropriate functional unit is available), even if previously fetched instructions have not executed. Out-of-order execution provides better performance than executing instructions in their program order.
register renaming	A hardware technique that maps the registers defined by a machine's architecture to the actual (physical) hardware registers. It is used to increase the number of registers available to a program and therefore the number of program instructions that can be executed in parallel. In the SMT processor we describe, each hardware context has 32 physical registers, plus some fraction of the 100 renaming registers, which are dynamically allocated to threads as needed.
false dependences	Because a program's execution contains many more values than available architectural registers, the compiler (specifically, the register allocator) reuses registers for the different values. This may impose artificial dependences between operands that are not computationally dependent. For example, two calculations that have the same (architectural) destination register have a false output dependence. This dependence can be removed by mapping the two destinations to different (physical) hardware registers. On an out-of-order processor, register renaming can therefore expose more parallelism.
instruction retirement	Instruction retirement consists of committing the changes to processor state that are caused by executing the instructions. For a dynamically-scheduled processor, such as the one used in this study, this can include deallocating a hardware register, removing instructions from the processor pipeline, and updating branch prediction information. A dynamically-scheduled processor executes instructions out-of-order, but to properly support activities such as debugging, interrupts, and exceptions, the processor most provide the illusion of behaving as an in-order processor. This can be guaranteed by retiring instructions in program order.
branch prediction	Modern processors include hardware that tries to predict whether a branch will be taken. If the prediction is wrong, the processor must discard the instructions that have been wrongly fetched, and fetch the correct ones. The resultant delay in execution is called the misprediction penalty.
speculative instructions	Instructions that enter the pipeline before it is known that they should be executed. An example is the instructions that follow a branch that is predicted taken. Until the branch is actually executed, it is not known whether its subsequent instructions will be needed. We call instructions that are not speculative, useful instructions.

Table 1: An explanation of terminology used in this article.

However, should SMT's implementation negatively impact either the targeted processor cycle time or the time to design completion, several different approaches could be taken to simplify it. Since most of SMT's implementation complexity stems

from its wide-issue superscalar underpinnings rather than from the hardware needed to realize simultaneous multithreading *per se*, many of the alternatives involve altering the superscalar implementation. For example, the functional units could be partitioned across a duplicated register file, as is done in the Alpha 21264 [1]. The technique halves the number of read ports (for each copy of the register file), at no cost in intra-partition register accesses and only a slight cost when passing values between partitions. For caches, the age-old technique of interleaving, augmented with multiple independently-addressed banks [7], can increase the number of simultaneous accesses without also increasing the number of cache ports. Wilson, *et al.* suggest placing recently accessed data in a small associative buffer, which is accessed when the main cache ports are busy [12]. Additional cache ports can also be provided by using wave-pipelining techniques, as in the 21264, which provides dual ports by starting a new access on each half-clock cycle. At last resort a less aggressive superscalar issue width (e.g., 8 functional units rather than 10) could be used to reduce the number of register and cache ports. This last alternative would have the most impact on performance.

2.2 Instruction Fetching on an SMT Processor

In a conventional processor, the instruction unit is responsible for fetching blocks of instructions from a single thread into the execution unit. The main performance issues revolve around maximizing the number of *useful* instructions that can be fetched (e.g., minimizing branch mispredictions) and fetching independent instructions quickly enough to keep functional units busy. An SMT processor places additional stress on the fetch unit. First, it is responsible for fetching instructions from up to eight *different* threads. And second, it has a harder time keeping up with its more efficient[1] dynamic scheduler, which can issue more instructions each cycle, because it takes them from multiple threads. Consequently, the fetch unit is SMT's performance bottleneck.

Surprisingly, the inter-thread competition for instruction bandwidth is also a vehicle for obtaining *better* performance. First, an SMT fetch unit can partition the instruction fetch bandwidth among the competing threads. Because of branch instructions and cache line boundaries, the fetcher has difficulty filling the issue slots each cycle from only a single thread. We fetch from two threads, in order to increase the probability of fetching only useful (nonspeculative) instructions. Second, the instruction fetcher can be smart about which threads it fetches, fetching those that will provide the most immediate performance benefit. To take advantage of both opportunities, we propose a fetch unit customized to take advantage of SMT's unique ability to fetch and issue instructions from multiple threads in the same cycle.

