The Effectiveness of Multiple Hardware Contexts

Radhika Thekkath and Susan J. Eggers†
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195
{radhika,eggers}@cs.washington.edu

Abstract

Multithreaded processors are used to tolerate long memory latencies. By executing threads loaded in multiple hardware contexts, an otherwise idle processor can keep busy, thus increasing its utilization. However, the larger size of a multi-thread working set can have a negative effect on cache conflict misses. In this paper we evaluate the two phenomena together, examining their combined effect on execution time.

The usefulness of multiple hardware contexts depends on: program data locality, cache organization and degree of multiprocessing. Multiple hardware contexts are most effective on programs that have been optimized for data locality. For these programs, execution time dropped with increasing contexts, over widely varying architectures. With unoptimized applications, multiple contexts had limited value.The best performance was seen with only two contexts, and only on uniprocessors and small multiprocessors. The behavior of the unoptimized applications changed more noticeably with variations in cache associativity and cache hierarchy, unlike the optimized programs.

As a mechanism for exploiting program parallelism, an additional processor is clearly better than another context. However, there were many configurations for which the addition of a few hardware contexts brought as much or greater performance than a larger multiprocessor with fewer than the optimal number of contexts.

1 Introduction

Multiple hardware contexts are a technique for tolerating long instruction latencies [3, 4, 12, 21]. When one thread interlocks for a multi-cycle operation, rather than waiting until the operation completes, a multithreaded processor switches to another context and continues execution. For coarse-grain multithreaded processors, a context switch is triggered by long latency memory references (e.g., cache misses), and synchronization instructions.

Two aspects of overall performance are affected, one counteracting the other. The performance benefit is increased processor utilization, because an otherwise idling processor does useful work on another thread. But the increase in processor utilization comes at the expense of degraded cache and network performance, because executing multiple threads in a single cache architecture increases cache conflicts. Both aspects of multithreading performance, processor utilization and cache/network traffic, have been investigated in the past, using studies based on simulation and analytical modeling of multithreaded processors [2, 6, 19, 26]. In this paper we address the performance bottom line, evaluating whether the benefits outweigh the drawbacks for coarse-grain multithreading, and whether the extra hardware needed for multiple contexts has adequate payoffs.

We approach the questions from several different angles. First, we evaluate the effectiveness of multithreaded architectures by examining the marginal performance improvement (or loss) when successively increasing the numbers of hardware contexts, both for a uniprocessor and multiprocessors of several sizes. Since several configurations suffer from conflict misses, we vary cache associativity, to gauge its ability to reduce conflicts. We also include a second-level cache to understand its impact on multithreading. Our results indicate that with programs optimized for data locality, performance improves with increasing numbers of contexts under all our architectural variations. However, with unoptimized applications, the lowest execution time often occurs with only 2–3 contexts, before either negative cache effects dominate or performance improvements level off. Set-associativity and a second-level cache have some benefits for programs with large cache conflicts. Multithreading does not significantly benefit uniprocessors; improvement is slight, and only with two contexts.

Multiple hardware contexts can also be viewed as a type of parallel hardware that exploits program parallelism by overlapping the execution of multiple threads. To evaluate multithreading as a parallel technique, we compare it to a more traditional form of parallel hardware, multiple processors. Multiprocessors and multiple hardware contexts both exploit parallelism, but in different ways and with different hardware costs. Multiprocessors execute multiple threads in parallel, each with their own functional units and caches. They bring the bigger benefit, and they should, because their hardware resources are vastly larger. As explained above, multithreaded architectures overlap, rather than parallelize, the execution of several threads on a single processor. Their gain is less, but so are their hardware requirements.

For our comparison of multithreading and multiprocessing we first examine their performance in their natural hardware environments, simultaneously decreasing the number of processors and increasing the number of multiple hardware contexts to maintain a "comparable level of parallelism", and fixing the cache size per processor. Although bigger multiprocessors generally have the best overall performance, our results showed that quite often the

†This work was supported in part by ONR grant #N00014-92-J-1395 and PYI award #MIP-9058439. R. Thekkath was supported by a fellowship from IBM Corporation.

Appeared in ASPLOS, San Jose, CA, October 5–7, 1994, pages 328–337.

Program Name	Number of Threads	Mean Thread length	Problem Domain	Optimization by
Barnes	12	5.7M	Scientific	Programmer
Locus	12	5.2M	CAD	Programmer
MP3D	12	9.1M	Scientific	Programmer
Water	12	4.9M	Scientific	Programmer
Pverify	12	101.0M	CAD	Compiler
Topopt	9	64.3M	CAD	Compiler
FFT	32	0.19M	Math	None
Gauss	127	0.21M	Math	None
Grav	22	0.76M	Scientific	None
Patch	77	0.49M	Graphics	None

Table 1: The workload

Parameters	Values	
Processors	1–128	
Hardware Contexts	1–128	
Non-memory Instruction	1 cycle	
Context Switch Time	6 cycles	
Context Swap Time	200 cycles	
	L1 Cache	**L2 Cache**
Cache Size	32KB, 64KB	infinite
Cache Associativity	direct, 2-way	direct
Cache Block Size	16 Bytes	16 Bytes
Cache Hit Time	1 cycle	4 cycles
Network Latency (no load)	50 cycles	

Table 2: Architectural parameters to the simulator

performance gained by adding a few hardware contexts to a small multiprocessor matched or bettered that of a larger machine.

We then partially rectify the hardware imbalance by giving multiple hardware contexts the same cache advantage as a multiprocessor, and evaluate the extent to which their performance approaches that of the more hardware intensive architecture. We found that, although the fairer cache organization eliminates conflict misses, it cannot compensate for the inability to execute in parallel.

For all experiments we use a workload that consists of ten explicitly parallel programs that are representative of applications in the scientific and engineering problem domains. Traces of these programs are simulated on a shared memory architecture, in which the number of hardware contexts, processors, remote latency, and cache details are varied, and the impact of hardware changes is reflected on cycle time. The thread placement strategy load balances threads across processors. Performance is measured by total simulated execution time; and component metrics are used to explain performance differences.

The remainder of the paper covers the following material: Section 2 describes the application suite and the simulation methodology; Section 3 evaluates the impact of multiple hardware contexts on execution time; Section 4 covers their comparison to multiprocessors; Section 5 discusses related work and Section 6 concludes.

2 The Application Suite and Simulation Methodology

The simulator is trace-driven, and simulates a distributed, shared-memory multiprocessor in which processors are connected by a two-dimensional mesh. All of physical memory is divided equally among the processors. The first few bits of an address determine the particular processor, and hence the physical location of a piece of data. A memory access is local when the address maps to the current processor, and remote for all other processors. Each processor has its own data cache, and coherency is maintained with a distributed, directory-based cache coherency protocol [7].

