Integrating Multiple Forms of Multithreaded Execution on multi-SMT Systems: A Study with Scientific Applications

Matthew Curtis-Maury, Tanping Wang,
Christos Antonopoulos and Dimitrios Nikolopoulos
Department of Computer Science
The College of William and Mary
McGlothlin–Street Hall. Williamsburg, VA 23187–8795

Abstract

Most scientific applications have high degrees of parallelism and thread-level parallel execution appears to be a natural choice for executing these applications on systems composed of SMT processors. Unfortunately, contention for shared resources limits the performance advantages of multithreading on current SMT processors, thus leading to marginal utilization of multiple hardware threads and even slowdown due to multithreading. We show, through a rigorous evaluation with hardware monitoring counters on a real multi-SMT system, that in traditionally scalable parallel applications conflicting resource requirements are – due to the high degree of resource sharing – accountable for deeply suboptimal performance. Motivated by this observation, we investigate the use of alternative forms of multithreaded execution, including adaptive thread throttling and speculative runahead execution, to make better use of the resources of SMT processors. Alongside the evaluation, we propose new methods to integrate these techniques into the same binary to maximize performance on multi-SMT systems. Our study shows that combining adaptive throttling and speculative precomputation with regular thread-level parallelization leads to significant performance improvements in parallel codes which suffer from inter-thread interference and contention on SMTs.

1 Introduction

Simultaneous multithreaded processors [17] continue to gain popularity in both mainstream and high-performance computing markets. These processors were introduced in academic studies [11, 17] and later adopted as a core design technology for mainstream processors from Intel, IBM and other vendors [9, 16]. SMTs provide an incremental path to improve the performance of conventional superscalar processors by converting thread-level parallelism to instruction-level parallelism, with only marginal increases in cost, area and power consumption.

The first contribution of this paper is an evaluation of parallel application execution on a real SMT-based multiprocessor, consisting of Intel Hyperthreaded processors. We attain detailed performance measurements from hardware event counters and use them to gain insight into parallel application execution on this architecture. One goal of our analysis is to identify architectural bottlenecks of real, commercially available SMT processors, that reduce the scalability of parallel applications.

Scientific applications typically exhibit a high degree of exploitable thread-level parallelism. SMTs seem to be a first-class choice for the execution of these applications. However, our results show that co-executing threads on SMT processors can often lead to limited performance gains, or even cause a slowdown, due to resource conflicts resulting from the high degree of resource sharing that characterizes the SMT architecture. We consider the effects of resource sharing on many resources within the processor by collecting data from hardware event counters. We specifically consider how resource sharing affects the number of L2 and L3 cache accesses and misses, bus accesses, data TLB misses, stall cycles and execution time.

The second contribution of this paper is the investigation of alternative forms of multithreaded execution, both previously proposed and new ones, in an effort to overcome the poor performance of SMT processors at executing scientific applications. We consider adap-

tive thread throttling and speculative runahead execution. Although these techniques have been used in earlier work as tools to enhance SMT performance, they have been explored in different contexts (i.e. for speeding up sequential desktop workloads and selecting loop schedules). Most importantly, effective integration of these techniques with regular thread-level parallel execution in the same code has not been explored before. We propose methods to integrate adaptive throttling and speculative runahead execution with regular thread-level parallelization using the OpenMP [12] programming paradigm and we show that combining these techniques results in significant performance improvements.

The study presented in this paper is a first step in deriving algorithms for selecting the best mode (or modes) of multithreaded execution for any given program. It is also the first study to investigate whether multiple multithreaded execution modes can be combined effectively in a synergistic manner to maximize performance on multi-SMT systems.

The rest of this paper is organized as follows. In Section 2, we briefly review related work on SMT processors and place our contributions in context. In Section 3 we provide an overview of the experimental setting. In Section 4 we discuss and evaluate the execution of a broad set of parallel applications on a multi-SMT system. Section 5 describes the adaptive thread throttling mechanism we implemented for multi-SMT systems and provides an analysis of its performance. Section 6 goes over our speculative runahead execution approach, its integration with thread-level parallelization and its performance. Finally, in Section 7, we summarize our conclusions.

2 Related Work

There is a significant body of work on hardware and software optimizations for SMT processors. We cover a small fraction of this work here, due to space limitations.

Of particular relevance to the work presented in this paper is related work on the performance trade-offs of SMTs for scientific and technical workloads. Earlier results from both research prototypes and real products indicate that SMTs are well suited for running multiprogram workloads in server environments, but are less effective at running standalone parallel applications [16, 17]. Therefore, many users consider SMTs primarily as throughput-oriented processors and most of the related research is oriented towards improving job scheduling rather than improving individual application performance on SMT processors [14]. One notable exception is the work of the Intel compiler group on exploiting Hyperthreading[1] technology for improving parallel execution speed in multimedia applications [15].

