Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading

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

Dept. of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350
{jlo,eggers,levy}@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
La Jolla, CA 92093-0114
tullsen@cs.ucsd.edu

A version of this paper will appear in ACM Transactions on Computer Systems, August 1997.

Abstract

To achieve high performance, contemporary computer systems rely on two forms of parallelism: instruction-level parallelism (ILP) and thread-level parallelism (TLP). Wide-issue superscalar processors exploit ILP by executing multiple instructions from a single program in a single cycle. Multiprocessors (MP) exploit TLP by executing different threads in parallel on different processors. Unfortunately, both parallel-processing styles statically partition processor resources, thus preventing them from adapting to dynamically-changing levels of ILP and TLP in a program. With insufficient TLP, processors in an MP will be idle; with insufficient ILP, multiple-issue hardware on a superscalar is wasted.

This paper explores parallel processing on an alternative architecture, simultaneous multithreading (SMT), which allows multiple threads to compete for and share all of the processor's resources every cycle. The most compelling reason for running parallel applications on an SMT processor is its ability to use thread-level parallelism and instruction-level parallelism interchangeably. By permitting multiple threads to share the processor's functional units simultaneously, the processor can use both ILP and TLP to accommodate variations in parallelism. When a program has only a single thread, all of the SMT processor's resources can be dedicated to that thread; when more TLP exists, this parallelism can compensate for a lack of

per-thread ILP.

In this work, we examine two alternative on-chip parallel architectures for the next generation of processors. We compare SMT and small-scale, on-chip multiprocessors (MP) in their ability to exploit both ILP and TLP. First, we identify the hardware bottlenecks that prevent multiprocessors from effectively exploiting ILP. Then, we show that because of its dynamic resource sharing, SMT avoids these inefficiencies and benefits from being able to run more threads on a single processor. The use of TLP is especially advantageous when per-thread ILP is limited. The ease of adding additional thread contexts on an SMT (relative to adding additional processors on an MP) allows simultaneous multithreading to expose more parallelism, further increasing functional unit utilization and attaining a 52% average speedup (versus a four-processor, single-chip multiprocessor with comparable execution resources).

This study also addresses an often-cited concern regarding the use of thread-level parallelism or multithreading: interference in the memory system and branch prediction hardware. We find that multiple threads cause inter-thread interference in the caches and also place greater demands on the memory system, increasing average memory latencies. By exploiting thread-level parallelism, however, SMT hides these additional latencies, so that they only have a small impact on total program performance. We also find that for parallel applications, the additional threads have minimal effects on branch prediction.

1 Introduction

To achieve high performance, contemporary computer systems rely on two forms of parallelism: instruction-level parallelism (ILP) and thread-level parallelism (TLP). Although they correspond to different granularities of parallelism, ILP and TLP are fundamentally identical; they both identify independent instructions that can execute in parallel and therefore can utilize parallel hardware. Wide-issue superscalar processors exploit ILP by executing multiple instructions from a single program in a single cycle. Multiprocessors exploit TLP by executing different threads in parallel on different processors. Unfortunately, neither parallel-processing style is capable of adapting to dynamically-changing levels of ILP and TLP, because the hardware enforces the distinction between the two types of parallelism. A multiprocessor must statically partition its resources among the multiple CPUs (see Figure 1); if insufficient TLP is available, some of the processors will be idle. A superscalar executes only a single thread; if insufficient ILP exists, much of that processor's multiple-issue hardware will be wasted.

Simultaneous multithreading (SMT) [31][30][12][14] allows multiple threads to compete for and share available processor resources every cycle. One of its key advantages when executing parallel applications is its ability to use thread-level parallelism and instruction-level parallelism interchangeably. By allowing multiple threads to share the processor's functional units simultaneously, thread-level parallelism is essentially converted into instruction-level parallelism. An SMT processor can therefore accommodate variations in ILP and TLP. When a program has only a single thread, i.e., it lacks TLP, all of the SMT processor's resources can be dedicated to that thread; when more TLP exists, this parallelism can compensate for a lack of per-thread ILP. An SMT processor can uniquely exploit whichever type of parallelism is available, thereby utilizing the functional units more effectively to achieve the goals of greater throughput and significant program speedups.

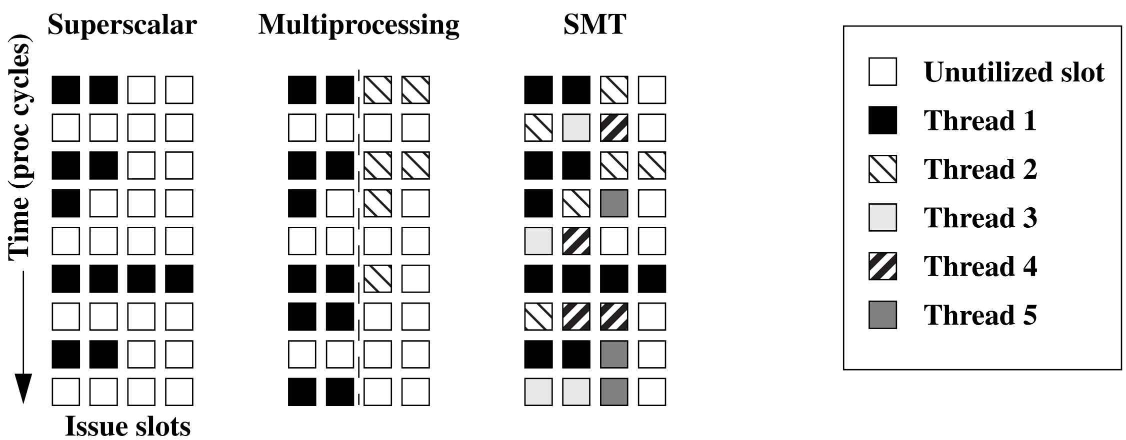

Figure 1: A comparison of issue slot (functional unit) utilization in various architectures. Each square corresponds to an issue slot, with white squares signifying unutilized slots. Hardware utilization suffers when a program exhibits insufficient parallelism or when available parallelism is not used effectively. A superscalar processor achieves low utilization because of low ILP in its single thread. Multiprocessors physically partition hardware to exploit TLP, and therefore performance suffers when TLP is low (for example, in sequential portions of parallel programs). In contrast, simultaneous multithreading avoids resource partitioning. Because it allows multiple threads to compete for all resources in the same cycle, SMT can cope with varying levels of ILP and TLP; consequently, utilization is higher and performance is better.

This paper explores parallel processing on a simultaneous multithreading architecture. Our investigation of parallel program performance on an SMT is motivated by the results of our previous work [31][30]. In [31], we used a multiprogrammed workload to assess the potential of SMT on a high-level architectural model and favorably compared total instruction throughput on an SMT to several alternatives: a superscalar processor, a fine-grained multithreaded processor, and a single-chip, shared-memory multiprocessor. In [30], we presented a micro-architectural design that demonstrated that this potential can be realized in an implementable SMT processor. The micro-architecture requires few small extensions to modern out-of-order superscalars; yet, these modest changes enable substantial performance improvements over wide-issue superscalars. In those two studies, the multiprogrammed workload provided plenty of TLP, because each thread corresponded to an entirely different application. In parallel programs, however, threads execute code from the same application, synchronize, and share data and instructions. These programs place different demands on an SMT than a multiprogrammed workload; for example, because parallel threads often execute the same code at the same time (the Single Program Multiple Data model), they may exacerbate resource bottlenecks.

In this work, we use parallel applications to explore the utilization of execution resources in the future, when greatly-increased chip densities will permit several alternative on-chip, parallel architectures. In particular, we compare SMT and small-scale on-chip multiprocessors (MP) in their ability to exploit both ILP and TLP. This study makes several contributions in this respect. First, we identify the hardware bottlenecks that prevent multiprocessors from effectively exploiting ILP in parallel applications. Then, we show that SMT (1) avoids these inefficiencies, because its resources are not statically partitioned, and (2) benefits from being able to run more threads on a single processor. This is especially advantageous when per-thread ILP is limited. The ease of designing in more thread contexts on an SMT (relative to adding more processors on an MP) allows simultaneous multithreading to expose more thread-level parallelism, further increasing functional unit utilization and attaining a 52% average speedup (versus a four-processor, single-chip multiprocessor with comparable execution resources).

Finally, we analyze how TLP stresses other hardware structures (such as the memory system and branch prediction hardware) on an SMT. First, we investigate the amount of inter-thread interference in the shared cache. Second, we assess the impact resulting from SMT's increased memory bandwidth requirements. We find that, although SMT increases the average memory latency, it is able to hide the increase by executing instructions from multiple threads. Consequently, inter-thread conflicts in the memory system have only a small impact on total program performance and do not inhibit significant program speedups. Third, we find that in parallel applications, the additional threads only minimally degrade branch and jump prediction accuracy.