The instruction fetching hardware has at its disposal eight program counters, one for each thread context. On each cycle, the fetch mechanism selects two threads (among threads not already incurring I-cache misses) and fetches eight instructions from each thread. To match the lower instruction width of the issue hardware, a subset of these instructions is then chosen for decoding: instructions are taken from the first thread until a branch instruction or the end of a cache line is encountered; the remainder come from the second thread. (We call this scheme **2.8**; it is described in more detail in [10].) Because the instructions come from two different threads, there is a greater likelihood of fetching useful instructions -- 2.8's performance was 10% better than fetching from only one thread at a time. The hardware cost is an additional port on the instruction cache and logic to locate the branch instruction, none out of the question for future processors. Fetching from multiple threads while limiting the fetch bandwidth to eight instructions is a less hardware-intensive alternative, but it has lower performance. For example, 2.8 saw a 5% improvement over fetching 4 instructions from each of 2 threads.

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

1. relative to an out-of-order superscalar

Not all threads provide equally useful instructions in a particular cycle; and an SMT processor can obtain better performance by predicting which threads will produce the fewest delays. Our thread selection hardware, called the **Icount** feedback technique (also described in [10]), gives the highest priority to the threads that have the *least* number of instructions in the decode, renaming, and queue pipeline stages (pictured in Figure 3). The technique benefits performance in several ways. First, it replenishes the instruction queues with instructions from the fast-moving threads, avoiding those that will fill the queues with instructions that are dependent upon and consequently blocked behind a long latency instruction. Second and most importantly, it maintains in the queues a fairly even distribution of instructions among these fast-moving threads, thereby increasing inter-thread parallelism (and the ability to hide each others' latencies). And last, 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. The net result is that, although up to 8 threads are sharing and competing for slots in the instruction queues, the percentage of cycles that the instruction queue is full is actually *less* than on the single-thread superscalar (8% versus 21%). All that is required to support Icount is a small amount of additional logic that increments/decrements per-thread counters when instructions enter the decode stage/exit the instruction queues and picks the two minimum. Our experiments found that Icount outperformed alternative schemes that addressed a particular cause of instruction queue inefficiency, such as minimizing branch mispredictions by giving priority to threads with the fewest outstanding branches or minimizing load delays by giving priority to threads with the fewest outstanding on-chip cache misses. Icount works because it addresses *all* causes of instruction queue inefficiency.

2.3 The Effect of Large Register Files in an SMT Processor

Following instruction fetch and decode, register renaming is performed, as in the superscalar processor. Each thread can address 32 architectural integer (and FP) registers. The register renaming mechanism maps these architectural 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 an increase in the processor cycle time, the SMT processor pipeline was extended by two cycles to allow two-cycle register reads and two-cycle writes. Figure 3 compares SMT's pipeline to that of the single-thread superscalar. On the first SMT register read cycle, data is read from the register file into a buffer. (In that same cycle, the instruction is also sent to a similar buffer.) Then in the next cycle data is sent to a functional unit for execution. Writes to the register file behave in a similar manner, also using an extra pipeline stage.

The two-stage register access has several ramifications on the architecture. First, it lengthens the pipeline distance between *fetch* and *exec*, increasing the branch misprediction penalty by 1 cycle. Mispredicted instructions consume instruction queue slots, renaming registers and possibly issue slots, all of which could be used by other threads on an SMT processor. Second, the two extra stages between *rename* and *commit* increase the minimum time that a hardware register is held by an executing

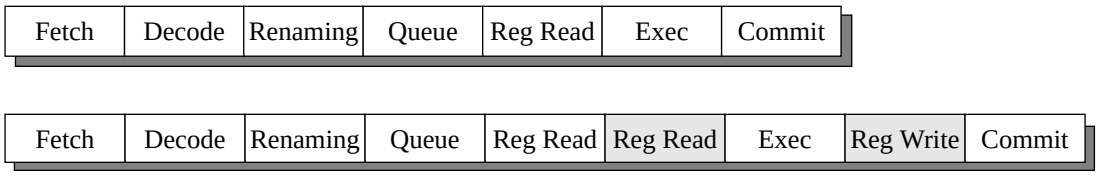

Figure 3: Comparison of the pipelines for a conventional superscalar processor (top) and SMT (bottom). The SMT pipeline is longer, because an additional cycle is needed to access the register file. For both pipelines the execute stage is a variable number of cycles, depending on the operation.

instruction, increasing the pressure on the renaming registers. Third, the extra cycle needed to write back results to the register file requires an extra level of bypass logic.