Although a standalone SMT processor by itself may not be the best choice for running a single parallel program, an SMT execution engine built around a superscalar core still provides the potential for significant performance improvements compared to conventional superscalar designs. Therefore, using SMT execution cores in place of superscalar execution cores is a cost-effective option for building chip multiprocessors with a clustered multicore design, such as the IBM Power5 [9, 10]. Our work is highly relevant to hybrid CMP/SMT processors as well, although our results are obtained from a somewhat less aggressive SMP system with Hyperthreaded processors.

SMTs are capable of accelerating sequential code via the use of hardware threads for speculative assisted execution mechanisms. Popular assisted execution mechanisms include speculative precomputation for prefetching of memory accesses and slipstreaming for control and value prediction [2, 6, 13, 18]. Assisted execution mechanisms using threads have been investigated almost exclusively in the context of sequential, pointer-chasing codes. One of the contributions of this paper is that it shows that speculative precomputation using a spare thread can occasionally be an effective alternative to thread-level parallel execution of scientific applications on SMTs. Furthermore, to the best of our knowledge, this paper is the first to integrate speculative precomputation with thread-level parallel execution in the same code, achieving a combined performance benefit.

Throttling concurrency is a simple and effective means for controlling execution and contention for shared resources on SMTs and CMPs. Throttling based on timing measurements of loops has been investigated in the context of parallel loop scheduling for OpenMP [20]. In this work, we evaluate timing-based thread throttling as an alternative to improve the performance of scientific applications on multi-SMT systems.

3 Experimental Setup

We experimented on a Dell PowerEdge server with four Hyperthreaded Intel Xeon MP processors each running at 1.4GHz. Table 1 summarizes the hardware characteristics of our experimental platform with memory

[1] Hyperthreading is a term coined by Intel and refers to the SMT technology used in Intel Pentium 4 processors since 2002.

Processors	4 Intel Xeon HTs, 1.4 GHz, 2 execution contexts per processor
L1 Cache	8KB shared, 4-way assoc. DCACHE. 12KB shared execution trace ICACHE
L2 Cache	256 KB shared, unified, 8-way assoc.
L3 Cache	512 KB shared, unified, 8-way assoc.
D-TLB	64 entries, shared, fully assoc.
I-TLB	2x64 entries, partitioned, fully assoc.
DRAM	1 GB
Bus	400 MHz

Table 1. Hardware configuration of the multi-SMT experimental platform.

hierarchy resources classified as shared or partitioned between the two hyperthreads of a single processor.

To evaluate multi-SMT system performance for parallel workloads, we used the OpenMP versions of seven of the NAS Parallel Benchmarks (version 3.1) [8]. We compiled the benchmarks using the Intel FORTRAN compiler version 7.1. We ran all of the NAS benchmarks with the problem size set to class A, which is large enough to yield realistic results, while ensuring that each application's data set fits entirely in memory.

The operating system was Linux (kernel version 2.4.25), which treats the two hyperthreads on each processor as two logical and equivalent processors. We ran experiments in six configurations: we executed the applications on 1, 2 and 4 physical processors, using either 1 or 2 hyperthreads per processor. We strictly bound threads to logical processors, using the Linux `sched_setaffinity()` system call. We collected both timing data and – using Intel's VTune Performance Analyzer [7] – data from hardware event counters that provide deeper insight into the performance of the processors.

4 Experimental Results

In this section we present the results from the evaluation of the relative performance of the benchmarks with the configurations discussed in Section 3. The different binding schemes are labeled as `(num_processors, num_threads)`, where `num_processors` stands for the number of physical processors onto which the threads are bound and `num_threads` for the total number of threads used for execution. We report timings and results obtained from monitoring several performance metrics, using hardware performance counters. We present a bottom-up evaluation starting from individual performance metrics that pin-point potential performance bottlenecks due to resources sharing on SMT processors and proceeding to the overall performance of the codes. Whenever metrics are normalized, the normalization is with respect to the value attained for the specific metric during the sequential execution, unless explicitly stated otherwise.

4.1 Cache Performance

The L2 cache of Intel's Hyperthreaded processor is shared between the two hyperthreads. The first metric we studied, the number of L2 cache misses observed under each configuration, is depicted in Figure 1. In most cases, the number of cache misses goes up, sometimes dramatically, when two threads instead of one thread are spawned on each physical processor. BT, FT and LU-HP demonstrate this pathology to a large extent. If the working sets of both co-executing threads do not fit together in the L2 cache, then cross-thread cache-line eviction significantly increases the number of misses. The average increase in the number of L2 misses when two threads are used on each processor, taken across the entire suite of benchmarks and all executions on 1, 2 and 4 processors, is 33%. In particular, in BT L2 cache misses increase by 69% on average and in FT L2 cache misses increase by 82% on average.

CG and UA, when executed with two threads on one processor, are exceptions to the dominant trend in L2 misses. These applications benefit from data sharing through the shared L2 cache. However, this benefit is not persistent through the parallel executions on 2 and 4 processors, since it is counterweighted by coherence misses due to invalidations. We believe that the reduction of L2 cache misses in SP when two threads are activated on 4 processors is likely attributed to the same trade-off between the benefit of data sharing through the L2 and the penalty of coherence misses, although we have yet to verify this observation.