The remainder of this paper is organized as follows. Section 2 describes the out-of-order superscalar processor that serves as the base for the SMT and MP architectures. Section 2 also discusses the extensions that are needed to build the SMT and MPs. Section 3 discusses the methodology used for our experimentation. In Section 4, we examine the shortcomings of small-scale multiprocessors and also demonstrate how SMT addresses these flaws. Section 5 presents an analysis of SMT's effect on the memory system and branch prediction hardware. Section 6 discusses some implications that SMT has for architects, compiler writers, and operating systems developers, and suggests areas of further research. In Section 7, we discuss related work, including a comparison with our previous results. Finally, we conclude in Section 8.

2 Simultaneous Multithreading and Multiprocessors

Both the SMT processor and the on-chip shared-memory MPs we examine are built from a common out-of-order superscalar base processor. The multiprocessor combines several of these superscalar CPUs in a small-scale MP, while simultaneous multithreading uses a wider-issue superscalar, and then adds support for multiple contexts. In the rest of this section, we will describe all three of these processor architectures.

2.1 Base processor architecture

The base processor is a sophisticated, out-of-order superscalar processor with a dynamic scheduling core similar to the MIPS R10000 [41]. Figure 2 illustrates the organization of this processor, while Figure 3 (top) shows its processor pipeline. On each cycle, the processor fetches a block of instructions from the instruction cache. After decoding these instructions, the register renaming logic maps the logical registers to a pool of physical renaming registers to remove false dependences. Instructions are then fed to either the integer or floating point instruction queues. When their operands become available, instructions are issued from these queues to the corresponding functional units. Instructions are retired in order.

2.2 SMT architecture

Our SMT architecture, which can simultaneously execute threads from up to eight hardware contexts, is a straightforward extension of the base processor. To support simultaneous multithreading, the base processor architecture requires

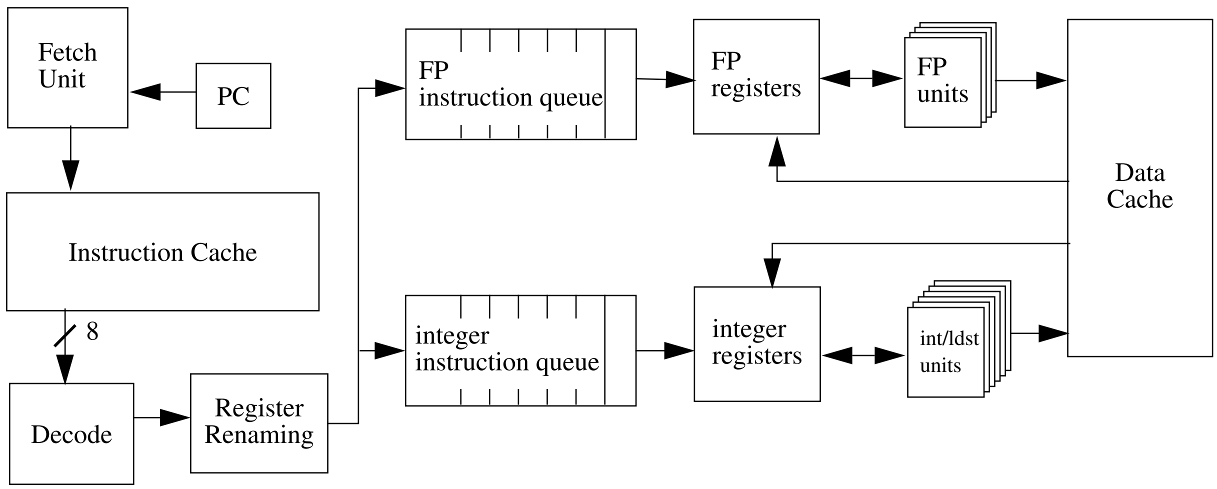

Figure 2: Organization of the dynamically-scheduled superscalar processor used in this study.

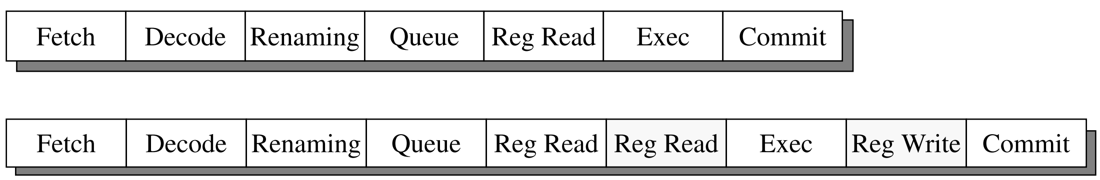

Figure 3: Comparison of the pipelines for a conventional superscalar processor (top) and the SMT processor's modified pipeline.

significant changes in only two primary areas: the instruction fetch mechanism and the register file.

A conventional system of branch prediction hardware (branch target buffer and pattern history table) drives instruction fetching, although we now have eight program counters and eight subroutine return stacks (one per context). On each cycle, the fetch mechanism selects up to two threads (among threads not already incurring I-cache misses) and fetches up to four instructions from each thread (the 2.4 scheme from [30]). The total fetch bandwidth of 8 instructions is therefore equivalent to that required for an 8-wide superscalar processor, and only two I-cache ports are required. Additional logic, however, is necessary in the SMT to prioritize thread selection. Thread priorities are assigned using the icount feedback technique [30], which favors threads that are using processor resources most effectively. Under icount, highest priority is given to the threads that have the least number of instructions in the decode, renaming, and queue pipeline stages. This approach prevents a single thread from clogging the instruction queue, avoids thread starvation, and provides a more even distribution of instructions from all threads, thereby heightening inter-thread parallelism. The peak throughput of our machine is limited by the fetch and decode bandwidth of 8 instructions per cycle.

Following instruction fetch and decode, register renaming is performed, as in the base processor. Each thread can address 32 architectural integer (and FP) registers. The register renaming mechanism maps these architectural registers (one set per thread) onto the machine's physical registers. An eight-context SMT will require at least 8*32 = 256 physical registers, plus additional physical registers for register renaming. With a larger register file, longer access times will be required, so the SMT processor pipeline is extended by two cycles to avoid an increase in the cycle time. Figure 3 compares the pipeline for the SMT versus that of the base superscalar. On the SMT, register reads take two pipe stages and are pipelined. Writes to the register file behave in a similar manner, also using an extra pipeline stage. In practice, we found that the lengthened pipeline degraded performance by less than 2% when running a single thread. The additional pipe stage requires an extra level of bypass logic, but the number of stages has a smaller impact on the complexity and delay of this logic ($O(n)$) than the issue width ($O(n^2)$) [21]. Our previous study [30] contains more details regarding the effects of the two-stage register read/write pipelines on the architecture and performance.

In addition to the new fetch mechanism and the larger register file and longer pipeline, only three processor resources are replicated or added to support SMT: per-thread instruction retirement and trap mechanisms, and an additional thread id field in each branch target buffer entry. No additional hardware is required to perform multithreaded scheduling of instructions to functional units. The register renaming phase removes any apparent inter-thread register dependences, so that the conventional instruction queues can be used to dynamically schedule instructions from multiple threads. Instructions from all threads are dumped into the instruction queues, and an instruction from any thread can be issued from the queues once its operands become available.

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

2.3 Single-chip multiprocessor hardware configurations

In our analysis of SMT and multiprocessing, we focus on a particular region of the MP design space, specifically, small-scale, single-chip shared-memory multiprocessors. As chip densities increase, single-chip multiprocessing will be possible, and some architects have already begun to investigate this use of chip real estate [20]. An SMT processor and a small-scale, on-chip multiprocessor have many similarities: e.g., both have large numbers of registers and functional units, on-chip caches, and the ability to issue multiple instructions each cycle. In this study, we keep these resources approximately similar for the SMT and MP comparisons, and in some cases, we give a hardware advantage to the MP.[1]

We look at both two- and four-processor multiprocessors, partitioning the scheduling unit resources of the multiprocessor CPUs (the functional units, instruction queues, and renaming registers) differently for each case. In the two-processor MP (MP2), each processor receives half of the on-chip execution resources described above, so that the total resources relative to an SMT are comparable (Table 1). For a four-processor MP (MP4), each processor contains approximately one-fourth of the chip resources. The issue width for each processor in these two MP models is indicated by the total number of functional units. Note that even within the MP design space, these two alternatives (MP2 and MP4) represent an interesting tradeoff between TLP and ILP. The two-processor machine can exploit more ILP, because each processor has more functional units than its MP4 counterpart, while MP4 has additional processors to take advantage of more TLP.