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

2.4 The Simulation Configuration

SMT's implementation parameters will, of course, change as technologies shrink. Our current parameters target an implementation realizable roughly two to three years in the future. The exact parameter values are shown in the following list.

- an 8 instruction fetch/decode width.
- 6 integer units, 4 of which can load/store from/to memory.
- 4 floating point units.
- 32 entry integer and floating point instruction queues.
- hardware contexts for 8 threads.
- 100 additional integer renaming registers and 100 additional floating point renaming registers.
- retirement of up to 12 instructions per cycle.
- 128KB, 2-way associative, level 1 instruction and data caches; the D-cache has 4 dual-ported banks, while the I-cache has 8 single-ported banks. The access time per bank is 2 cycles.
- a 16MB, direct-mapped, unified level 2 cache; the single bank has a transfer time of 12 cycles on a 256-bit bus.
- latency to memory is 80 cycles on a 128-bit bus.
- all caches have 64 byte blocks.
- 16 outstanding cache misses are allowed.
- the data and instruction TLBs contain 128 entries each.
- McFarling-style branch prediction hardware [1]: a 256-entry, 4-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 2048-entry local history table that indexes into a 4096-entry prediction table.)

3 Experimental Methodology

We compared simultaneous multithreading with its two ancestral processor architectures, wide-issue superscalars (SS) and fine-grain multithreaded superscalars (FGMT). Both are single-processor architectures whose purpose is 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 ILP, hide latencies of one thread by switching to and executing instructions from another, 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 (MP). We examined both two- and four-processor MPs, partitioning the scheduling unit resources of their CPUs (the functional units (and therefore the issue width), instruction queues, and renaming registers) differently for each case. In the two-processor MP (MP2), each processor received half of SMT's execution resources, so that the total resources of the two were comparable). For a four-processor MP (MP4), each processor contains approximately one-fourth of SMT's chip resources (we rounded up when necessary, giving the MP a slight

advantage). Within the MP design space, these two alternatives (MP2 and MP4) represent an interesting trade-off between thread-level and instruction-level parallelism. The two-processor machine can exploit more ILP, because each processor has more functional units than its MP4 counterpart, while MP4 has additional processors to take advantage of more TLP.

The simulators for the three alternative processors reflect the architecture described in the previous section, but without the SMT extensions. In particular, they use single-threaded fetching (per processor) and the shorter pipeline[2], without simultaneous-multithreaded issue. However, they have the SMT memory system described in Section 2.4. The fine-grain multithreaded processor simulator context switches between threads each cycle in a round robin fashion for instruction fetch, issue and retirement. (Refer to Table 2 for a summarized comparison of all architectures studied.) All processor simulators were

Features	Super-scalar	MP2	MP4	Fine-grain Multithreading	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)
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
Multi-thread fetch algorithm	n.a.	n.a.	n.a.	round-robin	ICOUNT

Table 2: A comparison of the architectures.

execution-driven, cycle-level simulators; they modeled the processor pipelines and memory subsystems (including inter-thread contention for all structures in the memory hierarchy and the busses between them) in great detail.

We evaluated simultaneous multithreading on two types of workloads, one a multiprogramming workload consisting of several single-threaded programs, and the other a group of parallel (multi-threaded) applications. We used both types, because each exercises different parts of an SMT processor. For example, the larger (workload-wide) working set of the multiprogrammed 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. The programs in the multiprogramming workload were chosen from the SPEC95 benchmark suite (the integer programs compress, go, ijpeg, li and perl), plus several floating point SPLASH2 programs (fft, lu, radix), all run in single-thread mode. When benchmarking SMT, each of the programs executed as a separate thread. To eliminate the effects of benchmark differences when simulating fewer than 8 threads, each data point in the results figures and tables in section 4 was composed of at least 4 simulation runs, where each of the runs used a different combination of the benchmarks.