When only one thread per SMT is used, the cumulative number of cache misses per processor goes down in a few cases when the benchmarks are executed in parallel. In these cases the benchmarks are able to benefit from the additional cache space provided by multiple processors. The effect is most pronounced in CG and somewhat in UA. However, in most cases the cumulative number of L2 misses remains about the same or even goes up, because of cache coherence protocol invalidations.

The number of L2 accesses (shown in Figure 2), a direct indicator of L1 cache misses, goes up from one thread to two threads per processor by 42% on average.

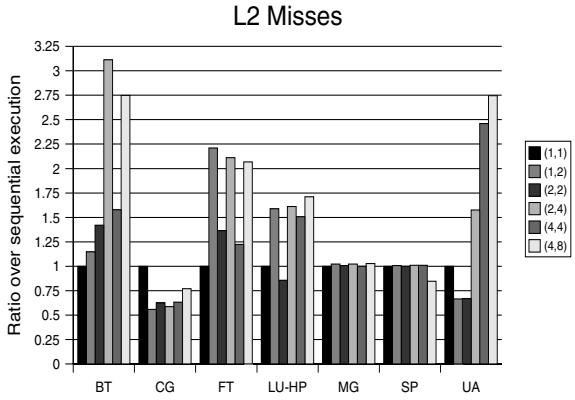

Figure 1. Normalized number of L2 cache misses of the benchmarks.

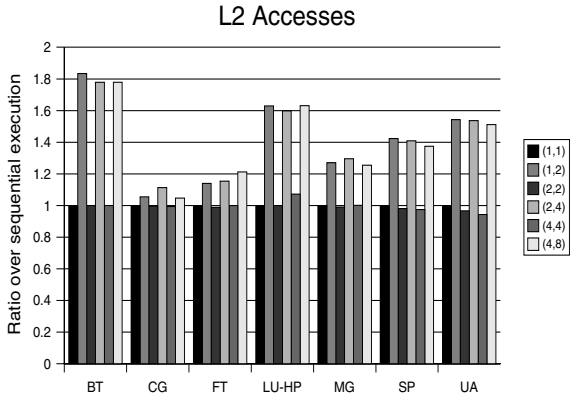

Figure 2. Normalized number of L2 cache accesses of the benchmarks.

In fact, in all cases, running two threads on a single processor resulted in an increase in the number of L2 accesses. BT, LU-HP and UA are the three benchmarks most prone to L1 cache contention.

Inter-thread interference does not seem to significantly affect the performance of the L3 cache (the results are omitted here due to space limitations, however the reader can refer to an extended version of this paper [3]). The notable exceptions are again BT and FT. In general, L3 misses do not seem closely correlated with either the number of threads used per processor or the number of processors used. The L3 cache is twice the size of the L2 cache and this alleviates, to some extent, the problem of inter-thread conflicts. Furthermore, since most of the memory accesses are filtered in the L1 and L2 caches, sharing the L3 cache does not significantly affect the performance of these benchmarks. In the three

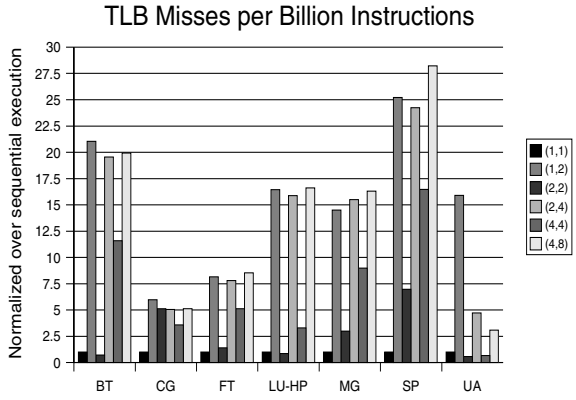

Figure 3. Normalized number of data TLB misses of the benchmarks per billion instructions executed.

benchmarks in which L2 suffers from inter-thread conflicts (BT, FT and UA), the L3 cache seems to exhibit a similar behavior, albeit at a much smaller scale for BT and UA. FT remains pathological and is a primary target for the multithreaded execution techniques we introduce later in this paper.

4.2 Data TLB Misses

The results observed for the number of data TLB misses per instruction show that intra-processor multithreading has a strongly adverse effect on the data TLB performance of SMTs (shown in Figure 3). The number of misses goes up by at least a factor of five when the configuration is changed from one thread to two threads on a single processor, and by factors of more than twenty in several cases. Overall, the rate of data TLB misses increases by an average of 925% from one thread to two threads per processor. The poor data TLB performance is explained in part by the fact that the data TLB of the Intel Hyperthreaded Pentium 4 is shared and relatively small (64 entries). Interestingly enough, although the processor uses a partitioned instruction TLB with 64 entries per thread, the data TLB has been chosen to be smaller and shared. It is often the case that co-executing threads work on different portions of the virtual address space, and therefore cannot share data TLB entries. This, in turn, results in an effective halving of the data TLB area per thread when both contexts of the SMT are activated.