Table 1 also includes several multiprocessor configurations in which we increase hardware resources. These configurations are designed to reduce bottlenecks in resource usage in order to improve aggregate performance. MP2fu (MP4fu), MP2q (MP4q), and MP2r (MP4r) address bottlenecks of functional units, instruction queues, and renaming registers, respectively. MP2a increases all three of these resources, so that the total execution resources of each processor are equivalent to a single SMT processor.[2] (MP4a is similarly augmented in all three resource classes, so that the entire MP4a multiprocessor also has twice as many resources as our SMT.)

For all MP configurations, the base processor uses the out-of-order scheduling processor described earlier, and the base pipeline from Figure 3 (top). Each MP processor only supports one context; therefore, its register file will be smaller and access will be faster than the SMT. Hence, the shorter pipeline is more appropriate.

2.4 Synchronization mechanisms and memory hierarchy

SMT has three key advantages over multiprocessing: (1) flexible use of ILP and TLP, (2) the potential for fast synchro-

1. Note that simultaneous multithreading does not preclude multi-chip multiprocessing, in which an SMT processor could be the individual processors of the multiprocessor.
2. Each thread can access integer and FP register files, each of which has 32 logical registers plus a pool of renaming registers. When limiting SMT to only two threads, our SMT processor requires 32*2+100 = 164 registers, which is the same number as in the MP2 base configuration. When we use all 8 contexts on our SMT processor, each register file has a total of 32*8 logical registers plus 100 renaming registers (356 total). We equalize the total number of registers in the MP2r and MP2a configurations with the SMT by giving each MP processor 146 physical renaming registers in addition to the 32 logical registers, so that the total in the MP system is 2*(32+146) = 356.

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

Table 1: Processor configurations. For the enhanced MP2 and MP4 configurations, the resources that have been increased from the base configurations are listed in boldface. In our processor, not all integer functional units can handle load and store instructions. The integer (load/store) column indicates the total number of integer units and the number of these units that can be used for memory references.

nization, and (3) a shared L1 cache. This study focuses on SMT's ability to exploit the first advantage by determining the costs of partitioning execution resources; we therefore allow the multiprocessor to use SMT's synchronization mechanisms and cache hierarchy to avoid tainting our results with effects of the latter two.

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

The entire cache hierarchy, including the L1 caches, is shared by all threads in an SMT processor. Multiprocessors typically do not use a shared L1 cache to exploit data sharing between parallel threads. Each processor in an MP usually has its own private cache (as in the commercial multiprocessors described in [29][27][5][26]), and therefore incurs some coherence overhead when data sharing occurs. Because we allow the MP to use the SMT's shared L1 cache, this coherence overhead is eliminated. Although multiple threads may have working sets that interfere in a shared cache, we will show in Section 5 that inter-thread interference is not a problem.

2.5 Comparing SMT and MP

In comparing the total hardware dedicated to our multiprocessor and SMT configurations, we have not taken into account chip area required for buses or the cycle-time effects of a wider-issue machine. In our study, the intent is not to claim that SMT has an absolute "x percent" performance advantage over MP, but instead to demonstrate that SMT can overcome some fundamental limitations of multiprocessors, namely, their inability to exploit changing levels of ILP and TLP. We believe that in the target design space we are studying, the intrinsic flaws resulting from resource partitioning in MPs will limit their effectiveness relative to SMT, even taking into consideration cycle time. (We will discuss this further in section 4.6).

3 Methodology

3.1 Why parallel applications?

SMT is most effective when threads have complementary hardware resource requirements. Multiprogrammed workloads and workloads consisting of parallel applications both provide TLP via independent streams of control, but they compete for hardware resources differently. Because a multiprogrammed workload (used in our previous work [31][30]) does not share memory references across threads, it places more stress on the caches. Furthermore, its threads have different instruction execution patterns, causing interference in branch prediction hardware. On the other hand, multiprogrammed workloads are less likely to compete for identical functional units.

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

Second, parallel applications illustrate the promise of SMT as an architecture for improving the performance of single applications. By using threads to parallelize programs, SMT can improve processor utilization, but more importantly, it can achieve program speedups. Finally, parallel applications are a natural workload for traditional parallel architectures, and therefore serve as a fair basis for comparing SMT and multiprocessors. For the sake of comparison, in Section 7, we will also briefly compare our parallel results with the multiprogrammed results from [30].

3.2 Workload

Our workload of parallel programs includes both explicitly- and implicitly-parallel applications (Table 2). Many multi-

Program	Source	Language	Parallelism	Description
FFT	SPLASH-2	C	explicit	Complex, 1-dimensional fast fourier transform
hydro2d	SPEC 92	Fortran	implicit	Solves hydrodynamical Navier Stokes equations to solve galactical jets
linpackd	Argonne National Laboratories	Fortran	implicit	Linear systems solver
LU	SPLASH-2	C	explicit	LU factorization
radix	SPLASH-2	C	explicit	Integer radix sort
shallow	Applied Parallel Research HPF test suite	Fortran	implicit	Shallow water benchmark
tomcatv	SPEC 92	Fortran	implicit	Vectorized mesh generation program
water-nsquared	SPLASH-2	C	explicit	Molecular dynamics N-body problem, partitioned by molecule
water-spatial	SPLASH-2	C	explicit	Molecular dynamics N-body problem, using 3-d spatial data structure

Table 2: Benchmark suite

processor studies look only at the parallel portions of a program for measuring speedups; we look at the execution of the entire program, including the sequential portions, for two reasons. First, a parallel program typically has a section of sequential code. In order to obtain a complete performance picture, it should be included (Amdahl's Law). Second, performance on the sequential sections is an indication of an architecture's effectiveness on applications that cannot be efficiently parallelized. Processor architectures need not sacrifice single-thread performance for good parallel performance. Because of its resource partitioning, a multiprocessor typically cannot get good single-thread performance; SMT can, by taking better advantage of ILP, even in sequential sections.[3]

Five of our benchmarks are explicitly-parallel programs from the SPLASH-2 suite [38], which are built on the Argonne National Laboratories parallel macro library [3]. Tomcatv and hydro2d from SPEC92 [7], as well as shallow and linpack, are implicitly-parallel programs for which we use the SUIF compiler [37] to extract loop-level parallelism. SUIF generates transformed C output files that contain calls to a parallel runtime library to create threads and execute loop iterations in parallel. In each application, all threads share the same address space, but each thread has its own private data and stack, which are stored in a distinct location in the address space.

For all programs in our workload, we use the Multiflow trace scheduling compiler [17] to generate DEC Alpha object files. The compiler generates highly-optimized code using aggressive static scheduling, loop unrolling, and other ILP-exposing optimizations, so that single-thread performance is maximized. These object files are linked with our versions of the ANL and SUIF runtime libraries to create executables.

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

Table 3: Processor instruction latencies. These values are the minimum latencies from when the source operands are ready to when the result becomes ready for a dependent instruction.

3.3 Simulation framework

Our simulator measures the performance of these benchmarks on the multiprocessor and SMT configurations described in Section 2.3. For each application on each of the hardware configurations, we look at how our results vary when we change the number of threads used (1 or 2 threads for MP2; 1, 2, or 4 for MP4, and up to 8 for SMT). To distinguish between the MP configurations and the number of threads running on them, we use the designation MP*x*.T*y*, where *x* refers to the particular hardware configuration (as named in Table 1), and *y* is the total number of threads that are running on the multiprocessor. In our MP experiments, we limit each MP processor to a single thread. A user-level threads package could be used to execute more than 1 thread, but without hardware support for multithreading, context switch overhead would overwhelm the performance benefits of software multithreading.

For both the SMT and MP architectures, the simulator takes unmodified Alpha executables and uses emulation-based, instruction-level simulation to model the processor pipelines, the TLBs, and the entire memory hierarchy. The instruction latencies for the functional units are similar to those of the DEC Alpha 21164 [8] and are listed in Table 3. The memory hierarchy in our processor consists of three levels of cache, with sizes, latencies, and bandwidth characteristics as shown in Table 4. We model the cache behavior, as well as the contention at the L1 banks, L2 banks, L1-L2 bus, and L3 bank. For branch

3. The SPLASH benchmark data sets are smaller than those typically used in practice. With these data sets, the ratio of initialization time to computation time is larger than with real data sets; therefore results for these programs typically include only the parallel computation time, not initialization and clean-up. Most parallel architectures can only achieve performance improvements in the parallel computation phases, and therefore, speedups refer to improvements on just this portion.In this study, we would like to evaluate performance improvement of the entire application on various parallel architectures, so we use the entire program in our experiments. Note that in addition to these results, we also present SMT simulation results for the parallel computation phases only in Section 4.5.

prediction, we use a 256-entry, 4-way set associative branch target buffer and a 2K x 2-bit pattern history table [4]. These structures are shared by all running threads (even if less than 8 are executing), allowing more flexible and therefore higher utilization. Most importantly, these structures are fully available even if only a single thread is executing. Of course, the competition for the shared resources among the threads may increase the frequency of cache misses and branch mispredictions. We discuss and quantify these effects in Section 5.