The simulator requires three inputs: a set of memory reference traces, a mapping that associates each program thread to a specific processor, and an architectural description file. Given these inputs the execution of the program is simulated on the given architecture and statistics are generated.

The workload spans a wide range of problem domains and comprises two types of explicitly parallel programs: data locality-optimized (either by the programmer or the compiler [13, 14]), and locality-unoptimized (Table 1). These two sets of programs were written using different parallel programming paradigms. The optimized programs, which include some of the SPLASH benchmarks [20], use *fork()* instructions to explicitly create processes. The unoptimized programs use the PRESTO parallel programming environment [5], and create threads using calls to a user-level threads library. In both types of programs, thread synchronization is performed using global barriers, and mutual exclusion is implemented with explicit lock instructions.

Memory reference traces from these programs were generated from a parallel execution on a Sequent Symmetry [22] using MPtrace [8], a parallel program tracing tool. Each thread is traced separately, enabling its separate simulation in the target architecture. The trace files capture synchronization instructions, but exclude spin-wait instructions, since locks may be acquired in a different order on the target architecture for a given simulated execution of the program. The simulator ensures that locks are acquired in some legal order, i.e., threads enter critical sections one at a time.

The application's threads are statically distributed among the processors and do not migrate during execution. The thread-to-processor mapping is fixed for a given application and number of processors. We use a load balanced thread placement. Since thread lengths are known from the trace file lengths, an oracle load-balancer distributes threads across processors, attempting to equalize the number of instructions executed per processor. In a previous study [23], we showed that load balancing was the superior placement algorithm on a wide range of processor/hardware context configurations for a similar suite of applications. Therefore we use it here. Load-balanced placements are practical to implement, and are frequently used in real systems by both compiler- and runtime-based scheduling algorithms [16, 18, 24].

The architectural description file specifies hardware parameters (Table 2). The simulator comprises three separate modules: processor, cache and interconnect. We discuss each in greater detail, including chip area and timing considerations.

The Processor

Each processor can model one or more hardware contexts. Simulations with single-context processors provide the base case for performance comparisons. Single-context processors usually context switch based on a timer interrupt; we use a 4000 cycle[1] interval to switch between threads scheduled on the same processor. The simulator assumes a 200 cycle[2] penalty for a context swap operation.

With multiple hardware contexts per processor, the processor switches between instruction streams when the executing thread incurs a cache miss or a synchronization operation failure. The switch to a thread loaded in another hardware context takes 6

[1] Estimated from a DECstation 5000/240 running Ultrix V4.2A.

[2] Number of cycles estimated to store/load a full register set to/from local memory.

cycles[3], and is done using a round-robin policy. With more threads than hardware contexts, the processor swaps the currently executing thread with another, also using a round-robin policy. Context swapping is initiated when there are unloaded ready threads, and a loaded thread is blocked by a synchronization event (locks or barriers) or when all loaded threads are blocked for a memory request.

Multiple hardware contexts incur a slight cost in processor chip area. Assuming a 32 integer and 32 floating point register set (64-bit machine) with two read and two write ports, the equivalent cache chip area is about 2 KBytes [17]. Thus, the area needed for multiple contexts is significant only with more than about eight register sets (16 KByte cache equivalent). The effect on cycle time is a more serious consideration. Doubling the number of register sets requires an extra demultiplexing stage, increasing the processor cycle time by about 5%, assuming a 4–8ns clock, and a 200ps gate delay. This increase is accounted for in the reported execution times.

We assume that all instruction references take 1 cycle, i.e., they always hit in a separate I-cache that is not modeled. We also assume that multi-cycle floating point units are pipelined to hide floating point operation latencies. Multi-cycle operations can increase the overall execution time of a thread, but for a fixed number of processors, given identical thread placements, all experiments should show comparable increases. Hence we ignore their detailed simulation.

In addition to reporting the simulated execution time, the processor unit also maintains statistics on the cycles spent doing useful work, context switching and context swapping

The Cache

The simulator can model two levels of a cache hierarchy with direct-mapped or set-associative caches. The first-level (L1) cache is assumed to be on-chip with a 1 cycle hit time. It is 32 KBytes for the locality-optimized programs and 64 KBytes for the unoptimized. The infinite second-level (L2) cache is assumed to be off-chip with a 4 cycle access time.

Larger or set-associative caches have longer access times, and this can increase the processor cycle; therefore the simulator makes cycle time adjustments when the on-chip cache size is bigger or set-associative. Assuming a 32 KByte, direct-mapped base cache, the simulator adds 10% to the processor cycle time with a doubling of the cache size or with two-way set-associativity [25], again, assuming a 4–8ns clock.

To support context switching on cache misses, a multithreaded processor must implement a lockup-free cache [15]. The lockup-free mechanism must allow at least as many outstanding memory requests as there are hardware contexts. A lockup-free cache that can support 16–32 outstanding memory requests is complex, although its chip area requirement is not significant [9]. Our simulator assumes that such a lockup-free cache can be implemented.

The cache unit maintains statistics on the individual cache miss components of first-reference, intra- and inter-thread conflict, and invalidation misses.

The Interconnect

We assume a local memory access time of 8 cycles and use 50 cycles for remote memory access times over the multipath interconnect (under no contention). Uniprocessors are assumed to have a large local memory and hence most references (not all) are local, only incurring an 8 cycle penalty.

[3] Typically the time taken to drain the pipeline. This can be reduced with extra hardware support.

We model network contention using the analytical model described and validated by Agarwal [1]. It is a reasonable compromise between a simplistic fixed latency model and a detailed, time-consuming interconnect simulation. The packet injection rate into the network is based on the per-processor cache miss rate, and is adjusted frequently during simulation to correspond to current cache miss rates.

3 Evaluating the Additional Context

The primary goal of this section is to evaluate the effectiveness of multithreading, using execution time as the base criterion. We do this by measuring the simulated execution time of an application with increasing numbers of hardware contexts. The underlying architecture for the experiments has a fixed-size, direct-mapped on-chip cache (32 or 64 KBytes, depending on the application), a fixed number of processors (which we vary from one to multiprocessors of several sizes) and a load balanced thread placement policy. The processor cycle time is increased with increasing numbers of contexts. For a given number of processors, the number of threads per processor is constant, regardless of the number of hardware contexts. First we present results for the base architecture described above, and then discuss program behavior when the architecture is varied.

3.1 Locality-Optimized Applications

Multiprocessor configurations for the locality-optimized programs show a consistent drop in execution time with increasing contexts. (Execution times for representative applications, BarnesHut and Mp3d, are shown in Figures 1 and 2.) Reductions across all contexts ranged up to 45%, depending on the size of the multiprocessor. The largest incremental gain was always seen when increasing contexts from 1 to 2 (26% on average), especially with fewer processors.

