An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors

Matthew Curtis-Maury, Xiaoning Ding,
Christos D. Antonopoulos and Dimitrios S. Nikolopoulos

Department of Computer Science
The College of William and Mary
McGlothlin–Street Hall. Williamsburg, VA 23187–8795

Abstract. Multiprocessors based on simultaneous multithreaded (SMT) or multicore (CMP) processors are continuing to gain a significant share in both high-performance and mainstream computing markets. In this paper we evaluate the performance of OpenMP applications on these two parallel architectures. We use detailed hardware metrics to identify architectural bottlenecks. We find that the high level of resource sharing in SMTs results in performance complications, should more than 1 thread be assigned on a single physical processor. CMPs, on the other hand, are an attractive alternative. Our results show that the exploitation of the multiple processor cores on each chip results in significant performance benefits. We evaluate an adaptive, run-time mechanism which provides limited performance improvements on SMTs, however the inherent bottlenecks remain difficult to overcome. We conclude that out-of-the-box OpenMP code scales better on CMPs than SMTs. To maximize the efficiency of OpenMP on SMTs, new capabilities are required by the runtime environment and/or the programming interface.

1 Introduction

As a shared-memory programming paradigm, OpenMP is suitable for parallelizing applications on simultaneous multithreaded (SMT) [17] and multicore (CMP) [16] processors. These processors appear to dominate both the high-end and mainstream computing markets. Products such as Intel's Hyperthreaded Pentium IV are already widely used for desktop and server computing, with similar products being marketed or in late stages of development by other vendors. At the same time, high-end, future microprocessors encompass aggressive multithreading and multicore technologies to form powerful computational building blocks for the next generation of supercomputers. All three vendors selected by the DARPA HPCS program (IBM, Cray and Sun) have adopted multithreaded and multicore processor designs, combined with different technological innovations such as streaming processor cores and proximity communication [5, 6, 15].

With the advent of multithreaded and multicore multiprocessors, a thorough evaluation of OpenMP on such architectures is a timely and necessary effort. In this paper we evaluate a comprehensive set of OpenMP codes, including complete parallel benchmarks and real-world applications, on both a real multi-SMT system, composed of Intel's Hyperthreaded processors, and on a simulated multiprocessor with CMP processors. For the latter architectural class we use complete system simulation to factor in any operating system effects. Our evaluation uses detailed performance measurements and information from hardware performance counters to pinpoint architectural bottlenecks of SMT/CMP processors that hinder the scalability of OpenMP, as well as areas in which OpenMP implementations can be improved to better support execution on SMT/CMP processors.

We observe that the extensive resource sharing in SMTs often hinders scalability, should threads co-executing on the same physical processor have conflicting resource requirements. The significantly lower degree of resource sharing in CMPs, on the other hand, allows applications to effectively exploit the multiple execution cores of each physical processor. We quantitatively evaluate the effects of resource sharing on the L2 miss rate, the number of stall cycles and the number of data TLB misses. We then evaluate the effectiveness of a run-time mechanism that transparently determines and uses the optimal number of threads on each SMT processor. This technique yields measurable, though limited performance improvement. Despite its assistance, the architectural bottlenecks of SMTs do not allow OpenMP applications to efficiently exploit the additional execution contexts of SMT processors.

The rest of the paper is organized as follows: In section 2 we outline related work. In section 3 we evaluate the execution of OpenMP codes on SMT- and CMP-based multiprocessors and pinpoint architectural bottlenecks using a variety of performance metrics. Section 4 evaluates a simple, yet effective mechanism that automatically determines and exploits the optimal number of execution contexts on SMT-based multiprocessors. In section 5 we outline some implications of the proliferation of hybrid, SMT- and CMP-based multiprocessors for OpenMP. Finally, section 6 concludes the paper.