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

Table 4: Memory hierarchy details

4 Exploiting parallelism

4.1 SMT and multiprocessing performance results

Performance on parallel applications is maximized when both instruction-level and thread-level parallelism can be effectively used by the architecture. In order to understand the benefits of both types of parallelism, we first compare the average speedup (Figure 4) and throughput (Table 5) for simultaneous multithreading and the two multiprocessors, MP2 and MP4, as more threads are used in the programs. With 1 and 2 threads, MP2 outperforms MP4 (by more than 40%), because its configuration of hardware resources allows each processor to exploit more ILP. MP4, on the other hand, has two additional

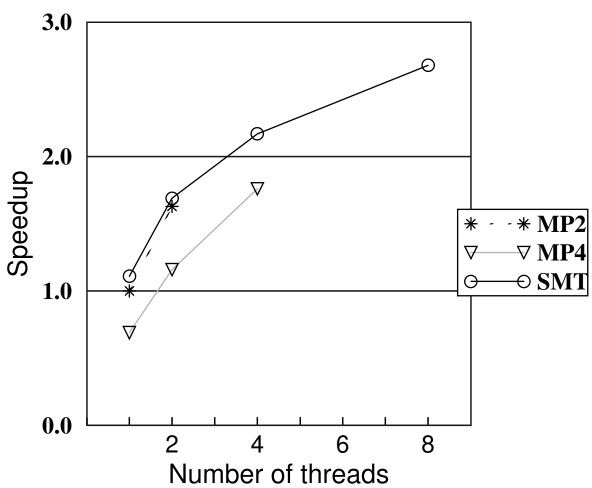

Figure 4: Speedups normalized to MP2.T1

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

Table 5: Throughput comparison of MP2, MP4 and SMT, measured in instructions per cycle.

We measure both speedup and instruction throughput for applications in various configurations. These two metrics are not identical, because as we change the number of threads, the dynamic instruction count varies slightly. The extra instructions are the parallelization overhead required by each thread to load necessary register state when executing a parallel loop. Speedup is therefore the most important metric, but throughput is often more useful for illustrating how the architecture is exploiting parallelism in the programs.

processors to take advantage of TLP. When each MP configuration is given its maximum number of threads (2 for MP2 and 4 for MP4), MP4.T4 has a 1.76 speedup, while MP2.T2 only reaches 1.63.

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

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

The following three subsections analyze the results from Table 5 in detail to understand why inefficiency exists in multiprocessors.

4.2 Measuring MP inefficiency

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

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

Issue limits: The number of functional units in each processor determines how many instructions of a particular type can be issued in a single cycle. We identify the number of cycles in which both (1) an instruction cannot be issued because there are insufficient functional units on a processor, and (2) an appropriate functional unit is free on another processor. We subcategorize this metric based on the particular functional unit type: integer issue, load/store issue, and FP issue.

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

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

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

4.3 Exploiting ILP

We examine in detail the performance of the multiprocessor scheduling unit (instruction issue, functional units, instruction queues, and renaming registers) by graphing the frequency of the metrics for MP2.T2 and MP4.T4 in Figures 5a and 5b, respectively. The high frequencies illustrate that (1) many of the programs have per-thread ILP that exceeds the amount that can be supported by a single MP2 or MP4 processor, and (2) there is a large degree of inefficiency for all programs but one.

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

For shallow and tomcatv, however, the FP renaming registers are also a large contributor to inefficiency. These two benchmarks have two of the highest average memory latencies (Section 5 will show this data in detail), which extends regis-

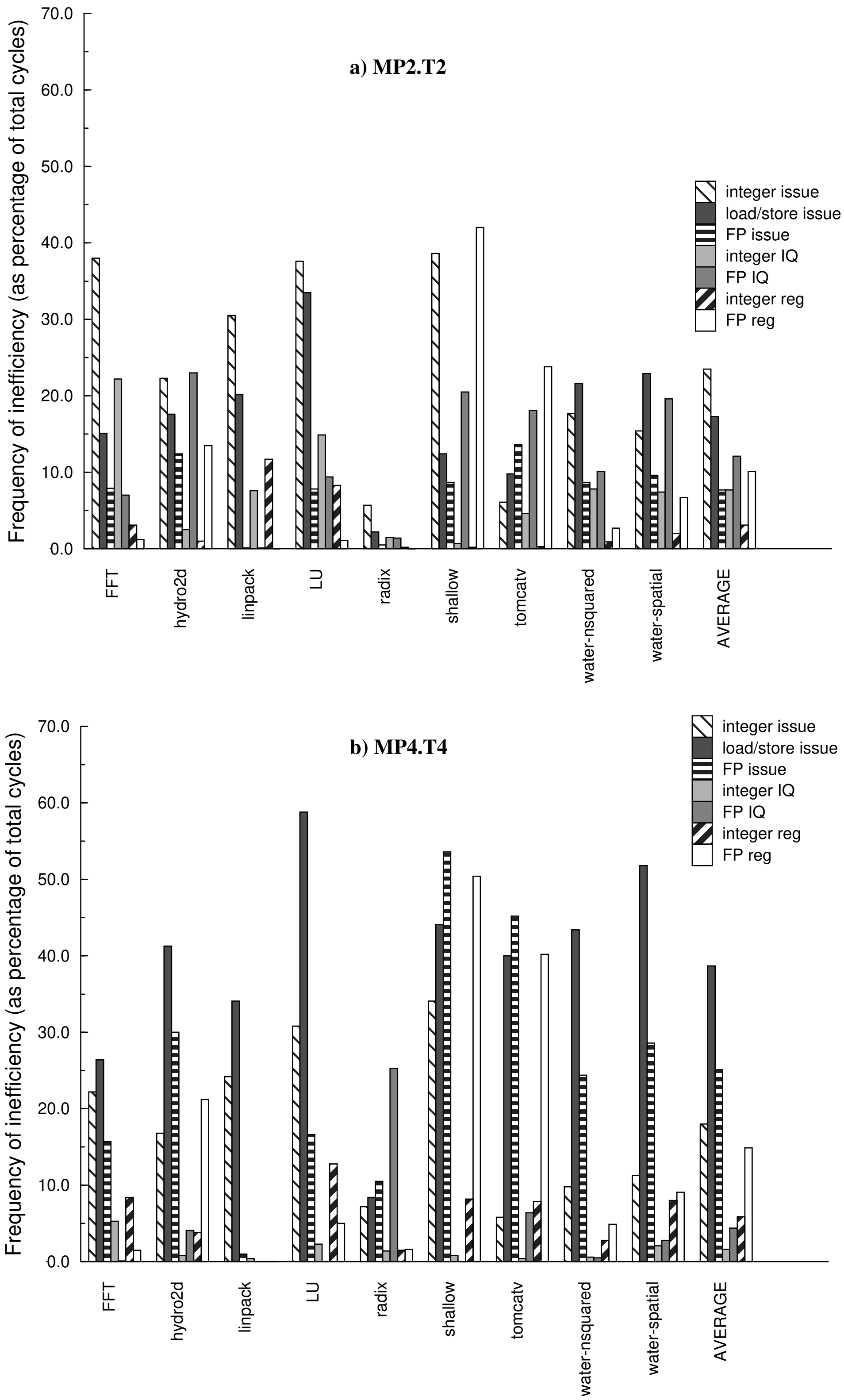

Figure 5: Frequencies of partitioning inefficiencies for MP2.T2 and MP4.T4. Each of the seven inefficiency metrics indicates the percentage of total execution cycles in which (1) one of the processors runs out of a resource, and (2) that same resource is available on the other processor. In a single cycle, more than one of the inefficiencies may occur, so the sum of all 7 metrics may be greater than 100%.

ter lifetimes. This heightened register pressure increases the likelihood that an individual processor will run out of FP renaming registers. In contrast, the integer renaming registers and the integer IQ are not a frequent source of waste in our programs, again, because the applications tend to be more floating-point intensive.

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

Addressing resource inefficiencies

In this section, we examine the performance of two sets of enhanced MP configurations, which are designed to address the resource inefficiencies indicated by Figures 5a and 5b. The first set of configurations serves two purposes. First, it determines if there is a single class of resources that is critical for effectively using ILP. Second, if there is a single resource bottleneck in an MP system, any reasonable MP implementation would be redesigned in a manner to compensate for that problem. As shown in Table 1, the MP2fu (MP4fu), MP2q (MP4q), and MP2r (MP4r) configurations have twice as many resources dedicated to functional units, queues, and renaming registers, respectively, relative to the base MP2 (MP4) configuration. Table 6 shows the average frequencies for each of the seven inefficiency metrics for these configurations. Each configuration targets a subset of the metrics and virtually eliminates the inefficiencies in that subset. Speedups, however, are small: Table 7 shows that these enhanced MP2 (MP4) configurations get speedups of 3% (4%) or less relative to MP2.T2 (MP4.T4). Performance is limited, because removing a subset of bottlenecks merely exposes other bottlenecks (Table 6).