The drop in execution time can be explained by examining processor utilization and cache miss components (Figures 3 and 4 show sample graphs for BarnesHut). As expected, processor utilization rose with increasing hardware contexts, as more and more idle cycles were utilized by work from another thread. However, even with the maximum number of contexts, the processors were not yet fully utilized and performance was not hindered by an increase in context switch overhead. The difference in the slopes of BarnesHut and Mp3d is due to the difference in the cache miss rates of the two programs. Mp3d's high cache miss rate (on average about 14%), results in very low processor utilization, as low as 10% with one context multiprocessors (Figure 5). With more latency to hide, multithreading has a bigger benefit. On the other hand, BarnesHut has a lower cache miss rate (2% on average), higher processor utilization, and as a consequence, the benefit from multithreading is smaller.

For most locality-optimized programs (LocusRoute is the only exception), and for each machine configuration (including uniprocessors), total cache misses remained relatively constant with increasing hardware contexts. However, the classification of misses within the total shifted. Intra-thread misses became inter-thread misses, as thread execution was interleaved and the caches contained the data of multiple threads. Invalidation and first reference misses remained constant for a particular multiprocessor, but rose with increasing numbers of processors, as threads that shared data were spread across more processors. This increase was usually at the expense of conflict misses (occasionally total misses rose, but only very slightly).

The consistent total cache behavior is primarily due to the data partitioning policy adopted in these programs: each thread only updates its own data space, but reads from the entire space, which

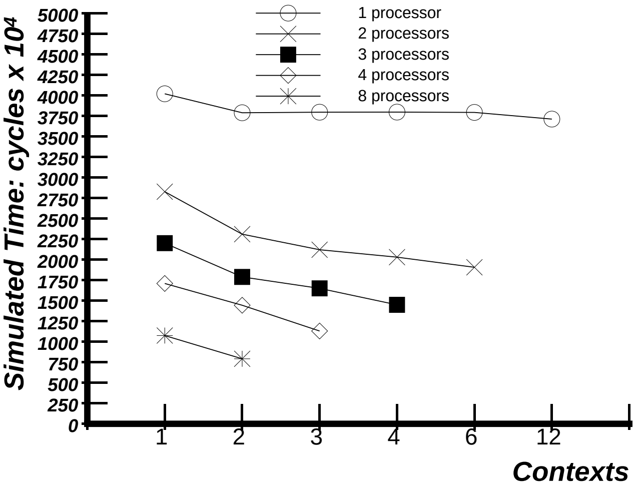

Figure 1: BarnesHut: Simulated execution time for the base hardware configuration. The X-axis is a selected number of contexts.

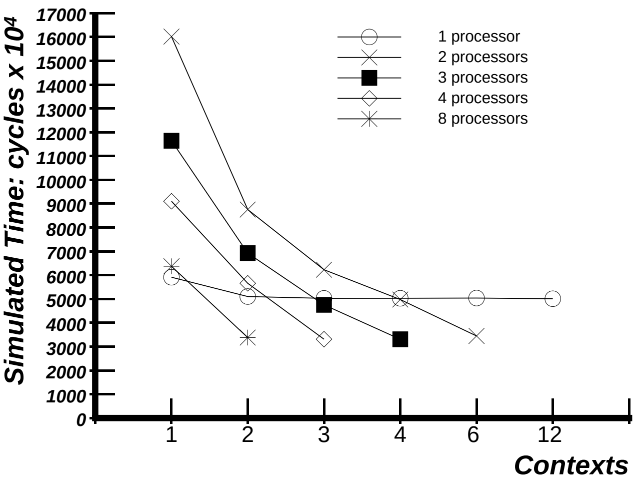

Figure 2: Mp3d: Simulated execution time for the base hardware configuration. The X-axis is a selected number of contexts.

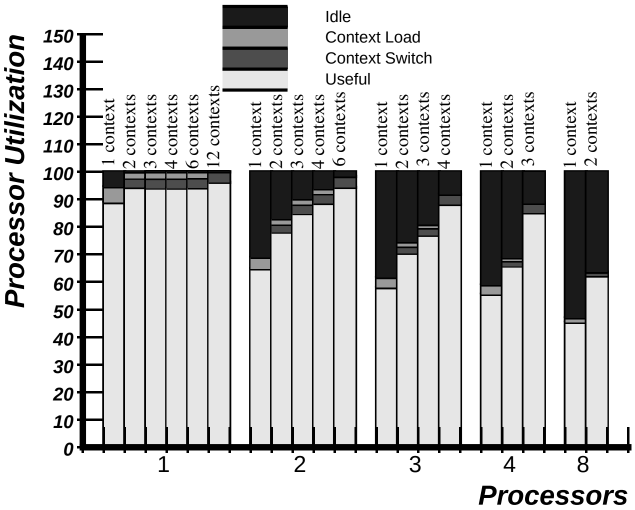

Figure 3: BarnesHut: Processor utilization (Y-axis) for selected hardware contexts, for fixed numbers of processors (X-axis), for the base hardware.

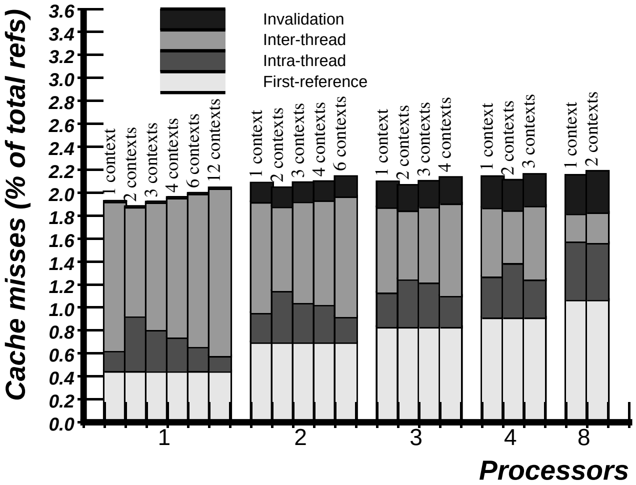

Figure 4: BarnesHut: Cache miss ratio (Y-axis) for selected hardware contexts, for fixed numbers of processors (X-axis), for the base hardware.

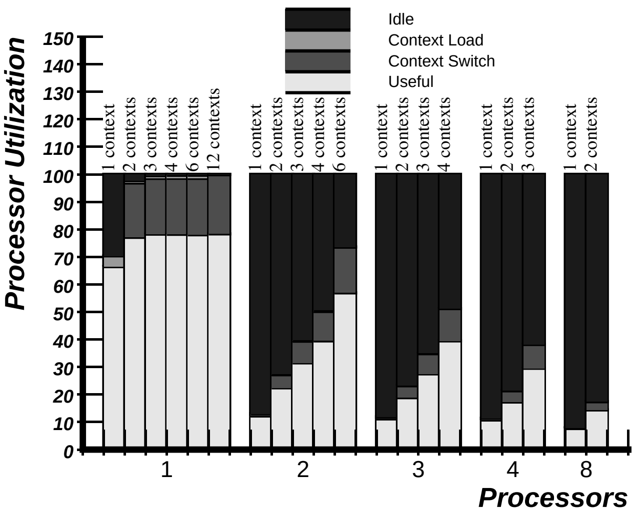

Figure 5: Mp3d: Processor utilization (Y-axis) for selected hardware contexts, for fixed numbers of processors (X-axis), for the base hardware.