2 Related Work

Earlier research efforts have ported and evaluated OpenMP on specific processor designs, including heterogeneous chip multiprocessors [14], slipstream processors [9] (a form of 2-way chip multiprocessors in which the second core is used for speculative runahead execution) and Cyclops, a fine-grain multithreaded processor architecture introduced by IBM [1]. Our evaluation focuses on commodity processors, with organizations spanning the design space between simultaneous multithreading and chip multiprocessors and a few execution contexts. Although not at the high end of the design space of supercomputing architectures, such processors are becoming commonplace and are natural building blocks for larger multiprocessors. A recent study of OpenMP loop scheduling policies on multiprocessors with Intel's Hyperthreaded processors indicated the need for adaptation of both the degree of concurrency and the loop scheduling algorithms when OpenMP applications are executed on simultaneous multithreading architectures, because of different forms of interferences between threads [18]. Our evaluation corroborates these results and provides deeper insight on the architectural reasons due to which adaptivity is an effective method for improving the performance of OpenMP programs on SMT processors.

3 Experimental Evaluation and Analysis

3.1 Hardware and Software Environment and Configuration

In order to ascertain the effects of the characteristics of modern processor architectures on the execution of OpenMP applications, we have considered two types of multiprocessors which are becoming more and more popular in today's computing environment, namely multiprocessors based on either SMTs or CMPs. SMTs incorporate minimal additional hardware in order to allow multiple co-executing threads to exploit potentially idle processor resources. The threads usually share a single set of resources such as execution units, caches and the TLB. CMPs on the other hand integrate multiple independent processor cores on a chip. The cores do, however, share one or more outer levels of the cache hierarchy, as well as the interface to external devices.

We used a real, 4-way server based on Hyperthreaded (HT) Intel processors as a representative SMT-based multiprocessor. Intel HT processors are a low-end / low-cost implementation of simultaneous multithreading. Each processor offers 2 execution contexts which share execution units, all levels of the cache, and a common TLB. The experiments targeted at the CMP-based multiprocessors have been carried out on a simulated 4-way system. The simulated CMP processors integrate 2 cores per processor. They are configured almost identically to the real Intel HTs, apart from the L1 cache and TLB which are private, per core on the CMP and shared between execution contexts on the SMT. Note that using private L1 caches and TLBs favors CMPs by providing more effective cache and TLB space to each thread and reducing contention. Therefore, our experimental setup seems to favor CMPs. Note however, that we are evaluating a CMP with in-order issue cores, which are much simpler than the out-of-order execution engines of our real SMT platform. Furthermore, the multicore organization of our simulated CMP enables a chip layout with private L1 caches at a nominal increase in die area [13]. For these reasons, the simulated CMP platform can still be considered as roughly equivalent (in terms of resources) to our real SMT platform. We used the Simics [7] simulation platform to conduct complete system simulations, including system calls and operating system overhead. Table 1 describes the configuration of the two systems in further detail.

	Processors	L1 Cache	L2 Cache	L3 Cache	TLB	Main Mem.
SMT	4 x Intel P4 Xeon, 1.4 GHz Hyperthreaded x 2 Execution Contexts per Processor	8K Data, 12K Trace (Instr.), Shared	256K Unified, Shared	512K Unified, Shared	64 Entries Data, 128 Entries Instr., Shared	1GB
CMP	4 Processors x 2 P4 Cores per Processor	2x8K Data, 2x12K Trace (Instr.) Private per Core	256K Unified, Shared	512K Unified, Shared	2x64 Entries Data, 2x64 Entries Instr., Private per Core	1GB

Table 1. Configuration of the SMT- and CMP-based multiprocessors used throughout the experimental evaluation.

We evaluated the relative performance of OpenMP workloads on the two target architectures, using 7 OpenMP applications from the NAS Parallel Benchmarks suite (version 3.1) [11]. We executed the class A problem size of the benchmarks, since it is a large enough size to yield realistic results. At the same time, it is the largest problem class that allows the working sets of all applications to fit entirely in the available main memory of 1GB.