Metric	Configuration									
	MP2.T2	**MP2fu.T2**	**MP2q.T2**	**MP2r.T2**	**MP2a.T2**	**MP4.T4**	**MP4fu.T4**	**MP4q.T4**	**MP4r.T4**	**MP4a.T4**
integer issue	23.5%	**0.4%**	27.9%	23.6%	**1.3%**	18.0%	**1.5%**	18.0%	21.6%	**2.5%**
load/store issue	17.3%	**1.1%**	18.0%	18.1%	**1.8%**	38.7%	**2.7%**	38.2%	42.2%	**3.8%**
FP issue	7.7%	**0.9%**	8.9%	7.9%	**1.4%**	25.1%	**2.3%**	24.5%	27.6%	**3.2%**
integer IQ	7.7%	5.7%	**2.2%**	9.5%	**2.8%**	1.6%	1.3%	**0.0%**	4.2%	**0.7%**
FP IQ	12.1%	11.3%	**1.1%**	15.6%	**4.1%**	4.4%	7.5%	**0.0%**	10.4%	**1.4%**
integer reg	3.1%	3.1%	5.4%	**0.0%**	**0.0%**	5.9%	7.1%	1.2%	**0.2%**	**1.3%**
FP reg	10.1%	10.3%	17.0%	**0.0%**	**0.0%**	14.9%	23.1%	6.5%	**3.9%**	**6.3%**
any inefficiency	58.9%	31.9%	55.9%	56.0%	12.0%	67.9%	26.5%	67.6%	71.4%	16.6%

Table 6: Average frequencies for inefficiency metrics. Each configuration focuses on a particular subset of the metrics for MP2.T2 and MP4.T4, printed in boldface. Notice that the boldface percentages are small, showing that each configuration successfully addresses the targeted inefficiencies. The inefficiency metrics can overlap, i.e., more than one bottleneck can occur in a single cycle; consequently, in the last row, we also show the percentage of cycles in which at least one of the inefficiencies occur.

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

Table 7: Speedups for MP2 and MP4 configurations relative to 1 thread MP2.

As Table 7 demonstrates, the three classes of resources have different impacts on program performance, with renaming registers having the greatest impact on performance. There are two reasons for this behavior. First, when all renaming registers are in use, the processor must stop fetching, which imposes a large performance penalty in an area that is already a primary bottleneck. In contrast, a lack of available functional units will typically only delay a particular instruction until the next cycle. Second, the lifetimes of renaming registers are very long, so a lack of registers may actually prevent fetching for several cycles. On the other hand, instruction queue entries and functional units can become available more quickly.

Although the renaming registers are the most critical resource, the variety of inefficiencies means that effective use of ILP (i.e., better performance) cannot be attained by simply addressing a single class of resources. In the second set of MP enhancements (MP2a and MP4a), we increase all execution resources to address the entire range of inefficiencies. Tables 6 and 7 show that MP2a significantly reduces all metrics to attain a 1.80 speedup. Although this speedup is greater than the 1.69 speedup we saw for SMT.T2 in Figure 4, there are two reasons why MP2a is not a cost-effective solution (compared to SMT) for increasing performance. First, each of the two processors in the MP2a configuration has the same total execution resources as our single SMT processor, but resource utilization in MP2a is still very poor. For example, the two processors now have a total of 20 functional units, but, on average, fail to issue more than 4 instructions per cycle. Second, Figure 4 shows that when allowed to use all of its resources (i.e., all 8 threads), a single SMT processor can attain a much larger speedup of 2.68. (Section 4.5 will discuss this further.) Resource partitioning prevents the multiprocessor configuration from improving performance in a more cost-effective manner that would be competitive with simultaneous multithreading.

4.4 Exploiting TLP

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

In Figure 6, we compare MP2 and MP4 speedups as we increase the number of threads. On average, MP4.T4 outperforms MP2.T2, but for individual programs, the relative performance of the two varies, depending on the amount of per-thread ILP in the benchmark. (We approximate the level of ILP by using the single-thread throughput data shown in Table 8.) In the programs with higher per-thread ILP (LU, linpack, FFT, and water-nsquared), MP2 has the performance edge over MP4, because there is sufficient ILP to compensate for having fewer processors. When programs lack ILP (the other five applications), however, MP4 uses the additional processors to exploit TLP, compensating for individual processors that have limited execution resources. The best examples are hydro2d and the ILP-starved radix, where MP4.T4 gets speedups of 1.95

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

Table 8: Benchmark throughput (instructions per cycle) for SMT.T1. The SMT.T1 configuration gives an indication of the amount of instruction-level parallelism in each program, using the decomposed application running one thread.

and 3.46, while MP2.T2 only gets speedups of 1.79 and 1.95, respectively.

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

4.5 Effectively using parallelism on an SMT processor

Rather than adding more execution resources (like the MP2a and MP4a configurations) to improve performance, SMT boosts performance and improves utilization of existing resources by using parallelism more effectively. Unlike multiprocessors which suffer from rigid partitioning, simultaneous multithreading permits dynamic resource sharing, so that resources can be flexibly partitioned on a per-cycle basis to match the ILP and TLP needs of the program. When a thread has a lot of

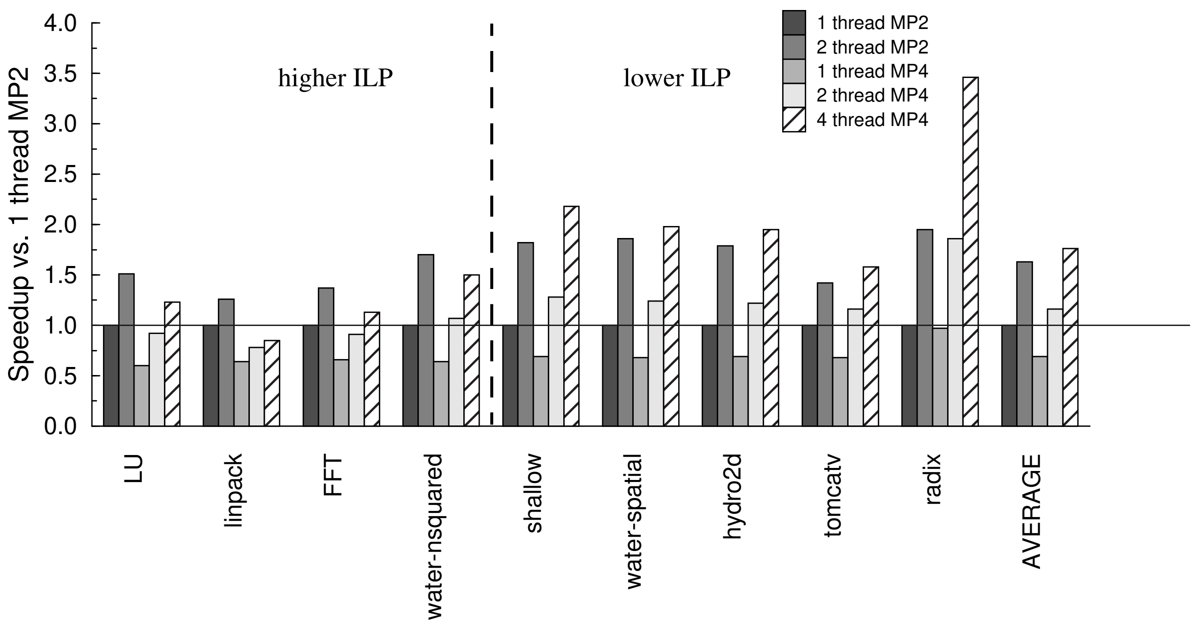

Figure 6: MP2 and MP4 speedups vs. one-thread MP2 baseline. Programs are listed in descending order based on the amount of ILP in the program. MP2.T2 outperforms MP4.T4 in the programs to the left of the dashed line. MP4.T4 has the edge for those on the right.

ILP, it can access all processor resources; and TLP can compensate for a lack of per-thread ILP. Table 9 shows that simultaneous multithreading exploits the two types of parallelism to improve speedups significantly. As more threads are used, speedups increase (up to 2.68 on average with 8 threads), exceeding the performance gains attained by the enhanced MP configurations.

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

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

Table 9: SMT speedups relative to 1 thread MP2. Benchmarks are listed in descending order, based on the amount of single-thread ILP.