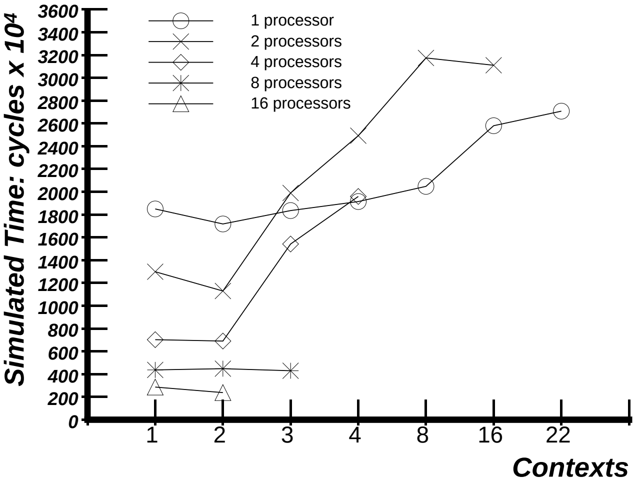

Figure 6: Grav: Simulated execution time for the base hardware configuration. The X-axis is a selected number of contexts.

is usually large. The rationale for this organization is that in a multiprocessor execution with only one hardware context per processor (the original target machine), it generates few invalidations. Multiple contexts hardly change the total cache miss picture: invalidations do not change significantly; and intra-thread conflicts translate to inter-thread conflicts, since the entire data space is still being accessed, but by different threads.

Other studies [2, 26] report an increase in the cache miss rate with increasing contexts, e.g., Weber and Gupta [26] reported increased cache misses for LocusRoute (0.55% to 1.22%) and Mp3d (3.5% to 4.2%) when increasing contexts from one to four. These studies equated hardware contexts with program threads, and varied the two together. More contexts meant more executing threads, and consequently increased cache contention. Since we are evaluating the efficacy of additional hardware contexts, we keep the other factors constant. Both the number of threads per application, and the number of threads per processor for a particular processor configuration, are the same for all experiments.

LocusRoute deviates slightly from the others. Its execution time follows the same declining trend, but its worst performance was with 2 contexts, rather than 1. LocusRoute's low cache miss rate (1.5%) delivers decent processor utilization (70%) without multiple hardware contexts. With only two contexts, the processor is unable to hide latency effectively, since the higher number of lock synchronizations work at cross-purposes with frequent context switching and swapping. Simply, when a thread holding a lock is switched out on a cache miss, utilization on the current processor is improved, but the progress of threads on other processors is delayed if they are blocked for the same lock. With only two contexts, the lock-holding thread is more likely to get swapped out, further delaying program completion. Additional contexts reduce swapping frequency, allowing the lock-holding thread to resume execution sooner. Consequently, processor utilization rises and execution time drops.

Uniprocessor performance improved with 2 contexts, but, unlike the multiprocessor configurations, benefits from additional contexts were marginal. The lackluster performance of additional contexts for uniprocessors can be explained by their lower memory access times. Recall that in the uniprocessor configuration most memory accesses are local. Consequently, they have less latency to hide and relatively high processor utilization (a range of 66%–88% for the whole workload), even with a single context. Two contexts have some benefit, and execution time drops marginally. However, there are not sufficient idle processor cycles to support a large number of contexts. Since cache misses remain constant with increasing hardware contexts, the execution time is flat.

The relationship between uniprocessor and multiprocessor execution times is very sensitive to the number of cache misses. Since multiprocessors (with mostly remote memory) pay a greater cache miss penalty than uniprocessors, programs with worse cache behavior have better uniprocessor execution relative to the execution of small multiprocessors. For example, the execution of uniprocessor Mp3d (12%) is far better than uniprocessor BarnesHut (2% miss rate) relative to their multiprocessor executions.

3.2 Locality-Unoptimized Programs

Multiple hardware contexts had a much less favorable effect on unoptimized programs (e.g., Grav in Figure 6). Although execution time fell or remained constant when increasing contexts from one to two, additional contexts usually degraded performance (by as much as 123% for Grav from 2 to 3 contexts with 4 processors); the lone exception was Gauss (Figure 9), where performance improved with increasing contexts, like the optimized programs.

Cache conflict misses increased with increasing hardware contexts, degrading performance. Unlike their optimized counterparts, the unoptimized programs do not have a partitioned or regular data access pattern, and have smaller data set sizes. With only one or two contexts, all accessed data fit in the cache (less than a 2% cache miss rate, except Gauss with 4%–8%). With more than two contexts, the working sets of the threads begin to interfere in the cache, drastically increasing total cache misses (a 60%–300% increase in cache misses over the entire range of contexts after 2).

Increased cache misses increase the context switching frequency, and the context switch overhead. All threads eventually block for memory requests, and the processor idles or threads are swapped (Grav in Figures 7 and 8 is typical). With more processors (and consequently fewer scheduled threads per processor), multiple contexts have a smaller effect on cache conflicts.

With Gauss, (Figures 9 and 10), the rise in misses was caused by invalidation misses rather than conflict misses: increasing hardware contexts decreased processor locality more than it increased inter-thread conflicts, since threads share a lot of data, but have small working sets. This effect is, of course, greater with more processors. Like conflicts, increased invalidation misses can stall threads; however, the percentage increase in invalidations for Gauss is smaller than the percentage rise in conflict misses for the other programs. This, plus Gauss's finer granularity, which provides more threads for instruction overlap, reduces execution time when adding contexts.

Unlike the optimized programs, uniprocessor execution time of the non-optimized programs follows the multiprocessor behavior, increasing beyond 2 or 3 contexts.

3.3 Using Set-Associativity to Reduce Conflicts

Results from Section 3.2 indicate that for some unoptimized programs (Grav, Patch and FFT), the effectiveness of multithreading is limited due to increasing cache conflict misses. Although the problem is not as severe for the optimized applications, conflict misses are the largest component of their total misses. Thus, cache organizations designed to reduce conflict misses should expand the range of effective multithreading. To test this hypothesis, we converted the direct-mapped cache into a two-way set-associative cache of the same size, and repeated the experiments. The increase in cache access time with associativity was reflected by a 10% increase in processor cycle time.