We executed all the benchmarks on the SMT with 1, 2, 4, and 8 threads. The main goal of this experiment set was to evaluate the effects of executing 1 or 2 threads on the 2 execution contexts of each processor. We thus ran our experiments under six different thread placements: i) 1 thread, ii) 2 threads bound on 2 different physical processors, iii) 2 threads bound on the 2 contexts of 1 processor, iv) 4 threads bound on 4 processors, v) 4 threads paired on the execution contexts of 2 processors and vi) 8 threads paired on 4 processors. Each thread is pinned on a specific execution context of a specific processor using the Linux `sched_setaffinity` system call. The applications were executed using Intel VTune [10] performance analyzer. We recorded both the execution time and a multitude of additional performance metrics attained from the hardware performance counters available in the processor. Such metrics provide insight into the interaction of applications with the hardware, thus they are a valuable tool for understanding the observed application performance.

The same experiments have been repeated on the simulated CMP-based multiprocessor. Full system simulation with Simics introduces an average 7000-fold slowdown in the execution time of applications, compared with the execution on a real machine. We simulated the same application binaries, using the same data sets, however we reduced the number of iterations[1] we ran on the simulator in order to limit the execution time to reasonable levels. More specifically, we executed only 3 of the outermost iterations of each benchmark, discarding the results from the first iteration in order to eliminate transient effects due to cache warmup. The simulator directly provides similar, detailed performance information as Vtune.

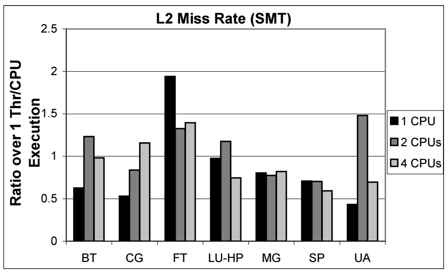
All experiments were performed on a dedicated machine in order to rule out data perturbations due to interactions with third-party applications and services. The operating system on both the real and the simulated system was Linux 2.4.25.

3.2 Experimental Results

We evaluated the relative performance of the benchmarks on the real SMT-based and the simulated CMP-based multiprocessors when 1 or 2 threads are activated per physical processor, using the different binding schemes described in section 3.1. We monitored a multitude of direct (wall clock time, number of instructions, number of L2 and L3 references and misses, number of stall cycles, number of data TLB misses, number of bus transactions) and derived (CPI, L2 and L3 miss rates) performance metrics. Due to space limitations we only present and discuss the results for L2 miss rates, stall cycles, data TLB misses and execution time.

The results for the L2 miss rate evaluation are depicted in Figure 1. The reported values are for 2 threads per processor and have been normalized with respect to the single-thread per processor execution of each benchmark on the specific architecture and number of processors. This way, the graphs emphasize the effects of using a second

[1] All the NAS applications we used are iterative. The computational routines are enclosed in an external, sequential loop.



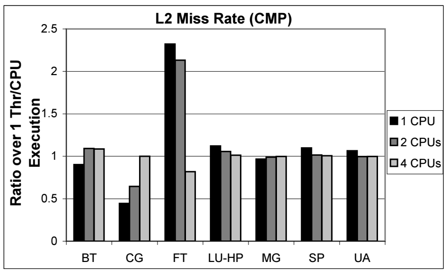


Fig. 1. Normalized L2 miss rates of the benchmarks on the SMT and CMP multiprocessor (left and right diagrams respectively). The corresponding 1 thread/processor miss rates on each architecture and number of processors have been used as references for the normalization.

thread per processor. The relative L2 cache performance of applications when 1 and 2 threads are executed on each physical processor depends highly on the specifi c characteristics of the application. If the working sets of both threads do not fi t in the L2 cache, there is an increase in the L2 miss rate, since cross-thread cache-line eviction results in more misses. If, on the other hand, the 2 threads executing on the same processor share data, then each of them will probably benefi t from data already fetched to the cache by the other thread.