For all programs, as we use more threads, the combined use of ILP and TLP increases processor utilization significantly. On average, throughput reaches 4.88 instructions per cycle, while several benchmarks attain an IPC of greater than 5.7 with 8 threads, as shown in Table 10. These results, however, fall short of peak processor throughput (the 8 instructions per cycle limit imposed by instruction fetch and decode bandwidth), in part because they encompass the execution of the entire program, including the sequential parts of the code. In the parallel computation portions of the code, throughput is closer to the maximum. Table 11 shows that with 8 threads in the computation phase of the SPLASH-2 benchmarks, average throughput reaches 5.91 instructions per cycle and peaks at 6.83 IPC for LU.

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

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

Table 10: SMT throughput in instructions per cycle.

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

Table 11: SMT throughput (measured in instructions per cycle) in the computation phase of SPLASH-2 benchmarks.

5 Effects of thread interference in shared structures

From the perspective of the execution resources, ILP and TLP are identical. From the perspective of the memory system and branching hardware, however, TLP introduces slightly different behavior. In this section, we examine three ways in which simultaneous multithreading impacts other resources when using thread-level parallelism. First, the working sets of multiple threads may interfere in the cache. Second, the increased processor throughput places greater demands on the memory system. Third, multithreading can introduce interference in the BTB and branch prediction tables. We quantify these effects by measuring the hit and miss rates on the shared memory and branch prediction structures. We also measure the impact they have on memory latency and total program performance.

Inter-thread cache interference

Threads on an SMT processor share the same cache hierarchy, so their working sets may introduce inter-thread conflict misses. Figure 7 categorizes L1 misses as first reference misses, inter-thread conflict misses, and intra-thread conflict misses. (We identify inter-thread misses as the misses that would have been hits, if each thread had its own private 32KB L1 D-cache.) As the number of threads increases, the number of inter-thread conflict misses also rises, from 1.4% (2 threads) to 4.8% (4 threads) to 5.3% (8 threads) of total memory references. The primary concern, however, is not the impact on the hit rate, but the impact on overall program performance.

In order to assess the effect of inter-thread misses on program performance, we simulated program behavior as if there was no inter-thread interference, treating inter-thread conflict misses as if they were hits. We found that with eight threads, performance was, on average, only 0.1% better than the SMT results. There are two reasons why inter-thread cache interference is not a significant problem for our workload. First, the additional inter-thread conflict misses in the direct-mapped L1 cache are almost entirely covered by the four-way set associative L2 cache, as shown in Figure 8. The fully-pipelined L2 cache has a relatively low latency (6 additional cycles) and high bandwidth, so average memory access time (AMAT) increases only slightly. Figure 9 graphs the various components of average memory access time (including components for both cache misses and bank or bus contention) for each benchmark. When increasing the number of threads from 1 to 8, the cache miss component of AMAT increases by less than 1.5 cycles on average, indicating the small effect of inter-thread con-

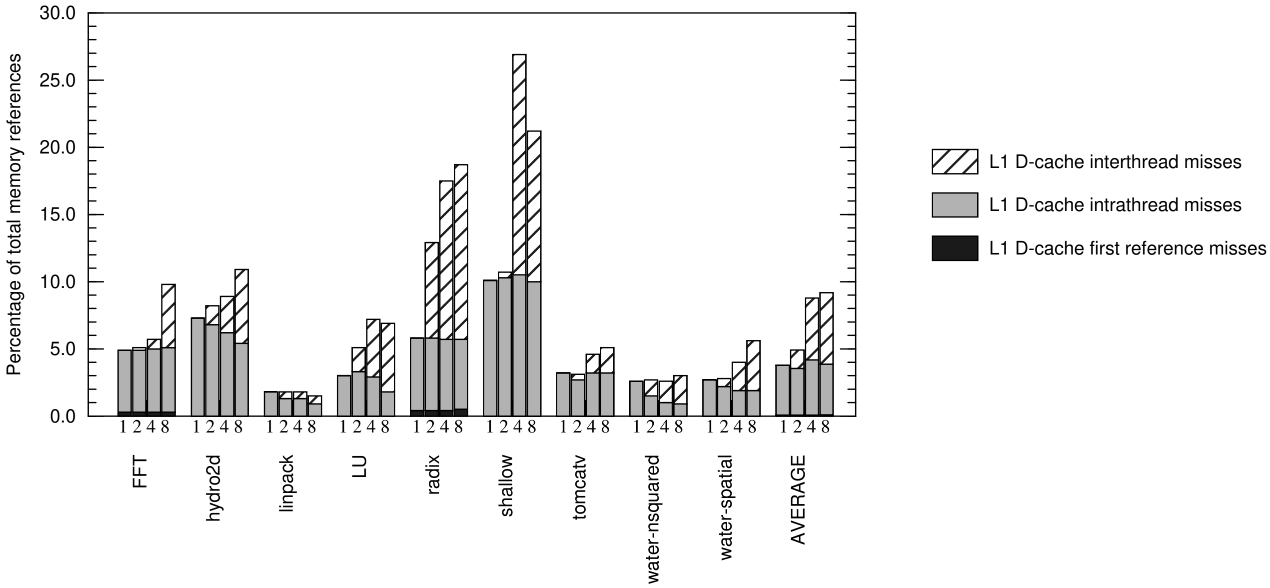

Figure 7: Categorization of L1 D-cache misses (shown for each benchmark with 1, 2, 4, and 8 threads).

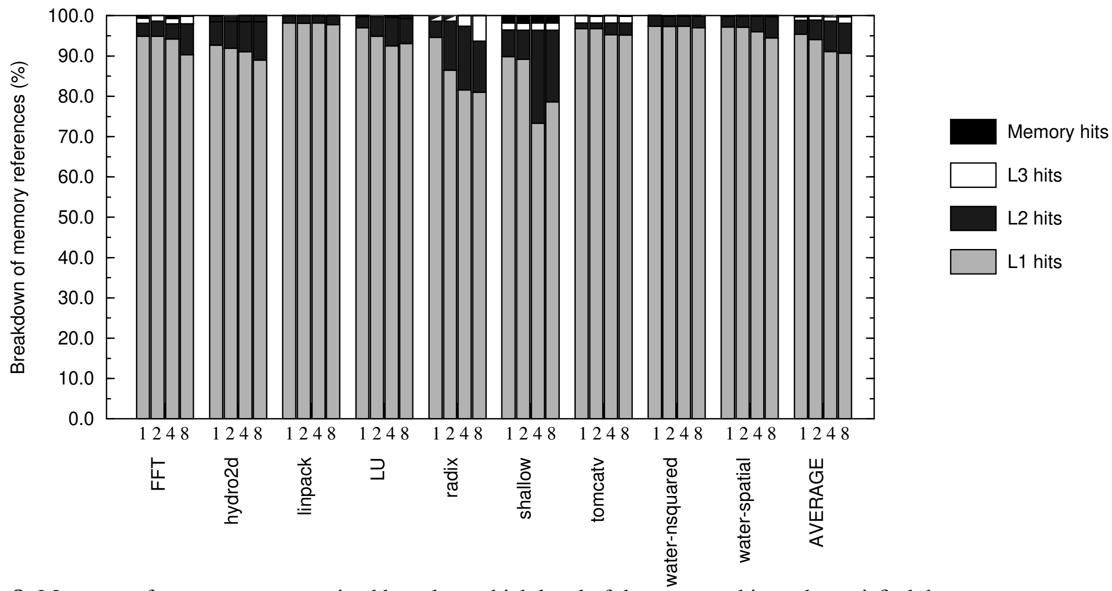

Figure 8: Memory references are categorized based on which level of the memory hierarchy satisfied the request (hit in the L1, L2, or L3 caches, or in main memory). Data is shown for 1, 2, 4, and 8 threads for each benchmark. Notice that as the number of threads increases, the combined percentage of memory references satisfied by either the L1 or L2 caches remains fairly constant. Most of our applications fit in the 8MB L3 cache, so very few references reach main memory.

flict misses. Second, out-of-of-order execution, write buffering, and the use of multiple threads allow SMT to hide the small increases in additional memory latency, and as shown earlier in Section 4, large speedups can be attained.

In addition, our study somewhat overstates the amount of inter-thread interference, because we have not applied compiler optimizations (such as cache tiling [10][22]) to minimize interference by reducing the size of the working sets. Thekkath and Eggers [24] found that for traditional multithreaded architectures, programmer- or compiler-based locality optimizations can significantly reduce inter-thread interference. We believe that the same should hold for simultaneous multithreading, and this is an area of further research.