Two-way set associativity showed some benefits for the unoptimized programs. For higher degrees of multithreading, where conflict misses increased greatly, set-associativity reduced the miss rate and improved execution time over the direct-mapped cache (see the comparative curves in Figure 12). Depending on the application, the number of processors, and the number of hardware contexts, cache misses were reduced by 2.3%–87%, while execution time dropped by as little as 0.2%, and as much as 61%. Set-associativity had no benefits for Gauss: this is not surprising, since invalidation misses, rather than conflict misses, were the predominant cache miss component.

The effect on the execution times of the optimized programs was insignificant. Topopt and Pverify saw the largest impact with 25% and 40% drops in cache miss rates, respectively, resulting in a 2%–5% reduction in execution time. With Mp3d, BarnesHut (Figure 11), and LocusRoute, the decrease in cache miss rate (0.5%–2%) was only sufficient to offset the increase in cycle time, and the effect on execution time was negligible.

3.4 Effect of a Large Second-Level Cache

To evaluate the impact of multiple hardware contexts on processors with a cache hierarchy, we simulated a configuration with an off-chip, infinite second-level (L2) cache, in addition to the L1 cache. The L2 cache is assumed to be slower, with an access time of 4 cycles. We ran two sets of experiments, based on whether a context switch was triggered by a miss in the L1 or L2 cache.

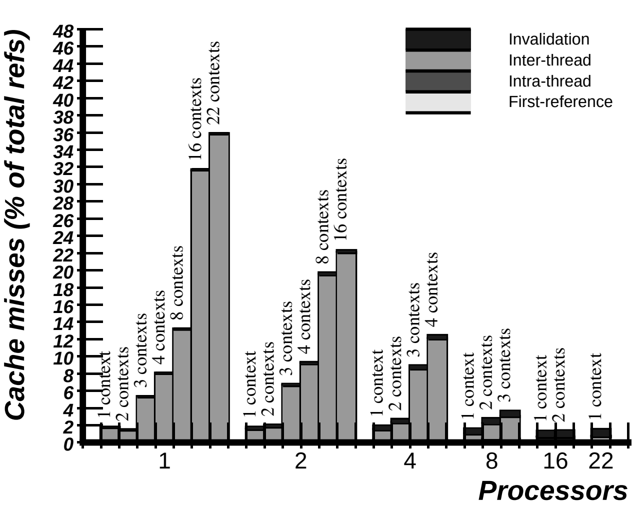

Figure 7: Grav: Cache miss ratio (Y-axis) for selected hardware contexts, for fixed numbers of processors (X-axis), for the base hardware.

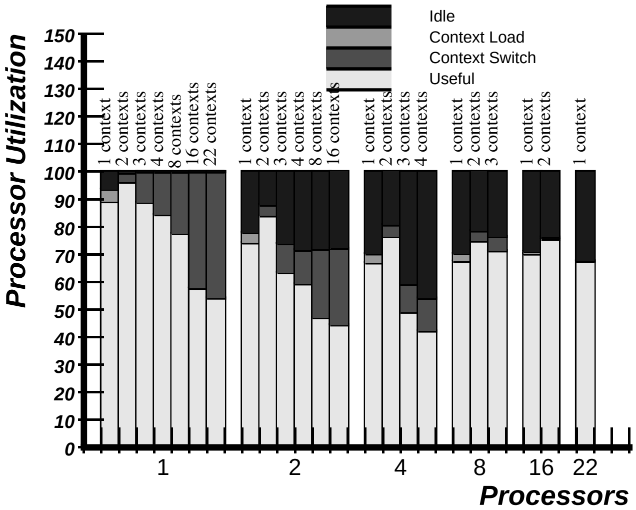

Figure 8: Grav: Processor utilization (Y-axis) for selected hardware contexts, for fixed numbers of processors (X-axis), for the base hardware.

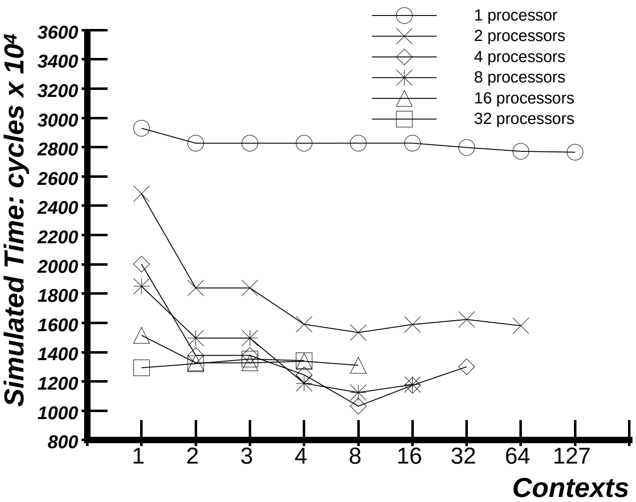

Figure 9: Gauss: Simulated execution time for the base hardware. The X-axis is a selected number of contexts.

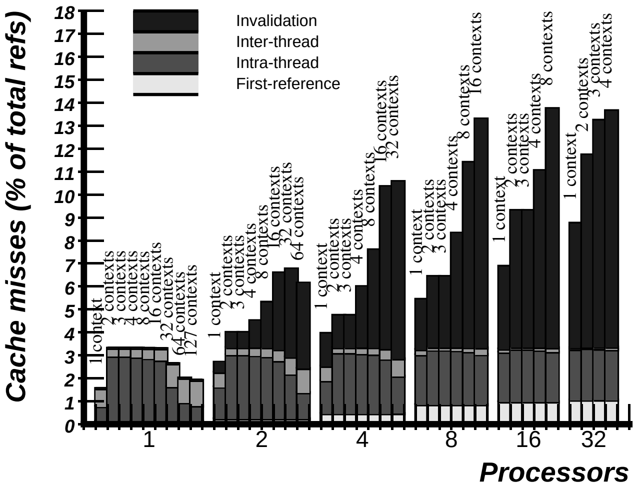

Figure 10: Gauss: Cache miss ratio (Y-axis) for increasing hardware contexts, for fixed numbers of processors (X-axis), for the base hardware.

For the unoptimized programs, the L2 cache improved performance only when the number of hardware contexts was large and the number of processors small, i.e., when conflict misses were quite high. (See Grav in Figure 12, 8 and 16 contexts.) With fewer hardware contexts the L2 cache sometimes had a detrimental effect. This can happen when the processor context switches on an L1 cache miss, and the miss is satisfied by the 4 cycle L2 cache. The speedy return of the data replaces a cache block from the working set of threads that will execute prior to the missed thread, hence increasing inter-thread conflict misses. This occurs for Patch and Grav, where the initial dip in execution time with 2 and 3 contexts and no L2 cache, is replaced by a rise in execution time (again, Figure 12). Hence, for the unoptimized programs, the L2 cache can block the limited performance gain with multiple contexts.