In most cases, executing 2 threads per processor on the SMT system proved benefi - cial for L2 cache performance. On average, thread pairing resulted in 1.05 times lower miss rates in comparison with the single-thread per processor execution. An application in which thread cross-eviction appears is FT. The FT threads have large working sets that can not entirely fi t into any level of the cache hierarchy. Moreover, the degree of data sharing between threads co-executing on the same processor is low. As a result, miss rates increase signifi cantly if both execution contexts of each processor are activated. Another interesting pattern can be observed in CG. Although the exploitation of the second hyperthread of each processor results in a signifi cant reduction in miss rate in the single processor experiments, as more physical processors are added the trend is reversed. CG has a high degree of data sharing between the threads. If few threads are active, the benefi ts of the shared cache are evident. However, as more physical processors are added, inter-processor data sharing results in a large number of cache-line invalidations, which eventually outweigh the benefi t of intra-processor data sharing.

On the CMP-based multiprocessor the L2 cache miss rate generally appears to be uncorrelated to the exploitation of 1 or 2 execution cores per physical processor. Although the L2 is shared between both cores, the private, per core L1 caches function as a buffer that prevents many memory accesses from reaching the second level of the cache. In fact, the use of a second thread on SMTs results in an increase in the number of L2 cache accesses, due to the inter-thread interference in the L1 cache. More specifically, the number of L2 accesses always increases, by 1.42 times on average, when the second execution context is activated on each physical processor. The private L1 caches in CMPs alleviate this problem. The number of L2 accesses is reduced by an average factor of 1.37 when the second core - and its private L1 cache - are activated on each

CPU. The behavioral patterns observed for CG and FT on the SMT-based multiprocessor are repeated on the CMP as well.

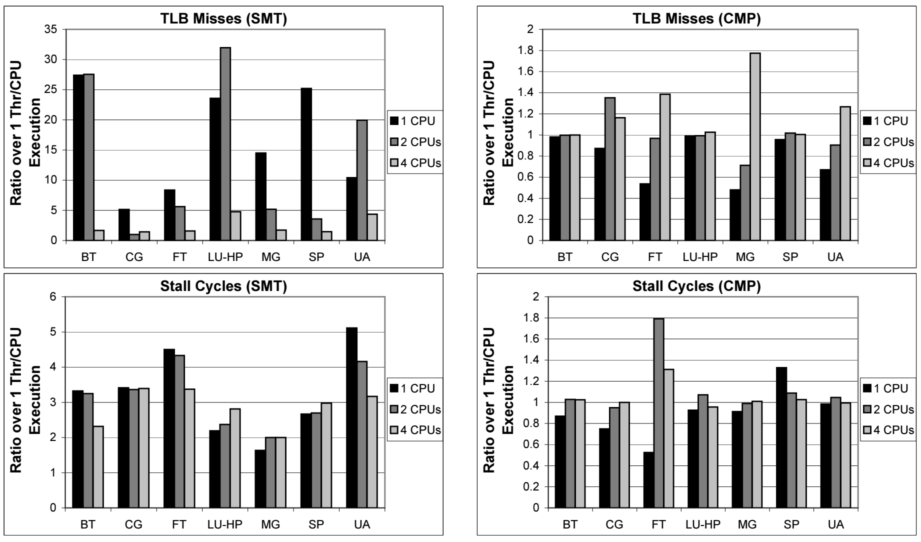


Fig. 2. Normalized number of TLB misses (top diagrams) and stall cycles (bottom diagrams) of the benchmarks on the SMT and CMP multiprocessor (left and right diagrams respectively). The corresponding 1 thread/processor stalls and TLB misses on each architecture and number of processors have been used as references for the normalization.