The parallel workload was used for two reasons. First, it consists of coarse-grain (parallel threads) and medium-grain (parallel loop iterations) parallel programs that were written for shared memory machines, and therefore serves as a fair basis for eval-

2. The fine-grain multithreaded processor uses the longer pipeline. Although it could be implemented with separate (and therefore smaller) register files for each context, it still requires the same number of register ports and similar wire lengths to the functional units as SMT.

uating the MPs. Second, it 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, for example, a need for the same functional units at the same time. Consequently, there is potentially more contention for these resources than in a multiprogramming workload. The applications were selected from the SPLASH2 (fft, lu, radix, water-nsquared, water-spatial) and SPEC95 (applu, hydro2d, mgrid, su2cor, swim, tomcatv, turb3d) 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 larger data sets. The SPEC programs were parallelized with the SUIF compiler [13], using parallelization policies that were developed for shared memory machines.

The multiprogramming workload was compiled with cc; the parallel benchmarks were compiled with the Multiflow compiler [5], which has been modified to produce Alpha executables.[3] For all programs most compiler optimizations were set to maximize each program's performance on the superscalar, for example, by taking advantage of deep loop unrolling, and, for the Multiflow compiles, instruction scheduling for an eight-wide machine. However, trace scheduling was disabled, so that speculation could be guided by the branch prediction and out-of-order execution hardware.

4 The Performance Results

Simultaneous multithreading outperformed the other processor architectures that were also designed to increase instruction throughput: single-threaded superscalars, fine-grain multithreaded superscalars and single-chip, shared-memory multiprocessors, whose processors were also superscalars. In addition to the better speedups and instruction throughput, two potential performance pitfalls did *not* occur. First, SMT's extended pipeline had only a minor impact on single-thread execution, slowing down our single-threaded programs executing alone by less than 2% when compared to the eight-wide superscalar with the shorter pipeline. Second, inter-thread conflicts for several shared resources, such as the caches, TLBs and branch prediction hardware, were offset by SMT's ability to hide the additional delays. This section discusses those performance results.

4.1 Simultaneous Multithreading

4.1.1 Multi-thread and single-thread performance

Simultaneous multithreading's instruction throughput was much higher than the one to two instructions per cycle normally reported for current wide-issue superscalars (see Tables 3 and 4). Instruction throughput consistently rose as the number of threads increased, reaching a maximum of 6.2 for the multiprogramming workload and 6.1 for the parallel applications at 8 threads. SMT's speedups for parallel applications at 8 threads averaged 1.8 over a one-thread SMT, demonstrating its ability as a parallel processor.

Recall that, in order to absorb the longer access of its large register file, SMT's pipeline is two cycles longer than that of the superscalar and the processors of the single-chip multiprocessor (see Section 3.2). Therefore, a single thread executing on an SMT will see additional latencies from a longer mispredicted branch penalty and less renaming register availability. Despite this potential for lower performance, SMT's single-thread performance was only 1% (parallel workload) and 1.5% (multiprogrammed workload) worse than the single-threaded superscalar. Accurate branch prediction hardware and the shared pool of renaming registers prevented the additional penalties from occurring frequently.

4.1.2 Contention for shared resources

Many of SMT's hardware data structures, such as the caches, TLBs and branch prediction tables, are shared by all threads. The

3. We thought it important that all programs in each workload be compiled with the same compiler; cc was needed for the integer programs in the multiprogrammed workload, because our version of the Multiflow compiler does not handle varargs.

Threads	Super-scalar	Multi-thread	SMT
1	3.17	3.12	3.12
2		3.52	4.24
4		3.66	5.76
6		3.17	5.88
8		2.83	6.17

Table 3: Instruction throughput (instructions/cycle) for the multiprogramming workload.

Threads	Super-scalar	MP2	MP4	Multi-thread	SMT
1	3.44	2.49	1.49	3.40	3.40
2		4.38	2.61	4.18	4.73
4			4.29	4.28	5.67
6				4.03	5.95
8				3.29	6.10

Table 4: Instruction throughput (instructions/cycle) for the parallel workload.