The increase in inter-thread conflicts does not occur with the locality-optimized programs, because of the large amount of read sharing among threads. Without the negative effect of the new inter-thread conflicts, the normal benefit of a second level cache, a reduction in miss penalties for conflict and capacity misses (the largest miss components in these programs), dominates. Consequently, the infinite L2 cache always improves performance, reducing execution time from 4%–25%. The biggest improvement (15%–25%) was seen for the single context machines, since they have the lowest processor utilization.

Finally, we compare the two context switch triggering policies. For most programs and hardware configurations, performance between the two was comparable. In only two situations, very high and very low conflict misses in the unoptimized programs, were differences observed. First, with many contexts and few processors (when conflict misses are high), switching on L2 misses increased idle cycles because of the wait for the 4 cycle L2 access time, Hence a policy that switches on L2 misses can do slightly worse than one that switches on L1 misses. On the other hand, when L1 cache misses are low, switching on L2 misses can pay off marginally, because it allows a thread to build up more of its working set in the L1 cache.

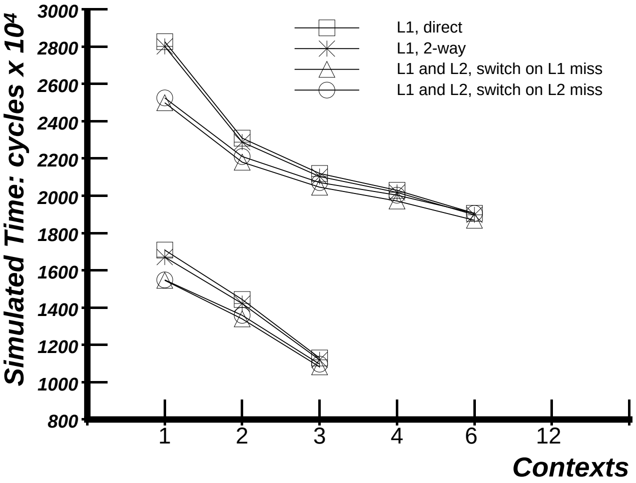

Figure 11: BarnesHut: Comparing different cache architectures for 2 processors (top) and 4 processors (bottom) and selected hardware contexts.

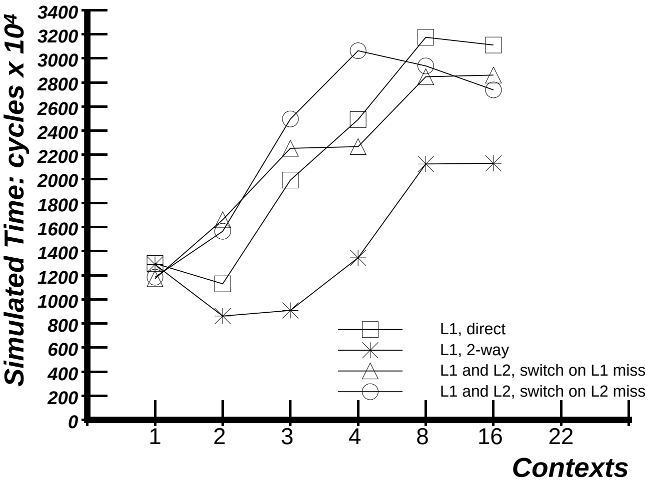

Figure 12: Grav: Comparing different cache architectures for 2 processors and selected hardware contexts.

4 Comparing Multiple Contexts with Multiple Processors

This section evaluates multithreading as a parallelism-exploiting technique by comparing its performance relative to multiple processors. Multiprocessors, of course, have a greater hardware cost (advantage): more functional units and a larger system-wide cache memory. To partially rectify this hardware imbalance, we increase the cache size when there are fewer processors and more contexts per processor, keeping the total cache hardware constant for the entire machine. We then reevaluate the performance, using these results to isolate the gain from parallel functional units. All performance results reported in this section use a two-way set associative L1 cache (Section 3.3 showed set-associativity to have equal or better performance than a direct-mapped cache of the same size) and no L2 cache (the L2 cache was not consistently beneficial).

Results in Sections 3.1 and 3.3 demonstrated that optimized program performance improved with additional hardware contexts for all processor configurations (on average 10%, for each additional context). Increasing the number of processors for specific hardware context sizes has an even greater positive effect on execution time (on average 32%, for each additional processor). The overall best performance was obtained with the maximum number of processors, although the largest incremental gain from multiprocessing was seen when the number of contexts was largest, i.e., going from 1 to 2 processors. Pverify (dotted lines in Figure 13), is representative of the optimized applications.

These studies demonstrate that, for best performance for the optimized programs, trade-offs have to be made in favor of processors, i.e., it pays to spread a fixed number of contexts across as many processors as possible. In addition, for a given multiprocessor, the optimal number of contexts (the lowest points in the graphs), is the maximum number of contexts that an application's threads will support.

Mp3d was the exception (Figure 15). With smaller multiprocessors and the maximum number of contexts per processor, execution times were similar to those of larger machines. Multithreading's success with Mp3d is due to the interaction between its having program behavior that is conducive to multithreading (a relatively high miss rate, 12% on average, and very low processor utilization, 10% at worst, on single context multiprocessors) and its thread synchronization mechanism. Mp3d uses barriers to synchronize threads, and does not have any high contention locks. With locking, an increase in the number of hardware contexts can slow down the progress of the lock-holding thread. On the other hand, with few contexts, the lock-holding thread may be swapped out. In either case program execution suffers. With barriers, context switching passes by those threads that have already reached the barrier, allowing the others to make better progress. In addition, and just as important, Mp3d's threads were approximately the same length, and therefore busywaiting for barriers was low.

Unlike the optimized programs, the optimal number of contexts for the unoptimized programs is typically two or three, for all multiprocessor sizes. Moreover, there is a performance benefit from increasing processors (on average 42% for each additional processor), only until saturation is reached, i.e., several of these programs do not scale beyond some number of processors. At saturation, execution time shows no improvement regardless of the number of contexts (see Patch in Figure 14, with 16 processors and greater). The occasional exception to improved performance is with the transition from 1 to 2 processors, when execution time rises (Patch with 8 and 16 contexts, again in Figure 14).

Gauss is an exception among the unoptimized programs. As discussed in the previous section, Gauss's execution is governed by invalidation misses and context switch time, rather than conflict misses. Best performance is achieved, not with the maximum number of processors, but when processors and hardware contexts are balanced (4 processors and 8 contexts).

That larger multiprocessors have the better performance is not surprising, since they have a great hardware advantage, i.e., more overall cache and functional units for parallel execution. What *is* surprising, however, is that smaller configurations (fewer processors) often have comparable or better performance than larger machines. (Such instances are shown using solid lines in Figures 13 and 14.) For example, in Pverify the execution time of 3 processors and 4 contexts per processor is 16% faster than 4 processors and 2 contexts. For the unoptimized programs performance is often better with both fewer processors and fewer total hardware contexts. For example, in Patch 3 processors and 4 hardware contexts (12 contexts in total) is 61% faster than 4 processors and 8 contexts (32 total). Since adding hardware contexts is less costly than increasing the number of processors, they provide a cheaper alternative for obtaining comparable or better performance, in many cases. In addition, smaller multiprocessors often have a smaller interconnect, with a subsequent reduction in both interconnect cost and remote memory latency.