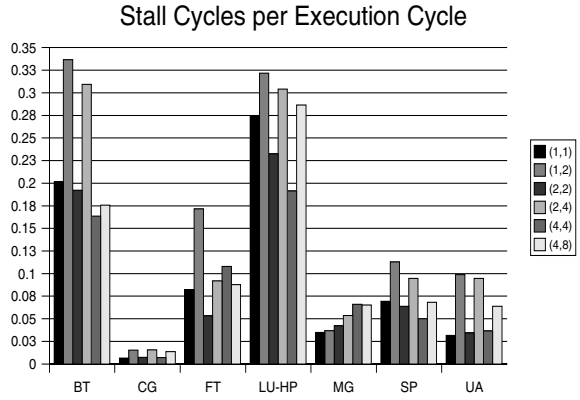

Figure 4. Number of stall cycles of the benchmarks normalized by the total number of cycles.

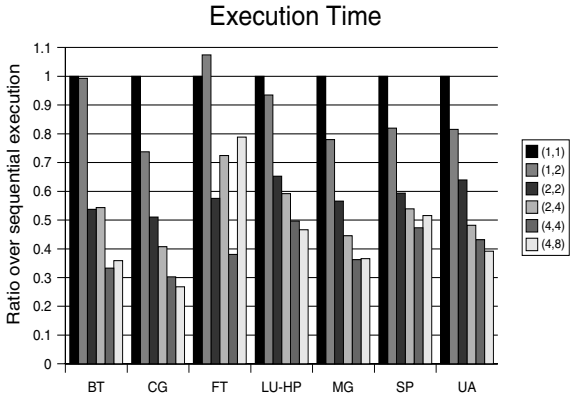

Figure 5. Normalized execution time of the benchmarks on the SMT multiprocessor.

4.3 Stall Cycles

SMT architectures share the vast majority of their resources, including functional units and all levels of the memory hierarchy. Stall cycle rate is a metric indicating how much time each application spends waiting, without making further progress, for either functional units or any level of the memory hierarchy to return a result. A relative change in stall cycles when using two threads instead of one thread per processor is an indirect indication of thread contention for all shared resources in the processor.

As shown in Figure 4, for nearly every benchmark and number of processors tested, the stall cycle rate (calculated over the total number of execution cycles) goes up dramatically when the configuration is changed from one to two threads per processor. There is an average increase in the number of stall cycles of 3.1 times when the second context of each SMT is used to run an additional application thread.

We have used an indirect experimental method to confirm the intuition that the lack of resource replication should be blamed for the suboptimal performance when two threads run simultaneously on each processor. We executed the same experiments on a simulated chip multiprocessor (CMP) architecture with 4 dual-core processors, using the Simics simulation environment [4]. In contrast to our real SMT processor, the CMP we modeled only shares the two outermost levels of the cache hierarchy and none of the functional units. This reduces the possible sources of stall cycles due to inter-thread interference, because on the CMP their number can be affected only by cross-thread interference in the L2 and L3 caches. On our simulated CMP machine, stall cycles were much more stable and relatively immune to changes in the number of threads per processor as well as to thread placement. The number of stall cycles only increased by an average of 3% on the CMP machine when the second hardware context was activated. These results provide evidence that the vast majority of stall cycles on the SMT machine were caused by the conflicting requirements of co-executing threads for internal processor resources.

4.4 Execution Time

Execution time is the ultimate performance metric. Although the results we obtained from hardware performance counters indicate several flaws of hyperthreading, it is important to observe whether these flaws translate into severe performance penalties. Figure 5 summarizes the normalized execution times of the benchmarks.

Not surprisingly, on the SMT architecture, execution time is always improved by running a fixed number of threads on as many different processors as there are threads, instead of co-executing two threads on each processor. This is because running threads on separate physical processors eliminates the effects of inter-thread interference on the shared resources. Interestingly, it is not always clear whether it is better to use one thread or two threads per processor. For example, in CG, LU-HP and UA, using the second thread is beneficial, whereas FT always suffers a slowdown when a second thread is used on each processor.

One more interesting phenomenon is that for a given application, it may be the case that neither one thread nor two threads per processor is always the most effec-

tive choice. In SP, for example, using two threads per processor is more effective in the 1 and 2-processor executions, but less effective for 4-processor executions.

When two threads run together on a single processor, there is an average speedup of merely 7% (taken across all benchmarks running on 1, 2 and 4 processors), compared to running with one thread per processor. The CG benchmark enjoys the best gains from two threads per processor with an average speedup of 25%. UA is a close second with 22%. There are two benchmarks which, on average, see a slowdown with the use of the second hyperthread, namely BT (4%) and FT (35%). In the rest of the paper we focus on these two benchmarks, and use them as motivating examples for introducing adaptive and more flexible multithreaded execution mechanisms to utilize SMTs in parallel programs.