Figure 2 depicts the normalized number of stall cycles and TLB misses. Once again, the reported values have been normalized with respect to the corresponding single-thread per processor execution of each benchmark on the specifi c architecture and number of processors. The results indicate that using the second execution context of the SMT processors has a signifi cant effect on the number of TLB misses. Their number suffers an up to 27-fold increase when we move from binding schemes that assign 1 thread per processor to those that assign 1 thread per execution context. On average, TLB misses increase by 10.78 times. The 2 threads on each processor often work on different areas of the virtual address space, thus being unable to share TLB entries. Furthermore, the Intel SMT processor has a surprisingly small data TLB (64 entries), which can not achieve a good coverage of the virtual address space of the benchmarks we executed. As a result, the effective per thread size of the shared TLB is reduced drastically when both execution contexts of each processor are activated. The CMP processor provides private TLBs for each core. As a consequence, the number of TLB misses is much more stable than on the SMT system. In fact, the execution of 1 or 2 threads per processor has, on average, no effect on the number of TLB misses.

The behavior in terms of stall cycles also varied signifi cantly between the two architectures. On SMT processors the number of stall cycles represents the cumulative effect of both cycles spent waiting for data to be fetched from any level of the memory hierarchy and cycles during which execution was stalled because of conflicting resource requirements of the threads executing on the different execution contexts of the processor. On CMPs, co-executing threads share only the 2 outer levels of the cache and the interface to external devices, thus the second factor does not contribute to the total number of stall cycles. For all benchmarks executed on the SMT, the number of stall cycles increased – on average by 3.1 times – when the configuration was changed from 1 to 2 threads per processor. The corresponding average overhead on the CMP is a mere 1.03. This is a safe indication that the vast majority of stall cycles on the SMT can be attributed to conflicting requirements of co-executing threads for internal processor resources.

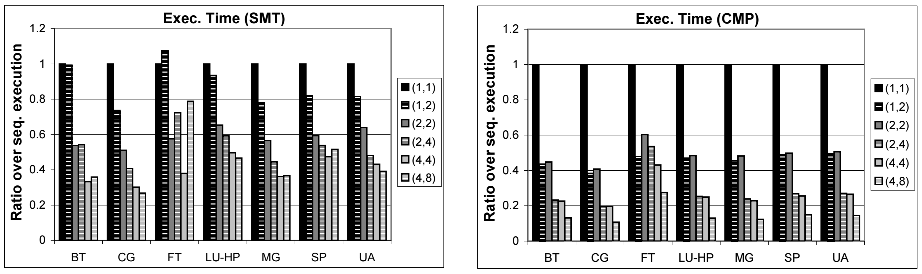


Fig. 3. Normalized execution time of the benchmarks on the SMT and CMP multiprocessor (left and right diagram respectively). The single-threaded (sequential) execution time on each architecture is used as a reference for the normalization.

Finally, Figure 3 depicts the results from the execution time of applications on the two target multiprocessor architectures. This time, the reported values have been normalized with respect to the sequential (single-threaded, single-processor) execution time of applications on each architecture. 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 number of threads used for the application execution.

All 7 benchmarks scale well on both the SMT and the CMP as more physical processors are made available to the application. This indicates that potential performance problems under some binding schemes can not be attributed to the scalability characteristics of the benchmarks. In fact for the 2-threaded BT and CG execution on the CMP the speedups are superlinear, due to the availability of cumulatively larger L1 cache and TLB when more than 1 threads are used.

Given a specific number of threads, execution times on the SMT multiprocessor are always lower if the threads are spread across as many physical processors as possible, instead of being placed on both execution contexts of each processor. Moreover, in 7 out of 21 experiments the activation of the second execution context, given a specific number of physical SMT processors, resulted in a reduction of the observed application performance. It should also be pointed out that, even for a given application, it is not always clear whether the exploitation of all execution contexts of each processor is the optimal strategy or not. In the case of SP, for example, exploiting 2 execution contexts per processor is optimal when 1 and 2 processors are available, however it results in performance penalties when all 4 processors are used.