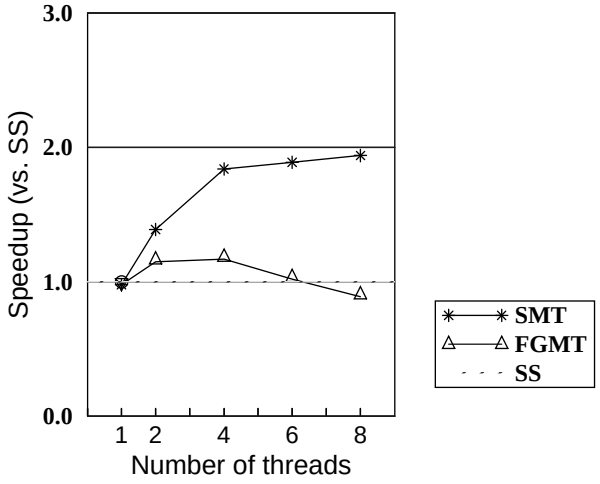

Figure 4: Speedups (normalized to SS) for the multiprogramming applications.

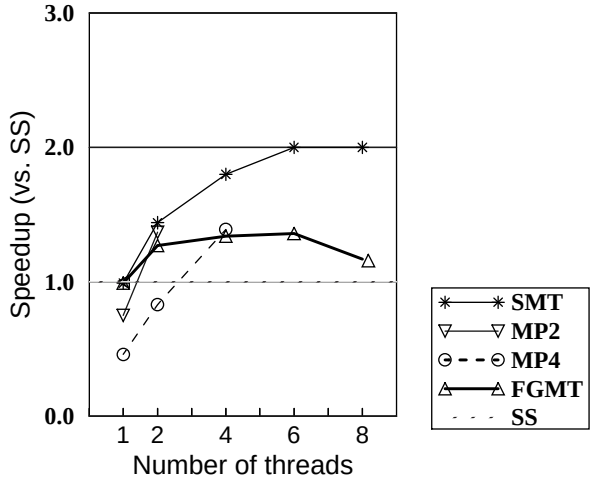

Figure 5: Speedups (normalized to SS) for the parallel applications.

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 down side, inter-thread usage leads to competition for the shared resources, driving up cache and TLB misses and branch mispredictions. In this section we quantify the additional conflicts and assess their effect on performance.

We examined the impact of inter-thread interference on the shared hardware data structures.[4] The impact 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)

4. Recall that the simulations modeled all inter-thread contention for each level of the memory hierarchy and the busses between the levels.

and 66% (multiprogramming workload) and branch mispredictions by 50% (parallel workload) and 27% (multiprogramming workload), as the number of threads increased from 1 to 8. We also measured execution time when treating the inter-thread misses as though they were hits, i.e., eliminating their negative effect on performance. Both sets of results, with and without an inter-thread conflict penalty, were within 1%. The additional conflicts of the larger working sets were absorbed by SMT in several ways: first, many of the L1 misses were covered by the fully pipelined 16MB L2 cache whose latency was only six times that of the L1 cache; second, TLB and branch prediction conflicts did not occur frequently enough to severely impact overall performance; and finally, the additional sources of latency were hidden by executing instructions from other threads. This last factor is the most important: although SMT introduces additional conflicts for the shared hardware structures, it also has an increased ability to hide them

4.2 A Comparison to Superscalars and Fine-grain Multithreaded Superscalars

The single-thread superscalar fell far short of SMT's performance. For the multiprogrammed workload its instruction throughput averaged 3.2 instructions per cycle, out of a potential of eight; the parallel workload had only slightly higher average throughput, 3.4. Consequently, SMT executed both types of workloads almost twice as fast (at 8 threads). The superscalar's inability to exploit more instruction level parallelism and any task-level parallelism (and consequently hide horizontal and vertical waste) are responsible for its lower performance.