5 Adaptive Thread Throttling

Based on our observations about the execution of parallel applications on SMT architectures, we implemented a means by which each application can adaptively choose the optimal number of threads to use per physical processor. The mechanism exploits the fact that under some circumstances two threads per processor will execute more effectively and other times only one will, due to contention. A similar adaptive execution mechanism was used in [20] to select the optimal number of threads and the best scheduler for each parallel loop, using code injected by an OpenMP compiler. Our implementation of this mechanism differs in that it is based on a compiler-independent runtime library which can be used via a preprocessor.

Our runtime library is called at the beginning and end of each parallel region so that a decision can be made for the optimal number of threads to use for this region. In scientific codes, parallel regions typically delimit phases of the code with different execution properties and performance characteristics. Controlling concurrency at the entry points of parallel regions is therefore an effective means to also control the performance of the code. Furthermore, since in most scientific applications parallel regions are executed many times across different iterations of outermost loops, the first few executions of each parallel section can be used to come to a conclusion on the optimal concurrency. In our experiments, we used three iterations for this purpose. We ignore the first iteration to account for cache warmup effects. Then the library tries both one thread per processor and two threads per processor, for one iteration each, and compares their execution times. Whichever number of threads results in the lower execution time is used in the future whenever this loop is encountered.

The major advantage of adaptive throttling is that it can achieve runtime performance optimization without modifications to the parallelization and execution strategies of applications. Adaptive thread control is an easy feature to implement in any thread management library. With proper engineering, it can be used transparently and effortlessly in any shared-memory programming model. One disadvantage of adaptive throttling is that it leaves idle hardware threads, which could be used for the purpose of accelerating the code running on the non-idle hardware threads. We explore this option via speculative runahead execution in Section 6. Another disadvantage is that it is applied effectively only in applications with a uniform iterative structure (e.g. iterative PDE solvers running over multiple time steps), but is not as effective in adaptive and irregular applications that do not have the same structure.

5.1 Results from Adaptive Thread Throttling

We tested our adaptive thread throttling mechanism, using the NAS Parallel Benchmarks, and the OpenMP codes MM5 and COBRA. MM5 [5] is a mesoscale weather prediction model. COBRA [1] is a matrix pseudospectrum computation code. We ran each of the benchmarks with one and two threads per processor on 1, 2, 3 and 4 processors, with the number of threads per processor fixed throughout the execution of each benchmark, to obtain a baseline for comparison against adaptive throttling. Following, we activated adaptive thread throttling and repeated the experiments.

The results, presented in Figure 6, show that adaptive thread control fares well compared to a static execution scheme selected from an oracle. When compared to the optimal static number of threads for each case, the adaptive mechanism is only 3.0% slower on average. In comparison, adaptation achieves a 10.7% average speedup over the worse static number of threads for each benchmark. The average speedup observed over all cases of executions with a fixed number of threads is 3.9%. A closer look at the results reveals that in 17 out of the total 36 experiments, adaptive throttling outperforms both static executions (with one thread and two threads per processor). Adaptive thread control consistently yields the best performance in COBRA on any number of processors and achieves a speedup improvement in several other cases as well.

Two applications for which adaptive throttling does not perform well are MG and FT. The main reason is