The results are totally different on the CMP-based multiprocessor. In 8 out of 14 cases placing a given number of threads on the cores of as few processors as possible yields higher performance than spreading them across processors. Moreover, the activation of the second core always resulted in performance improvements. The replication of execution units, L1 caches and TLBs on the CMPs allows threads to execute more effectively, without the limitations posed by resource sharing on SMTs. The reduction in resource conflicts due to hardware replication often allows the benefits of inter-processor cache sharing to be reflected in a reduction in execution time.

4 Adaptive Selection of the Optimal Number of Execution Contexts for OpenMP on SMTs

The selection of the optimal number of execution contexts for the execution of each OpenMP application is not trivial on SMT-based SMPs. We thus experiment with a performance-driven, adaptive mechanism which dynamically activates and deactivates the additional execution contexts on SMT processors to automatically approximate the execution time of the best static selection of execution contexts per processor. We used a simpler mechanism than the exhaustive search proposed in [18], which avoids modifications to the OpenMP compiler and runtime. Our mechanism identifies whether the use of the second execution context of each processor is beneficial for performance and adapts the number of threads used for the execution of each parallel region. The algorithm introduced in [18] also targets identification of the best loop scheduling policy.

Our method is based on the annotation of the beginning and end of parallel regions with calls to our runtime. The calls can be inserted automatically, by a simple preprocessor. Alternatively, run-time linking techniques such as dynamic interposition can be used to intercept the calls issued to the native OpenMP runtime at the boundaries of parallel regions and apply dynamic adaptation even to unmodified application binaries.

We slightly modify the semantics of the `OMP_NUM_THREADS` environment variable, using it as a suggestion for the number of processors to be used instead of the number of threads. Moreover, we add a new environment variable (`OMP_SMT`). If `OMP_SMT` is defined to be 1 or 2, the application always uses 1 and 2 execution contexts per physical processor respectively. If its value is 0, or the variable is not defined, adaptive execution is activated. In this case, each kernel thread is first bound on a specific execution context upon program startup. On the second and third time each parallel region is encountered, our runtime executes it using 1 and 2 execution contexts per processor and monitors execution time. After the third execution of each region, a decision is made using the timing results from the two test executions. Upon additional invocations of the parallel region, the runtime automatically adjusts the number of threads according to the decision. The first execution of each parallel region is not monitored, in order to avoid any interference in the decision process due to cache warmup effects. The runtime makes decisions independently for each parallel region. The execution of most applications proceeds in phases, with different execution characteristics for each phase. The boundaries of parallel regions often indicate phase changes. Thus, varying the number of threads at the boundaries of parallel regions offers context sensitive adaptation[2].

We evaluated the performance of our adaptive mechanism using the NAS Parallel Benchmarks along with two other OpenMP codes: MM5 [8], a mesoscale weather prediction model, and COBRA [4], a matrix pseudospectrum computation code. We ran each of the benchmarks statically with 1 and 2 threads per processor on 1, 2, 3, and 4 processors. We then executed each benchmark using the adaptive strategy. Even in the experiments using a static number of threads, threads are bound on specific execution contexts in order to avoid unfairly penalizing performance due to suboptimal thread placement decisions of the Linux scheduler. The results are depicted in Figure 4.