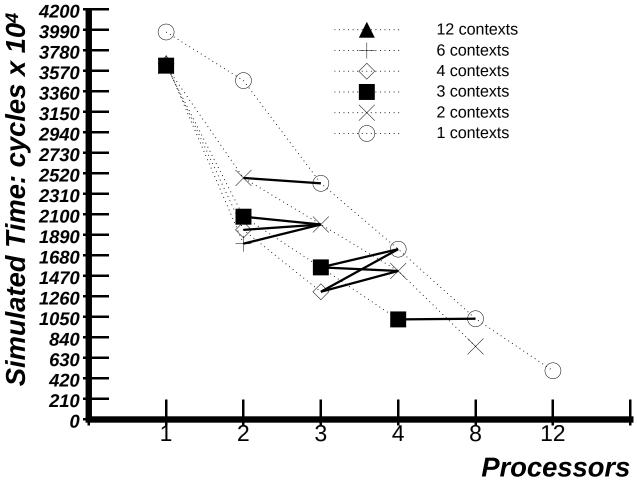

Figure 13: Pverify is typical of the optimized applications (2-way set-associative L1 cache, no L2 cache, a selected number of processors). Solid lines connect machines with more processors to those with fewer processors that have comparable or better performance. Dotted lines connect processors with the same number of hardware contexts, but varying numbers of processors.

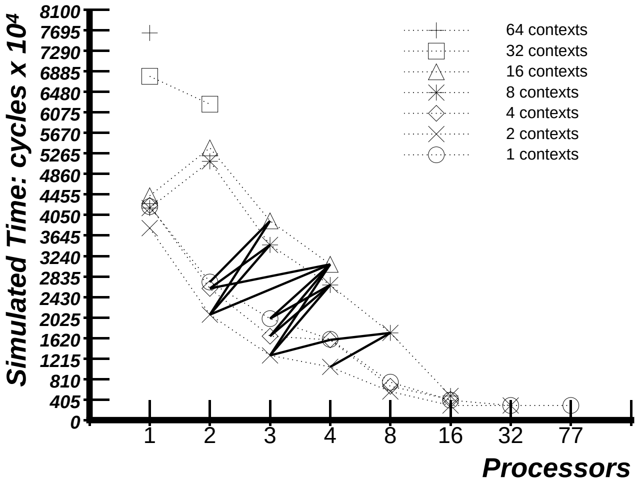

Figure 14: Patch is representative of the unoptimized programs. (Same configuration, and solid/dotted line semantics as Figure 13).

Equalizing Cache Hardware

Miss rate component analysis reveals that the combined intra- and inter-thread conflict misses dominate all others for both the optimized programs (over all processor/context configurations) and unoptimized (the higher context processors only). Therefore equalizing the cache hardware for all multiprocessor sizes, i.e., giving machines with fewer processors and correspondingly more hardware contexts the cache hardware advantage of the larger multiprocessors, may drive their performance toward that of the larger machines. To test this hypothesis, we scaled up the cache size of the small multiprocessors to match that of the overall cache size in the largest multiprocessors. For example, FFT has 32 threads; its largest multiprocessor configuration was therefore 32 processors, each with a single context and a 64 KByte cache. For each multiprocessor configuration that halved the number of processors and consequently doubled the number of hardware contexts, we also doubled the cache size. Since the goal of the experiment was to study the limit on the performance improvement, we made optimistic assumptions about the feasibility of such large first-level caches, and continued to assume a one cycle access time.

Two aspects of performance were compared to previous results that used smaller caches: the improvement in execution time for the same size multiprocessor with the same number of hardware contexts, and whether the best performance obtained for a specific multiprocessor size (the lowest point on the dotted curves in Figures 13 and 14) could be improved.

Equalizing the system-wide cache showed a 33%–94% drop in the overall cache miss rate, with a corresponding 5%–46% drop in execution time, over all configurations. Applications for which multithreading had already paid off relative to multiprocessing, despite a caching handicap (Mp3d: Figure 16 and Gauss), saw the most benefit from the additional cache size. Their gains for the configurations with the best performance ranged from 7% to 17%. For the other programs, however, the best case configurations rarely improved, because their cache miss rate was already small. The exceptions were Pverify and Topopt, which showed 5% drops in execution time.

Using a Per-context Cache

Increasing the cache size does not always completely eliminate conflict misses. Hence we ran experiments that increased the degree of set-associativity as well. Thus, when raising the number of hardware contexts from 1 to 4, we enlarged the cache size by 4 and also made it 4-way set-associative. Note that increasing the degree of associativity with the cache size models a processor configuration with a separate cache per hardware context.

These experiments showed a 3.5%–81% reduction in cache miss rates when compared to the same size direct-mapped cache, corresponding to a 0.5%–52% reduction in execution time. At the high end of the range, associativity eliminated specific cases of thrashing in the direct-mapped cache. For example, a 4-way set-associative cache completely removed inter-thread conflicts in Pverify (8 processors); total cache misses and execution time reduced by 38% and 52%, respectively. With smaller multiprocessors, execution time per processor is larger, and a modest decrease in inter-thread conflicts does not significantly affect the overall execution time.

There were a few exceptions, cases when the execution time increased slightly with set-associativity. This happened when the cache miss rate (conflict misses) of the direct-mapped cache was very small, and higher associativity marginally increased the invalidation miss rate.

However, set-associativity did not improve the performance of the best case multiprocessor configurations significantly. Hence, we conclude that for most of the workload, functional unit parallelism, rather than lack of caching, is responsible for the better performance of multiple processors over multiple hardware contexts.

5 Related Work

Past evaluation of multithreaded architectures has mainly focussed on the specific components of performance that are affected by multiple hardware contexts: processor utilization and cache and network performance. Weber and Gupta [26] estimated, via simulation, the extent to which a multithreaded architecture can overcome the effects of long access latencies. They varied the number of contexts (with threads), the context switch overhead and the

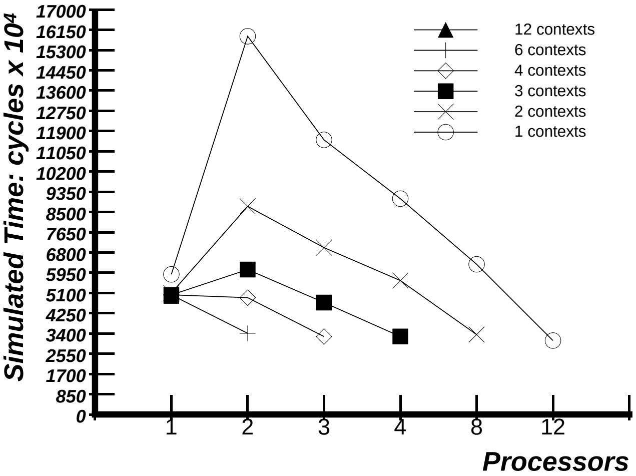

Figure 15: Mp3d's performance is atypical. Fixed cache size per processor, 2-way set-associative L1 cache, no L2 cache and a selected number of processors.