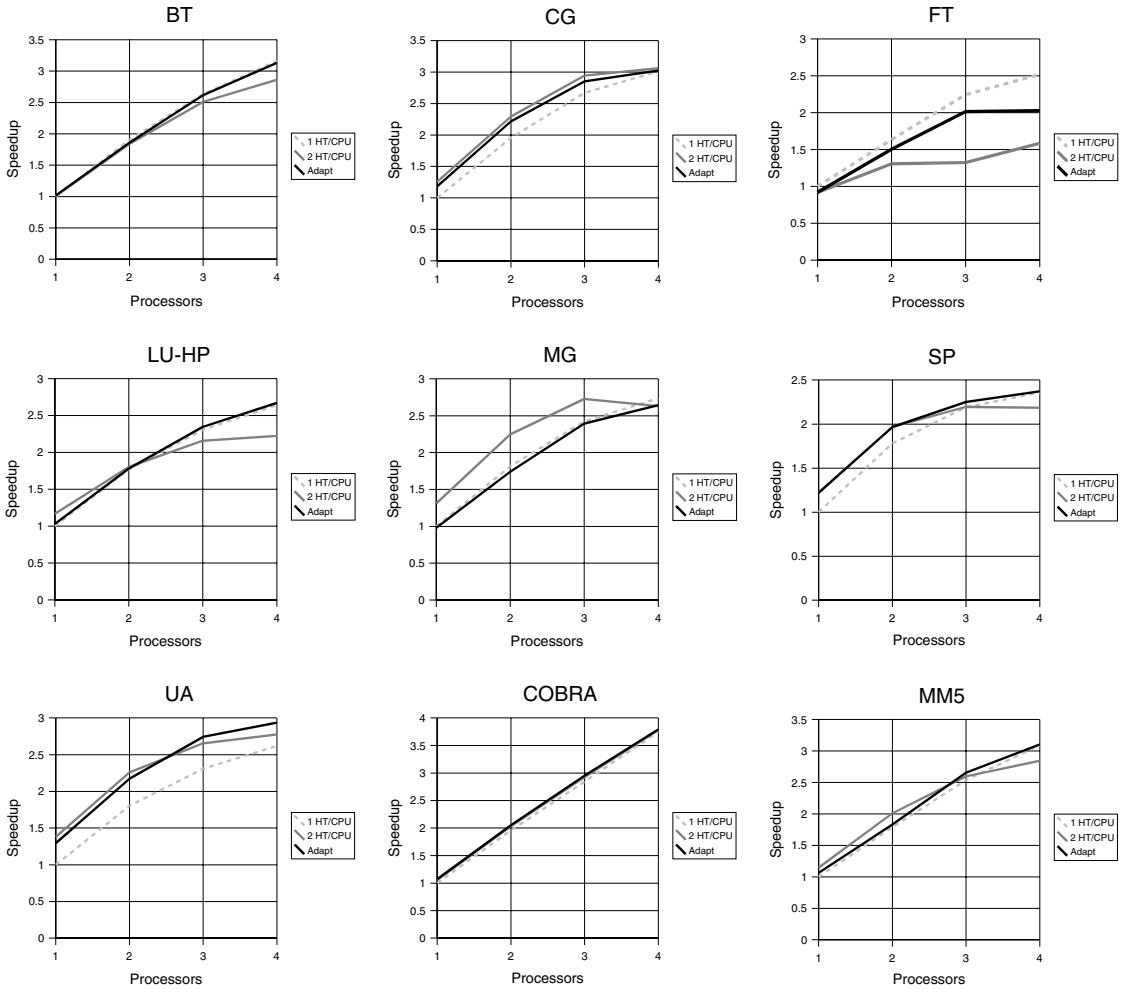

Figure 6. Relative performance of adaptive and static one/two threads per physical processor execution strategies. The execution times have been normalized with respect to the sequential execution time of each experiment.

that neither has a sufficient number of loops to amortize the startup costs of searching for the best number of threads. MG performs only 4 and FT only 6 iterations. With the 3 iteration initialization phase needed by adaptive throttling, the applications use the intelligently selected number of threads for only 1 and 3 iterations respectively.

One question that arises from these experiments is how the adaptive throttling mechanism performs against a static execution with an oracle which knows how many threads to use on a loop by loop basis, rather than in the entire program. To answer this question, we conducted additional experiments with BT, which suffers an average 4% slowdown when the second hardware context is used. BT and FT are the two benchmarks in which there seems to be no benefit from using the second hyperthread on each processor. We recorded the number of threads used for each section by the adaptive approach and hardcoded it in the application.

When the best number of threads for each parallel section is given from an oracle, BT enjoys a speedup of 1.8% from the occasional use of the second hyperthread per processor in a selected set of parallel loops. The adaptive throttling mechanism converges to the optimal number of threads for each parallel section, but yields an average slowdown of 1.1% compared to static execution with one hyperthread used per processor. This indicates that the adaptive mechanism performs comparably, but

still pays some measurable overhead to converge to the best degree of multithreading across the code.

Another drawback of adaptive throttling is related to potential side-effects in the memory performance of applications. Memory performance, both within and across loops, is often the target of extensive optimizations by programmers. However, inter-loop memory optimizations are based on the assumption of a constant number of threads used for the execution of all loops. Adaptive, per-loop selection of the number of threads may result in shuffling of data touched by each thread across loops, thus having an adverse effect on locality.

Such drawbacks necessitate the consideration of other options for utilizing simultaneous multithreading to improve performance.

6 Integrating TLP with Speculative Precomputation

The earlier discussion revealed that some programs are highly sensitive to contention for shared resources on SMT processors. The most characteristic example is FT, in which using the second thread on an SMT yields a slowdown of 35% on average. The main computational kernel of FT is composed of three FFTs (along the x, y and z dimensions), each of which walks two arrays (named x and $xout$ in the code) with very long strides (equal to 256×128 elements, or 32K in the tested problem size). The long strides cause an excessive amount of cache and TLB misses, as well as high contention for memory bandwidth. In memory-intensive codes such as FT, simultaneous multithreading intensifies the effect of the aforementioned bottlenecks.

Speculative precomputation (SPR for short) is a memory latency reduction technique proposed for SMTs and CMPs [2]. SPR uses either microarchitectural or software support to employ an otherwise idle thread for precomputing addresses of *critical* memory accesses and prefetching the data touched from these accesses into the cache, anticipating that the data will arrive early enough to be used from a sibling computation thread without suffering cache misses. Critical accesses are those responsible for a large number of cache misses in the outermost levels (typically L2 and L3) of the cache.

SPR is a resource-conserving approach to leverage multithreading on a single chip, since the speculative thread works in synergy and to the benefit of the non-speculative thread. It has been used successfully to speed up sequential codes dominated by pointer chasing [2, 18] but no emphasis has been placed on using SPR in scientific applications, partially because memory access patterns in these codes are often highly predictable. We argue that SPR, using a spare hardware context, has value in scientific codes with long streams of long-strided memory accesses as well, not necessarily because of better prediction abilities, but because of being less resource-consuming than in-place sequential prefetching, without sacrificing timeliness. As such, SPR can also be an effective alternative to TLP for loops in which TLP causes excessive contention for shared resources. To evaluate this hypothesis, we applied SPR in FT in loops in which the performance proved to be highly problematic when the loops were parallelized using multiple threads within an SMT.