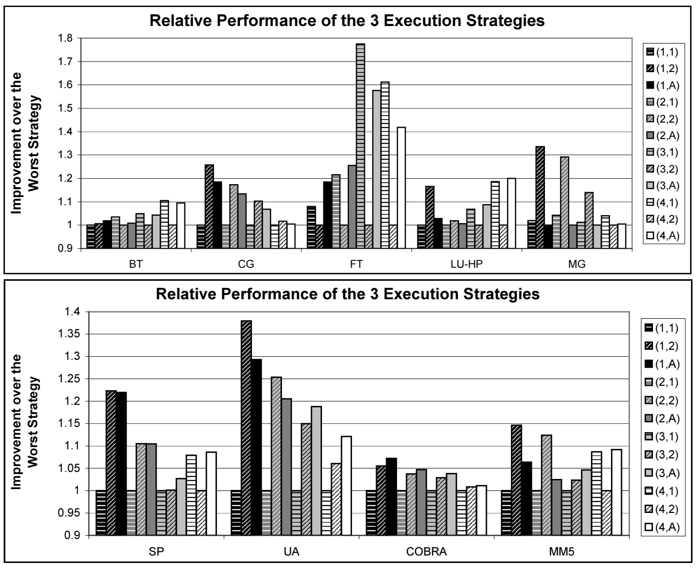


Fig. 4. Relative performance of adaptive, 1 and 2 threads per physical processor execution strategies. The execution times have been normalized with respect to the execution time of the worst strategy for each experiment.

Compared with the optimal static number of threads for each case, our approach was only 3.0% slower on average. At the same time, it achieved a 10.7% average speedup over the worse static number of threads for each (benchmark, number of processors) combination. The average overall speedup observed over all static configurations was 3.9%. In 17 out of the total 36 experiments the adaptive mechanism even provided a performance improvement over both static strategies for selecting the number threads.

[2] In fact loop boundaries can offer a better approximation of application phases. OpenMP specifications however, prohibit varying the number of active threads inside a parallel region, thus adaptive mechanisms like ours can not be used to make decisions at a loop-level resolution.

This can be attributed to the flexibility of the adaptive mechanism and its ability to decide the optimal number of threads independently for each parallel region.

The adaptive technique did not perform well for MG. MG performs only 4 outermost iterations. Given that 3 iterations are needed for the initialization and decision phases, MG executes in adaptive mode for only 1 iteration. However, it does not take many iterations for the adaptive execution to compensate for the overhead of the monitoring phase. CG, for example, performs just 15 iterations and the adaptive strategy is only slightly inferior than the best static strategy.

The performance benefi ts attained by our simple mechanism are lower than those attained by the combined adaptation of the number of threads and loop schedules in [18]. They indicate that dynamic adaptation can provide some speedup on SMT-based multiprocessors, however the inherent architectural bottlenecks of contemporary SMTs hinder the effi cient exploitation of the additional execution contexts.

5 Implications for OpenMP

Our study indicates that although scaling OpenMP on CMPs can be effortless, scaling on SMTs is hindered by the effects of extensive resource sharing. We argue that it is still worthwhile to consider performance optimizations for OpenMP on SMTs. In addition to the current Intel family of SMT processors, multicore architectures with SMT cores are also gaining popularity, because such designs often achieve the best balance between energy, die area and performance [12]. In our view, optimizing OpenMP for SMTs entails both additional support from the runtime environment and possible extensions to the programming interface. Clearly, the runtime environment should differentiate between threads running on the same SMT and threads running across SMTs. This can be achieved in a number of ways: For example, a new `SCHEDULE` clause would allow the loop scheduler to assign iterations between SMTs using a given policy and then use an SMT-aware policy for splitting iterations between threads on the same SMT. Alternatively, OpenMP extensions for thread groups [3] can be exploited, so that threads within the same SMT processor belong to the same group and use their own scheduling and local synchronization mechanisms. Note that using groups in this case does not necessarily imply the use of nested parallelism. SMT-aware programs may utilize just a single level of parallelism but use different policies for executing threads within SMTs. In fact, current SMTs do not allow the exploitation of parallelism with granularity much fi ner than what can be exploited by conventional multiprocessors [2]. If no extensions to the OpenMP interface are desired, then more intelligence should be embedded in the runtime environment, to dynamically identify threads sharing an SMT and differentiate its internal thread management policies. Although such an expectation is not unreasonable for regular iterative scientifi c applications, it is diffi cult to achieve the same level of runtime sophistication for irregular applications.