By eliminating vertical waste, the fine-grain multithreaded architecture provided speedups over the superscalar as high as 1.2 on both workloads. However, this maximum speedup occurred at only 4 threads, and with additional threads performance fell. Two factors were responsible. First, fine-grain multithreading can only eliminate vertical waste, and, given the latency-hiding capability of its out-of-order processor and lockup-free caches, four threads were sufficient to do that. Second, as in SMT, fine-grain multithreading suffers from inter-thread competition for the shared processor resources. However, unlike SMT, it is less able to hide the additional conflicts, because it can only issue instructions from one thread each cycle. Neither limitation applies to simultaneous multithreading, which can simultaneously exploit instruction-level and task-level parallelism to reduce both horizontal and vertical waste, and can hide latencies from inter-thread conflicts by simultaneously issuing instructions from different threads. Consequently, SMT obtained higher instruction throughputs and greater program speedups than fine-grain multithreading when executing multiple threads, at all numbers of threads.

4.3 A Comparison to Single-chip Shared Memory Multiprocessors

Simultaneous multithreading obtained better speedups than the multiprocessors, not only when simulating the machines at their maximum-thread capability (8 threads for SMT, 4 for MP4, 2 for MP2), but also for a given number of threads (see Figure 5). At maximum-thread capability SMT's throughput reached 6.1 instructions per cycle, compared to 4.4 for MP2 and 4.3 for MP4. These results have an impact for the implementation of these machines, as well as their performance. Because of their narrower issue width, the MPs could very well be built with a shorter cycle time. The speedups indicate that the MP's cycle time must be less than 69% that of an SMT before it obtains comparable performance.

Better speedups on the MPs were hindered by the fixed partitioning of their hardware resources across processors. Static partitioning prevents the MPs from responding well to changes in levels of ILP and TLP in the executing programs. For example, when TLP was less than the number of processors, several processors lay idle; the MPs also had difficulty taking advantage of large amounts of ILP in the unrolled loops of individual threads, because of their narrower issue width. An SMT processor, on the other hand, dynamically partitions its resources, and therefore can respond well to variations in both ILP and TLP, allowing them to be exploited interchangeably. When only one thread is executing, (almost) all machine resources can be dedicated to it; and additional threads (more TLP) can compensate for a lack of ILP in any single thread.

To understand how the static partitioning hurt MP performance, we measured the number of cycles in which a processor could

have used an additional hardware resource, and, in the same cycle, that resource lay idle in another processor. (In an SMT the idle resource would have been used by another thread.) The results appear in Figure 6, where each stipple pattern represents a different resource. Static partitioning of the integer units, for both arithmetic and memory operations, were responsible for most of the MP's inefficient use of resources. The floating point units were also a bottleneck for MP4 on this largely floating point-intensive workload. Selectively increasing the MP hardware resources to match that on the SMT eliminated a particular bottleneck, but did not improve the speedups, because the bottleneck simply shifted to a different resource. Only when *each* MP processor was given *all* the hardware resources of an SMT did the MPs obtain greater speedups; and this occurred only when the processors executed the same number of threads. At maximum thread capability the SMT still did better.

5 Related work

Improved performance for the next-generation of processors will depend heavily on the ability to exploit any and all types of parallelism. We will limit our discussion of related work to the more recent SMT studies and several architectures that represent alternative approaches to exploiting parallelism.

In Tullsen, *et al.*, [11] we evaluated the potential of SMT, comparing performance of SMT with superscalars, multithreaded processors, and multiprocessors, using a more abstract processor model and an older (SPEC92) workload. We then presented a realizable architecture for SMT and investigated the fetch bottlenecks in such a system [10]. In addition to our previous work on SMT, Gulati and Bagherzadeh [4] also proposed extensions to superscalar processors to implement simultaneous multithreading. In contrast to our processor model, their base processor was a 4-issue machine with fewer functional units, which limited the speedups they obtained when using additional threads. Yamamoto and Nemirovsky [14] evaluated an SMT architecture with separate instruction queues for up to 4 threads.