To use SPR, we had to devise a method that would allow assisted runahead execution within an SMT, while still enabling loop-level parallel execution across SMTs. We used the nested parallel execution features of OpenMP. We organized the code of each one-dimensional FFT in two parallel sections, one executing the main FFT loops with multiple threads bound to different SMTs and the other executing speculative precomputation threads also bound to different SMTs, in the contexts left unused by the main computation threads. Effectively, the program uses two levels of heterogeneous parallelism to merge precomputation with regular multithreaded code.

We have identified critical memory accesses using an execution-driven cache simulator derived from Valgrind. We ran a stripped-down binary of FT with only two iterations of the PDE solver. We identified that more than 95% of the L2 and L3 cache misses in FT occur in three routines of the code (`cffts1`, `cffts2` and `cffts3`, corresponding to FFT's in the x, y and z direction respectively), and on just two elements, x and $xout$, corresponding to the input and output vectors of each FFT. Note that x and $xout$ point to the same vector in `cffts2` and `cffts3` but to different vectors in `cffts1`. Interestingly enough, we found that the dominating misses are in most applications the same, regardless of the problem size, the data input and the number of processors/threads used to execute them. This indicates that a profile-driven approach for identifying critical loads is quite effective. More details of our precomputation code generation scheme are given in [3, 19].

Figure 7 summarizes the performance results of the three execution strategies. `Adapt+prec` corresponds to experiments in which parallel execution across processors (using selectively one or two hyperthreads per processor during initialization and lhs update) is combined with SPR within processors for the three main FFT routines. This hybrid execution method is com-

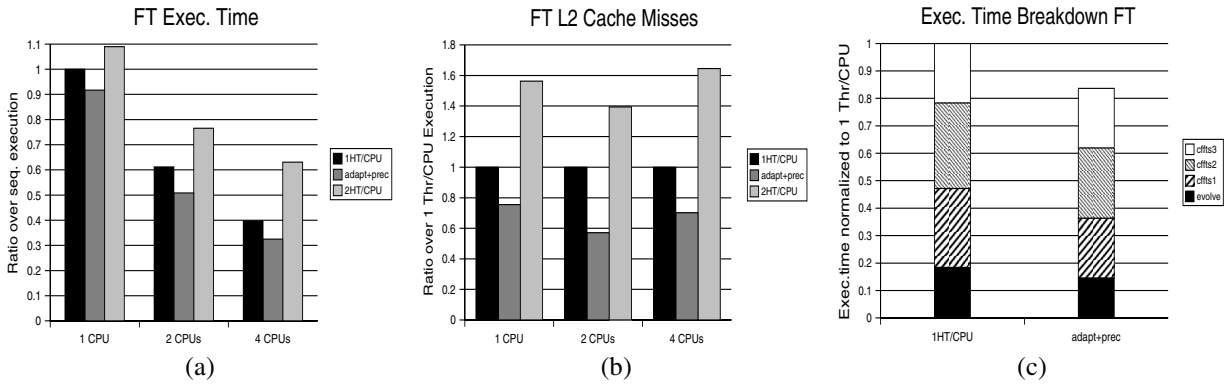

Figure 7. Impact of using selective speculative precomputation in conjunction with thread-level parallelism in NAS FT.

pared against thread-level parallel execution with 1 or 2 hyperthreads per processor. In the thread-level parallel execution we have activated the software prefetching functionality offered by the Intel compiler. Moreover, the hardware prefetcher of Intel processors was active throughout all the experiments.

In the three memory-bound loops, our SPR code eliminates 25–43% of the L2 cache misses in FT (Figure 7(b)). This translates to 9%–22% faster execution than the parallel one with one hyperthread per processor (Figure 7(a)), as opposed to an average slowdown of 35% incurred in executions in which two hyperthreads split the parallel computation on each processor. The speedup over the sequential execution time of FT is improved from 2.5 to 3.0 on 4 processors. It is important to note that this speedup arises mostly but not solely from the use of SPR. Figure 7(c) gives a breakdown of execution time between the four primary subroutines of FT, obtained during execution on 4 processors. The subroutine `evolve` executes the lhs update code of the PDE solver. Two of the three FFT routines (`cffts1` and `cffts2` along the x, and y dimensions) benefit significantly from prefetching and enjoy speedups of 18% and 25% respectively, while the third FFT routine (`cffts3` along the z direction) obtains little benefit. On the other hand, the routine `evolve` benefits from regular parallelization with two hyperthreads per processor, with a speedup of almost 20%.

In conclusion, mixed-mode multithreaded execution yields superior performance in a code in which thread-level parallelization within an SMT suffers from conflicts due to simultaneous multithreading. Several scientific applications exhibit such behavior, therefore the proposed adaptive multithreaded execution mechanisms are expected to play an important role in scaling these applications on SMT-based multiprocessors.