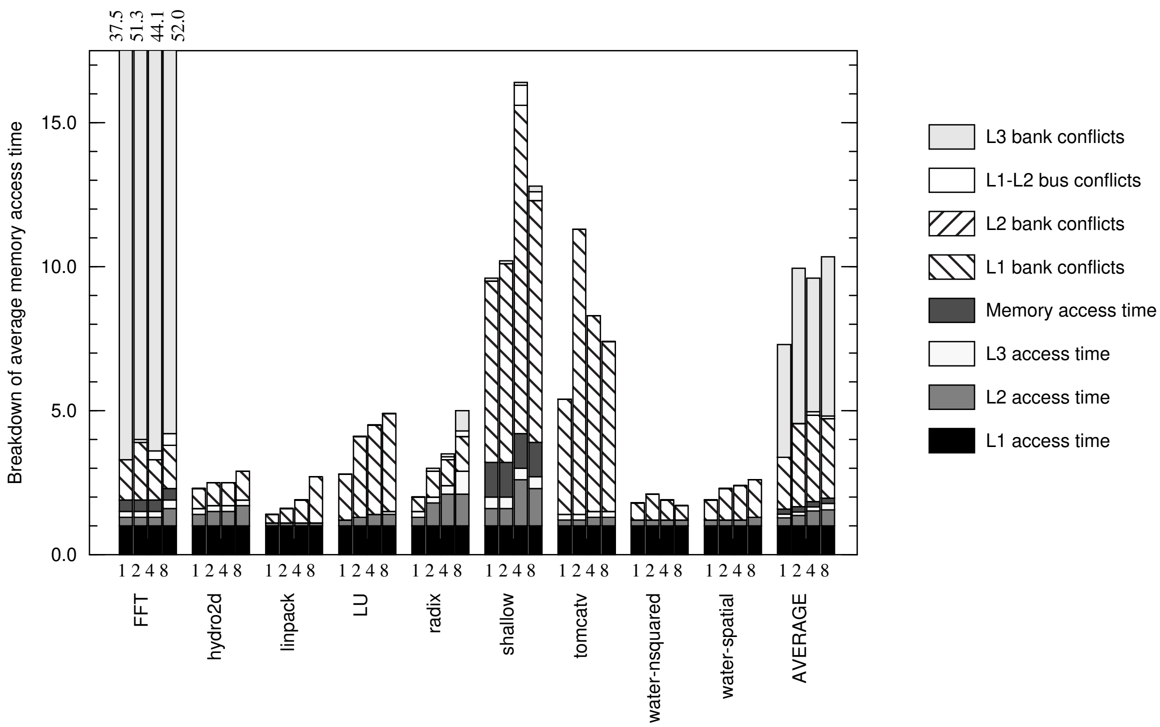

Figure 9: Components of average memory access time (shown for each benchmark with 1, 2, 4, and 8 threads). Each bar shows how cache misses and contention contribute to average memory access time. The lower four sections correspond to latencies due to cache misses, while the four upper sections represent additional latencies that result from conflicts in various parts of the memory system.

Increased memory requirements

We also investigated the impact of the increased demand placed on the memory system by SMT. As more threads are used in our applications, total program execution time drops and utilization rises, thus increasing the density of memory references per cycle. Table 12 shows that our memory requirements have doubled from 0.82 to 1.64 memory references per cycle, as we increase the number of threads from 1 to 8. Looking back to Figure 9, we can determine how our memory system handles the increased load. The most important area of concern is the bank conflicts in the L1 cache. (Note that the bus and the L2 cache are not bottlenecks in our memory system.) For most programs, the L1 bank conflicts account for the largest component of average memory access time. Many of these bank conflicts, especially in the one-thread case, are the result of the long 64-byte cache lines used in our architecture. For applications with unit-stride memory accesses, this block size increases the likelihood that bank conflicts will occur. The demands on a particular bank may also increase as we add more threads, because the threads may access data that also maps to the same cache line. We found, however, that when using longer cache lines, the gains due to better spatial locality outweighed the costs associated with the increase in L1 bank contention. Furthermore, some of the contention may be reduced by using recently-proposed architectural techniques to use cache ports more efficiently [36].)

In order to measure the performance degradation due to inter-thread bank conflicts, we modeled an L1 cache that ignored inter-thread bank conflicts, while still accounting for same-thread bank conflicts. These simulation results showed a 3.4% speedup over the baseline SMT results, demonstrating that inter-thread bank conflicts have only a small effect on performance.

The L3 conflicts are a smaller concern, although Figure 9 illustrates that they have a noticeable impact on FFT. The L3 cache is slower than the other cache levels, and can only be accessed once every four cycles. This lack of pipelining means

Configuration	FFT	hydro2d	linpack	LU	radix	shallow	tomcatv	water-nsquared	water-spatial	Average
SMT.T1	0.76	0.75	1.06	1.47	0.07	0.70	0.87	0.91	0.77	0.82
SMT.T2	1.01	1.19	1.36	2.05	0.13	1.12	1.08	1.45	1.34	1.19
SMT.T4	1.10	1.55	1.46	2.34	0.25	1.20	1.58	1.81	1.79	1.45
SMT.T8	1.11	1.77	1.46	2.45	0.45	1.65	1.94	1.97	1.93	1.64

Table 12: Average number of memory references per cycle.

that a burst of references to the L3 cache can result in queueing delays. For FFT, the L3 queueing delays result from a large burst of memory writes; however, these latencies are hidden by the write buffer, so that the impact on overall performance is small. Note that even without simultaneous multithreading (i.e., only 1 thread), L3 bank conflicts are still an issue.

Interference in branch prediction hardware

As with the memory system, branch prediction hardware can experience inter-thread interference when multiple threads are executed simultaneously. Table 13 compares the branch prediction and jump prediction accuracy for increasing numbers of threads. In parallel applications, the distinct threads tend to exhibit very similar branching behavior. Consequently, minimal interference occurs in the branch prediction structures.

Number of threads	Branch misprediction	Jump misprediction
1	2.0%	0.0%
2	2.5%	0.1%
4	2.8%	0.1%
8	2.5%	0.0%

Table 13: Branch and jump misprediction rates.

In this study, we found that the use of multiple, simultaneous threads can introduce a small degree of inter-thread cache and branch prediction interference and can also burden the memory subsystem with a greater density of memory requests. Fortunately, these effects only have a modest effect on average memory access time, and simultaneous multithreading effectively hides the additional latencies. On an SMT processor, the benefits of using additional thread-level parallelism far outweigh its costs, so that with 4 and 8 threads, significant speedups are realized.

6 Placing these results in context

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

Implications for architects

As chip densities continue to increase, architects must decide what is the best use of chip area. In evaluating the merits of SMT and MPs as two potential design alternatives, architects should also recognize two additional factors. First, potential cycle time impacts should be taken into consideration when interpreting the results of this study, which uses total execution cycles to compare SMT and MPs. Our SMT architecture (as described in Section 2) is designed in a manner that limits the cycle time impact of SMT vs. a modern out-of-order execution superscalar of similar width, i.e., an extra cycle is taken to access the larger register file. However, the MP2 or MP4 processors could still have a faster cycle time than the 8-wide SMT. Without specific implementations of each of the processors, it is difficult to make detailed quantitative comparisons of the cycle times for 2, 4, and 8-wide issue machines. Our experiments, however, show that when both SMT and MPs are at full capacity with a parallel workload (i.e., maximum number of threads -- 8 for SMT, 2 for MP2, and 4 for MP4), SMT outperforms MP2 (MP4) by more than 62% (54%); in order for MPs to overcome SMT's performance advantage, the improvement in MP cycle times would have to be comparable. Although other comparisons can also be made, e.g., SMT.T4 vs. MP.T4, they are less indicative of SMT’s true potential, because they restrict SMT's advantage -- its ability to take advantage of extra thread-level parallelism.

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

Implications for compiler writers

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

Simultaneous multithreading offers compilers and programmers more flexibility to extract whatever form of parallelism exists, because the underlying architecture can use ILP and TLP equally effectively. Aggressive superscalar or VLIW compilers already use many optimizations to extract large amounts of ILP. An SMT processor gives these compilers yet another mechanism for exploiting parallelism that can be used to fill wide-issue processors. For example, SMT offers an opportunity to expose new parallelization techniques, because of the potential for faster synchronization. In comparison to multiprocessors, even those with shared L2 caches, SMT provides an opportunity for cheaper synchronization and communication, either through a shared L1 cache or through functional units or registers. For example, SMT could enable the parallelization of loops with tight doacross synchronization because of the reduction in synchronization overhead. Applications with high communication-to-computation ratios, like ear from the SPEC92 suite, can also benefit from cheap synchronization via the shared L1 cache.[19]

Implications for operating systems developers

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

7 Related work

Simultaneous multithreading has been the subject of several recent studies. We first compare this study with our previous work and then discuss studies done by other researchers. As indicated in the previous section, SMT may exhibit different behavior for parallel applications when compared to a multiprogrammed workload because of different degrees of inter-thread interference. Table 14 compares the SMT throughput results from this study (parallel workload) with the multiprogrammed results in [30]. Note that because the applications themselves are different, and the simulation methodology is slightly different (mainly a 2MB L3 cache vs. an 8MB L3), we can only make rough comparisons and highlight some differences.