Thread or task-level parallelism is also essential to other next-generation architectures. As discussed in this paper, single-chip multiprocessing exploits this type of parallelism with additional processors. Olukotun, *et al.*, [6] investigated design tradeoffs for a single-chip multiprocessor and compared the performance and estimated area of this architecture with 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 [2] is a fine-grain multithreaded processor, which issues up to three operations each cycle. The M-Machine [3] utilizes multithreading in a different manner, relying on two levels of parallelism, called H-threads and V-threads. H-threads are composed of a sequence of LIW operations that exploit ILP, and

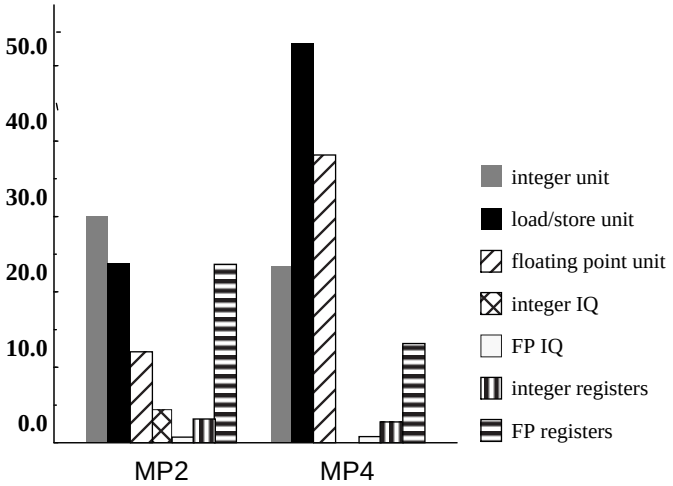

Figure 6: Frequencies of partitioning inefficiencies

are run on one of the machine's three clusters of functional units. A larger granularity of parallelism can be exploited by grouping multiple H-threads together into a V-thread and running the V-thread across all clusters. One could think of the M-Machine as a coarser-grain, compiler-driven SMT.

Threads can also be used in a speculative manner to exploit both task-level and instruction-level parallelism, as in the Multiscalar [8] and superthreaded [9] architectures. In the Multiscalar, tasks (which can be as big as a collection of basic blocks) can be speculatively executed by hardware using dynamic branch prediction techniques. A circular queue of processing units execute different tasks. Hardware support is provided to squash tasks if control (branches) or data (memory) speculation is incorrect. The superthreaded architecture also executes multiple threads concurrently, but does not speculate on data dependences. Run-time data dependence checking is performed, and hardware support is provided for control speculation.

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

6 Conclusions

Simultaneous multithreading is a processor design that permits the CPU to issue multiple instructions from multiple threads each cycle. SMT attacks multiple sources of lost resource utilization in wide-issue processors. Using both instruction-level and thread-level parallelism, it increases throughput for multiprogrammed workloads and improves speedup for parallel programs. Our measurements show how an SMT processor achieves superior performance to competing designs, such as superscalar, multithreaded, and shared-memory multiprocessor architectures. Hence SMT appears to be a highly-effective approach to benefit from the huge increase in chip densities we will see in the coming years.

We have described how an SMT processor can be achieved via straightforward modifications to modern state-of-the-art dynamic superscalars. In the future, we believe that SMT processors, and later multiple SMT processors per chip, will permit full and effective utilization of the resources we can deliver with future technologies.

Acknowledgments

We would like to 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 owe thanks to Jennifer Anderson of DEC Western Research Laboratory for copies of the SPEC FP95 benchmarks, parallelized by the most recent version of the SUIF compiler. This research was supported by NSF grants MIP-9632977, CCR-9200832 and CCR-9632769, NSF PYI Award MIP-9058439, NSF DEC Western Research Laboratory, and several fellowships (Intel, Microsoft and the Computer Measurement Group).

References

[1] Microprocessor Report, April 18, 1994 (PowerPC 604), May 30, 1994 (UltraSPARC 1), October 3, 1994 (HP PA-8000), October 24, 1994 (R10000), Feb. 16, 1995 (Intel Pentium Pro), Oct. 28, 1996 (DEC Alpha 21264).

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

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

[4] M. Gulati and N. Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. In *2nd International Symposium on High-Performance Computer Architecture*, February 1996.

[5] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell and J.C. Ruttenberg. The Multiflow trace

scheduling compiler. In *Journal of Supercomputing*, May 1993.

[6] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[7] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

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

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

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

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

[12] K.M. Wilson, K. Olukotun and M. Rosenblum. Increasing cache port efficiency for dynamic superscalar microprocessors. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[13] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, December 1994.

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