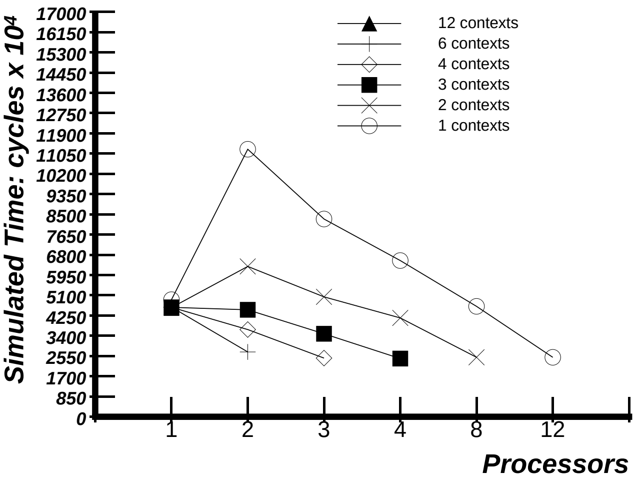

Figure 16: Mp3d with equalized cache hardware; all other parameters are identical to Figure 15.

memory access latency, and found substantial improvement in processor utilization for low context switch overhead configurations. Agarwal [2] presents an analytical performance model that incorporates network traffic, cache interference, context switching overhead and the number of hardware contexts. The results showed that high cache miss rates and increased network contention can hurt processor utilization. Saavedra-Barrera et al. [19] developed a Markov chain model for multithreaded processor efficiency that varies the number of contexts, the network latency, context switch times, and remote reference rate. (They also incorporate cache performance degradation in their model). The study shows that few contexts cannot effectively hide very long memory latencies.

The work by Boothe and Ranade [6] proposes an explicit context switch instruction to group together shared accesses, and hence avoid excessive context switching. As they show, this can improve multithreading's latency-hiding property, but the processor incurs the extra cost of managing multiple memory requests per thread. A bigger drawback comes from the limitation on the network bandwidth. On large-scale shared memory machines, the network can only support a limited number of concurrent memory requests per processor. With every thread grouping its memory operations, this limit could be reached fairly quickly. Such grouping might also be prone to creating hot spots in the interconnect.

Two studies addressed execution time, as does ours, but for different ends. Farrens and Pleszkum [10] studied the improvement in instruction issue with deeply pipelined processors, using two interleaved instruction streams. Gupta et. al. [11] evaluated multiple contexts in the presence of other latency-tolerating techniques, such as prefetching and sequential consistency. Their report is slanted toward this goal and does not evaluate the "goodness" of multithreading in an isolated environment.

6 Conclusions

Our experiments evaluating multiple hardware contexts have demonstrated that they do not bring unqualified benefits. Their impact on execution time depends on data locality in the application, the cache organization and the degree of multiprocessing. When programs have been optimized for data locality, either by the programmer or the compiler, multithreading is effective. For all multiprocessor sizes, improvements in processor utilization were not offset by a rise in cache conflicts, and execution time dropped.

With unoptimized applications, multiple contexts had limited value. Increasing conflict misses (i.e, increasing with the number of hardware contexts), was the chief cause of greater processor idling, and the subsequent rise in execution time. Best performance was seen with only two contexts, and only on small multiprocessors. Set associativity reduced the conflict misses, improving execution time and sometimes moving the best performance point to more contexts. However, second level caching often worsened performance for small multiprocessors with few contexts.

Multithreading was only beneficial on uniprocessors with two hardware contexts. Increasing the number of contexts beyond two had insignificant effect on the optimized programs and hurt the performance of the unoptimized programs.

As a mechanism for exploiting program parallelism, an additional processor is clearly better than another context. In fact, for the optimized programs, it was most beneficial to spread contexts across as many processors as possible. Even when maintaining a constant system-wide cache, or a separate cache per hardware context, multithreading could rarely top the performance of a larger multiprocessor. Multiple contexts had comparable performance only when the cache miss rate was very high and processor utilization low. In other words, the cache hardware could rarely compensate for a multiprocessor's ability to execute in parallel.

However, there were many configurations when the addition of a few hardware contexts brought as much or greater performance than a larger multiprocessor with fewer than the optimal number of contexts. Hence, for some configurations multithreading can provide a cheaper alternative for modest performance gains.

Acknowledgements

We thank Jean-Loup Baer and Hank Levy for helpful discussions that have improved this paper. We also thank the ASPLOS reviewers for their comments.

References

[1] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

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

[3] A. Agarwal, B-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. *17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. *International Conference on Supercomputing*, pages 1–6, June 1990.

[5] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[6] B. Boothe and A. Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. *19th Annual International Symposium on Computer Architecture*, pages 214–223, May 1992.

[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. *Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[8] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–46, May 1990.

[9] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. *21th Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

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

[11] A. Gupta, J. Hennesey, K. Gharachorloo, T. Mowry, and W-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. *18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[12] R. H. Halstead and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. *15th Annual International Symposium on Computer Architecture*, pages 443–451, May 1988.

[13] T.E. Jeremiassen and S.J. Eggers. Computing per-process summary side-effect information. *5th Workshop on Languages and Compilers for Parallel Computing*, August 1992. Also appeared as LNCS #757, pages 175–191.

[14] T.E. Jeremiassen and S.J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. *International Conference on Parallel Architectures and Compilation Techniques*, Montreal, August 1994.

[15] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. *8th Annual Symposium on Computer Architecture*, pages 81–87, May 1981.

[16] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *Supercomputing '92*, pages 104–113, November 1992.

[17] J. H. Mulder, N. T. Quach, and M. J Flynn. An area model for on-chip memories and its applications. *IEEE Journal of Solid-State Circuits*, 26(2):98–106, February 1991.

[18] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

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

[20] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[21] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE, Real-Time Signal Processing IV*, 298:241–248, 1981.

[22] Symmetry Technical Summary. Sequent Computer Systems, Inc.

[23] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. *21th Annual International Symposium on Computer Architecture*, pages 176–186, April 1994.

[24] T. H. Tzen and L. M. Ni. Dynamic loop scheduling for shared-memory multiprocessors. *1991 International Conference on Parallel Processing*, pages II:246–250, August 1991.

[25] T. Wada, S. Rajan, and S. A. Przybylski. An analytical access time model for on-chip cache memories. *IEEE Journal of Solid-State Circuits*, 27(8):1147–1156, August 1992.

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