7 Conclusions

This paper has illustrated the limitations of simultaneous multithreading processors when running scientific applications, using physical experimentation on a real multi-SMT system and several performance metrics obtained from hardware event counters.

Motivated by these limitations, we proposed the use of two software mechanisms for improving the performance of parallel codes running on multiple SMT processors. First, with adaptive thread throttling, the number of threads used on each SMT is regulated per parallel section, so that code phases that do not benefit from SMT execution are adaptively sequentialized. Secondly, we proposed a mechanism to merge speculative precomputation with conventional thread-level parallel execution in OpenMP and we have shown that this mechanism can be used to significantly improve performance in parallel codes with long streams of long-strided memory accesses. Combined with selective thread throttling, our hybrid execution mechanism achieved a 9%–22% performance improvement from the use of the second hyperthread on each processor in FT, a code which proved to be particularly difficult to scale otherwise within an SMT.

Our study has shown that significant performance gains can be achieved through the integration of multiple forms of multithreading on SMT-based multiprocessors. In the future we intend to derive new static and dynamic techniques to automate the use of polymorphic multithreading in parallel programs.

Acknowledgements

This work is supported by an NSF ITR grant (ACI-0312980), an NSF CAREER award (CCF-0346867) and the College of William and Mary. The authors would like to thank Xavier Martorell of DAC-UPC in Barcelona and IBM Research, for recommending the nested OpenMP execution model for merging speculative precomputation with thread-level parallel execution and other useful discussions on this work.

References

[1] C. Bekas and E. Gallopoulos. Cobra: Parallel Path Following for Computing the Matrix Pseudospectrum. *Parallel Computing*, 27(8):1879–1896, July 2001.

[2] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proc. of the 28th Annual International Symposium on Computer Architecture (ISCA–2001)*, pages 14–25, Goteborg, Sweden, July 2001.

[3] M. Curtis-Maury, T. Wang, C. Antonopoulos, and D. Nikolopoulos. Integrating Multiple Forms of Multithreaded Execution on SMT Processors: A Quantitative Study with Scientific Workloads. Technical report, Department of Computer Science, College of William and Mary, http://www.cs.wm.edu/ mfcurt/downloads/FormsTR.pdf, April 2005.

[4] F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. In *Proc. of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[5] G. A. Grell, J. Dudhia, and D. R. Stauffer. A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5). NCAR Technical Note NCAR/TN-398 + STR, National Center For Atmospheric Research (NCAR), June 1995.

[6] K. Ibrahim and G. Byrd. Extending OpenMP to Support Slipstream Execution Mode. In *Proc. of the 17th International Parallel and Distributed Processing Symposium (IPDPS'2003)*, Nice, France, April 2003.

[7] Intel Inc. Intel VTune Performance Analyser. http://www.intel.com/software/products/vtune, 2003.

[8] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report nas-99-011, NASA Ames Research Center, October 1999.

[9] R. Kalla, B. Sinharoy, and J. Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, March 2004.

[10] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-Core Chip Multiprocessing. In *Proc. of the 37th International Symposium on Microarchitecture (MICRO-37)*, pages 195–206, Portland, OR, December 2004.

[11] J. Lo, J. Emer, H. Levy, R. Stamm, D., and S. Eggers. Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, 15(3):322–353, August 1997.

[12] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, Version 2.5 edition, May 2004.

[13] A. Roth and G. Sohi. A Quantitative Framework for Quantitative Pre-Execution Thread Selection. In *Proc. of the 35th IEEE/ACM Annual International Symposium on Microarchitecture (MICRO–35)*, Istanbul, Turkey, November 2002.

[14] A. Snavely and D. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Processor. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IX)*, pages 234–244, Cambridge, Massachusetts, November 2000.

[15] X. Tian, Y. Chen, M. Girkar, S. Ge, R. Lienhart, and S. Shah. Exploring the Use of Hyper-Threading Technology for Multimedia Applications with Intel OpenMP Compiler. In *Proc. of the 17th International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[16] N. Tuck and D. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proc. of the 2003 International Conference on Parallel Architectures and Compilation Techniques (PACT'2003)*, New Orleans, LA, September 2003.

[17] D. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *In Proceedings of the 22nd Intenational Symposium on Computer Architecture*, pages 392 – 403, June 1995.

[18] H. Wang, P. Wang, R. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen. Speculative Precomputation: Exploring the Use of Multithreading for Latency. *Intel Technology Journal*, 6(1), February 2002.

[19] T. Wang, C. Antonopoulos, and D. Nikolopoulos. smtSPRINTS: Software Precomputation with Intelligent Streaming for Resource-Constrained SMTs. In *Proc. of the 11th European Conference on Parallel Computing (EuroPar'2005)*, Lisbon, Portugal, August 2005.

[20] Y. Zhang, M. Burcea, V. Cheng, R. Ho, V. Cheng, and M. Voss. An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs. In *Proc. of PDCS-2004: International Conference on Parallel and Distributed Computing Systems*, San Francisco, CA, September 2004.