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

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

Table 14: Throughput for parallel and multiprogrammed workloads. This table compares the throughput (in instructions per cycle) obtained for the multiprogrammed and parallel workloads, as the number of threads is increased. The last column compares the throughput achieved in the parallel computation phases of the parallel applications.

Second, when all threads are being used, the multiprogrammed workload sustains greater throughput than the parallel applications. However, when using only the parallel computation sections of the SPLASH-2 applications (last column reproduced from Table 11), parallel throughput is greater than the multiprogrammed results, for all numbers of threads.

Parallel applications experience less inter-thread interference in the branch hardware, because the distinct threads tend to exhibit similar branching behavior. Table 15 compares the parallel applications of this study with the multiprogrammed workload in our previous study [30], in terms of branch and jump misprediction rates. The data indicates that minimal branch prediction interference occurs in parallel applications, relative to a multiprogrammed workload.

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

Table 15: Branch and jump misprediction rates. The table compares branch and jump misprediction rates for parallel and multiprogrammed workloads, as the number of threads is increased.

Metric		parallel			multiprogrammed		
		1	4	8	1	4	8
L1 I-cache	miss rate	0.0	0.0	0.0	2.5	7.8	14.1
	misses per 1000 completed instructions (MPCI)	0.0	0.0	0.0	6	17	29
L1 D-cache	miss rate	4.6	8.9	9.3	3.1	6.5	11.3
	(MPCI)	14	26	27	12	25	43
L2 cache	(MPCI)	4	4	5	3	5	9
L3 cache	(MPCI)	1	1	1	1	3	4

Table 16: Comparison of memory system interference for parallel and multiprogrammed workloads.

Parallel applications also suffer from less inter-thread interference in the memory system, compared to multiprogrammed applications. For the sake of comparison, Table 16 presents resource contention statistics from [30]. Although contention occurs in the parallel applications, it is much less significant than in multiprogrammed workloads. This means that the use of multithreading to expose parallelism is truly useful, as the latency tolerance has the ability to hide additional latencies introduced by cache and branch interference. These results demonstrate that SMT is particularly well-suited for parallel applications, especially in comparison to multiprocessors.

In Tullsen, et al., [31], we compared the performance of SMT and various multiprocessor configurations and found that SMT outperforms an MP with comparable hardware for a multiprogrammed workload. In contrast to that study, this paper compares the architectures using a parallel workload and an implementable (rather than idealized) SMT architecture, and also identifies the sources of resource waste in multiprocessor configurations. Furthermore, this study also uses speedup, rather than processor utilization, as the primary performance metric. In the parallel applications used in this study, all threads work together toward completing the entire program, so speedup becomes the critical metric.

Other studies

Several variants of simultaneous multithreading have been studied. Gulati and Bagherzadeh [12] implemented simultaneous multithreading as an extension to a superscalar processor and measured speedups for a set of parallelized programs. 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.

Hirata, et al., [14] proposed an architecture that dynamically packs instructions from different streams. They evaluated the performance benefits of their architecture by parallelizing a ray-tracing application. Their simulations do not include caches or TLBs. Prasadh and Wu [23], as well as Keckler and Dally [15], have proposed architectures in which VLIW operations from multiple threads are dynamically interleaved onto a processor.

The architectures described by Govindarajan, et al. [11], Gunther [13], and Beckmann and Polychronopoulos [2] partition issue bandwidth among threads, and only one instruction can be issued from each thread per cycle. These architectures lack flexible resource sharing, which contributes to resource waste when only a single thread is running.

Studies by Daddis and Torng [6], Prasadh and Wu [23], and Yamamoto, et al. [40][39], as well as our previous work [31][30], also examined simultaneous multithreading architectures, but looked at multiprogrammed workloads, rather than parallel applications. The simultaneous multithreading in the study by Li and Chu [16] was based on an analytic model of performance.

Several other architectures have been designed to exploit multiple levels of program parallelism. The M-Machine [9], MISC [33][32], and Multiscalar [28] architectures all require significant compiler support (or hand-coding) to extract maximum performance. In all three designs, execution resources are partitioned in a manner that prevents them from being dynamically shared by all threads. The S-TAM architecture [34] exploits both instruction-level and thread-level parallelism by statically allocating threads to processors, and dynamically allocating each thread to a functional unit. A primary goal was to expose scheduling and allocation to the compiler. On their machine intrathread parallelism was avoided; parallelism was instead expressed only between threads. Olukotun, et al., [20] advocates the design of single-chip multiprocessors, instead of wider-issue superscalars, to exploit thread-level parallelism.

Cache effects of multithreading has been the subject of several studies. Yamamoto, et al., and Gulati and Bagherzadeh both found that the cache miss rates in simultaneous multithreading processors increased when more threads were used. Neither quantified the direct effect from inter-thread interference, however.

Thekkath and Eggers [24] examined the effectiveness of multiple contexts on conventional, coarse-grained multithreaded architectures. They found that cache interference between threads varied depending on the benchmark. For locality-optimized programs, the total number of misses remained fairly constant as the number of contexts was increased. For unoptimized programs, however, misses were more significant and resulted in performance degradations. Weber and Gupta [35] also studied the effects of cache interference in conventional multithreaded architectures, and found increases in inter-thread misses that are comparable to our results. Agarwal [1] and Saavedra-Barrera, et al., [25] used analytic models for studying the efficiency of multithreaded processors. Both models included factors for cache interference which they correlated with the results obtained by Weber and Gupta. None of the studies included the effects of bank contention in their results, since they did not use multiple banks.

Nayfeh and Olukotun [18] investigated the benefits of a shared cluster cache on a single-chip multiprocessor. They found that inter-thread interference can cause degradation for multiprogrammed workloads. For parallel applications, a shared cache could actually obtain superlinear speedups in some cases because of prefetching effects. They modeled a banked data cache, but did not discuss the effects of contention in their results.

8 Conclusions

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

Second, we illustrate that, in contrast, simultaneous multithreading allows compilers and programmers to focus on extracting whatever parallelism exists, by treating instruction-level and thread-level parallelism equally. ILP and TLP are fundamentally identical; they both represent independent instructions that can be used to increase processor utilization and improve performance. SMT has the flexibility to use both forms of parallelism interchangeably, because threads can share resources dynamically. Rather than adding more resources to further improve performance, existing resources are used more effectively. By using more hardware contexts, SMT can take advantage of TLP to expose more parallelism, and attain an average throughput of 4.88 instructions per cycle, while increasing its performance edge over MP2 and MP4 to 64% and 52%, respectively.

Third, our results demonstrate that SMT can achieve large program speedups on parallel applications. Even though these parallel threads have greater potential for interference because of similar resource usage patterns (including memory references and demands for renaming registers and functional units), simultaneous multithreading has the ability to compensate for these potential conflicts. We found that inter-thread cache interference, bank contention, and branch prediction interference on an SMT processor had only minimal effects on performance. The latency-hiding characteristics of simultaneous multithreading allow it to achieve a 2.68 average speedup over a single MP2 processor, while MP2 and MP4 speedups are limited to 1.63 and 1.76, respectively. The bottom line is that simultaneous multithreading makes better utilization of on-chip resources to run parallel applications effectively.

References

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

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

[3] J. Boyle, *et al*. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, Inc., 1987.

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

[5] IBM Corp. RISC System/6000 model J50. http://www.rs6000.ibm.com/hardware/entrprise/j50.html.

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

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

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

[9] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. The M-Machine multicomputer. In *28th*

Annual International Symposium on Microarchitecture, pages 146–156, November 1995.

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

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

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

[13] B. Gunther. *Superscalar performance in a multithreaded microprocessor*. Ph.D. thesis, University of Tasmania, December 1993.

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

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

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

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

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

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

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

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

[22] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. Ph.D. thesis, Rice University, May 1989.

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

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

[25] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 1990.

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

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

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

[29] Sun Microsystems, Inc. Ultra HPC Series Overview. http://www.sun.com/hpc/products/index.html.

[30] 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 Com-*

puter Architecture, pages 191–202, May 1996.

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

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

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

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

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

[36] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing cache port efficiency for dynamic superscalar microprocessors. In *23rd Annual International Symposium on Computer Architecture*, pages 147–157, May 1996.

[37] 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. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

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

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

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

[41] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, pages 28–40, April 1996.