Regarding portability (of both code and performance), one of the most important problems for implementing an SMT-aware version of OpenMP is thread binding to processors and execution contexts within processors. Clearly, if the programmer wishes to exploit a single level of parallelism in a non-malleable program, the issue of binding is irrelevant. If however the programmer wishes for any reason to utilize SMTs for an alternative multithreaded execution strategy (e.g. for nested parallelism, or for slipstream execution), then it is necessary to specify the placement of threads on processors. Although the OpenMP community has proposed extensions to handle similar cases (e.g. via an `ONTO` clause), exposing architecture internals in the programming interface is undesirable in OpenMP. Therefore, new solutions for improving the execution of OpenMP programs on SMTs in an autonomic manner are desirable.

6 Conclusions

In this paper we evaluated the performance of OpenMP applications on SMT- and CMP-based multiprocessors. We found that the execution of multiple threads on each processor is more efficient and predictable on CMPs than it is on SMTs due to the higher degree of resource isolation, which results in fewer conflicts between threads co-executing on the same processor. Although adaptive run-time techniques can improve the performance of OpenMP applications on SMTs, inherent architectural bottlenecks hinder the efficient exploitation of these processors.

Our analysis indicated that the interference between co-executing threads in the shared levels of the cache or the shared TLB may prove a determining factor for performance. Driven by this observation, we intend to evaluate run- and compile-time techniques for TLB partitioning on SMTs and cache partitioning on both SMT and CMP architectures. The forthcoming proliferation of processors which combine simultaneous multithreading and chip multiprocessing, such as the IBM Power5, and their use as basic building blocks of multiprocessors will certainly generate a multitude of challenging software optimization problems for system software and application developers.

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.

References

1. G. Almasi, E. Ayguade, C. Cascaval, J. Castanos, J. Labarta, F. Martinez, X. Martorell, and J. Moreira. Evaluation of OpenMP for the Cyclops Multithreaded Architecture. In *Proc. of the 2003 International Workshop on OpenMP Applications and Tools (WOMPAT'2003)*, pages 147–159, Toronto, Canada, June 2003. LNCS Vol. 2716.
2. C. D. Antonopoulos, X. Ding, A. Chernikov, F. Blagojevic, D. S. Nikolopoulos, and N. Chrisochoides. Multigrain Parallel Delaunay Mesh Generation: Challenges and Opportunities for Multithreaded Architectures. In *Proc. of the 19th ACM International Conference on Supercomputing (ICS'2005)*, Cambridge, MA, U.S.A., June 2005.
3. E. Ayguadé, M. Gonzàlez, X. Martorell, J. Oliver, J. Labarta, and N. Navarro. NANOSCompiler: A Research Platform for OpenMP Extensions. In *Proc. of the First European Workshop on OpenMP*, pages 27–31, Lund, Sweden, October 1999.
4. C. Bekas and E. Gallopoulos. Cobra: Parallel path following for computing the matrix pseudospectrum. *Parallel Computing*, 27(8):1879–1896, July 2001.

5. W. Dally, P. Hanrahan, M. Erez, T. Knight, F. Laboté, J. Ahn, N. Jayasena, U. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with Streams. In *Proc. of the IEEE/ACM Supercomputing'2003: High Performance Networking and Computing Conference (SC'2003)*, Phoenix, AZ, November 2003.
6. K. Ebcioglu, V. Saraswat, and V. Sarkar. The IBM PERCS Project and New Opportunities for Compiler-Driven Performance via a New Programming Model. Compiler-Driven Performance Workshop (CASCON'2004), October 2004.
7. 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.
8. 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.
9. 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.
10. Intel Inc. Intel VTune Performance Analyser. http://www.intel.com/software/products/vtune, 2003.
11. 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.
12. R. Kalla, B. Sinharoy, and J. Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, March 2004.
13. 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.
14. F. Liu and V. Chaudhary. Extending OpenMP for Heterogeneous Chip Multiprocessors. In *Proc. of the 2003 International Conference on Parallel Processing*, pages 161–168, Kaohsiung, Taiwan, October 2003.
15. J. Mitchell. Sun's Vision for Secure Solutions for the Government. National Laboratories Information Technology Summit, June 2004.
16. K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, October 1996.
17. D. M. 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. 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.