Implementation of Fine-Grained Cache Monitoring for Improved SMT Scheduling

Joshua L. Kihm and Daniel A. Connors
University of Colorado at Boulder
Department of Electrical and Computer Engineering
UCB 425, Boulder, CO, 80309
{kihm, dconnors}@colorado.edu

Abstract

Simultaneous Multithreading (SMT) is emerging as an effective microarchitecture model to increase the utilization of resources in modern super-scalar processors. However, co-scheduled threads often aggressively compete for certain limited resources, among the most important of which is space in the cache hierarchy. Rather than require future systems to have more cache resources, performance-aware scheduling techniques can be used to adapt thread scheduling decisions and minimize this inter-thread contention for cache resources. Although many processors currently have the ability to summarize the activity in each cache level, systems that monitor and collect detailed information about cache access behaviors can enable scheduling algorithms to fully exploit multithreaded cache workload characteristics in different cache regions. This paper explores the design of a novel fine-grained hardware monitoring system in an SMT-based processor that enables improved system scheduling and throughput.

1. Introduction

By concurrently executing instructions from multiple active threads, simultaneously multithreaded (SMT) processors avoid limitations on instruction-level parallelism and exploit thread-level parallelism. The SMT microarchitecture model more fully utilizes available system resources and thus is emerging as the leading, cost-effective method to sustain performance of commercial and scientific workloads. Unfortunately, as the gap between memory and processor core performance widens, cache penalties will become more damaging to overall system performance, even in SMT designs. For instance, studies [3] have demonstrated that performance of an SMT processor is directly tied to the efficiency of the lower cache levels. This effect will only grow as the penalties for main memory and lower-level cache accesses grow in terms of clock cycles. Likewise, SMT designs must also address the penalties due to increased cache miss rates caused by inter-thread contention for cache resources.

Fortunately, the effectiveness of multithreaded systems is not strictly controlled by hardware constraints. For example, experimental studies ([8][10]) have shown that in a multithreaded architecture, the selection of threads to schedule together has a major impact on overall system performance. Since threads are competing for finite processor resources, inter-thread contention for resources can invalidate the advantages of multithreading and reduce its effectiveness. Thus, the key to effective scheduling in a multithreaded system is determining which threads can be scheduled together as to minimize the inter-thread conflict rate as to maximize overall performance. Although static techniques have been developed to pre-select thread pairs can try to minimize thread interference, adaptive techniques have greater potential to overcome program execution variance and generate more symbiotic system schedules. In order to make effective run-time scheduling decisions, thread schedulers must be able to monitor the behavior of individual threads and predict future execution behavior in order to estimate the interference between individual threads.

To date, current microprocessors only include performance monitoring counters that can only summarize cache access behavior. Yet, the full potential of multithreaded systems can only be fully realized by using operating system scheduling algorithms that can effectively monitor details of cache behavior and track inter-thread cache contention. Likewise, since program behavior can vary drastically over time and programs execute in phases [7]. Therefore, an effective thread scheduler must be able to recognize and predict program behavior changes. It has been demonstrated [11] that SMT performance is dictated by not only which threads are scheduled together, but more importantly by the phase characteristics of each executing thread. As such, it is critical to develop dynamic thread scheduling techniques that can adapt to program phase changes. Ideally, the thread scheduler should predict future behavior as well as monitor program phase to maximize program performance during the next scheduling period.

This paper investigates architectural support for *performance-aware* SMT thread scheduling techniques that exploit the memory access behaviors and patterns of individual threads. We extend existing operating system scheduling techniques to predict inter-thread interaction before each scheduling interval, rather than treating thread behavior as monolithic. The hardware methods for tracking and predicting thread behavior are relatively simple by design, yet they can directly aid the operating system job scheduler in reducing inter-thread interference by an average of 10%, resulting in an average 7% performance improvement. Details of job scheduling techniques based on the presented hardware monitoring structures can be found in [6]. The remainder of the paper is organized as follows: Section 2 discusses the motivation for this work, Section 3 illustrates the proposed hardware structures, Section 4 describes the experimental methodology and results, and Section 5 concludes the paper.

2. Motivation

Job scheduling can alter the collective system workload by selecting threads based on their individual characteristics. A number of architecture resources and their potential bottlenecks to overall performance can be examined to determine the combined SMT execution efficiency of multiple threads. For example, limitations in number of integer units, floating-point units, cache ports, fetch bandwidth, and cache space are just a few of the processor components that can be overrun by job mixes that include too many simultaneous resource requests made from two or more threads during the same interval. Thus, with the effectiveness of cache memories being of primary importance to reducing the frequency of long latency memory requests, it is essential that job scheduling for an SMT involve matching the memory request characteristics of paired threads. To do this, effective ways of observing and communicating exploitable runtime memory behavior to the operating system are needed.

Figure 1 illustrates temporal cache use behavior of *188.ammp* and *253.perlbmk* from the SPEC CPU2000 benchmark suite. The x-axis of each graph is the time measured in samples of one million clock cycles. Along the y-axis is cache position (grouping of cache sets into 32 regions or super sets). A *super set* is a grouping of several contiguous cache sets. The color of each super set indicates cache activity, dark represents very low usage and light represents very high. These cache maps demonstrate behavior that changes not only temporally, but also spatially with some regions hosting the majority of overall cache activity. Each application has distinct patterns (temporal and spatial) in each of the different levels of cache, all of which is meaningful to understanding how to create the best job mix for an SMT system.

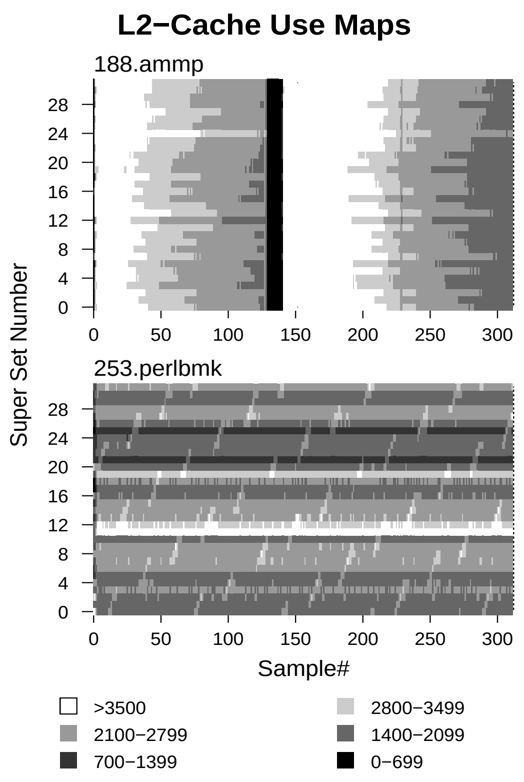

Figure 1. Level-2 cache activity characteristics for *188.ammp* and *253.perlbmk*.

The consequence of scheduling two jobs with high activity at the same super set positions during the same scheduling interval is directly correlated to the inter-thread kick outs (ITKO) incurred during paired execution. An ITKO is a line of the cache from thread A being pushed out of the cache because thread B requires that cache line. If thread A is not finished with that data, it will have to recall it from lower in the memory hierarchy and incur a performance penalty that it would not have taken in a single threaded environment. Under evaluation, we observed that inter-thread conflict misses account (on average) for more than 30% of the overall data misses, and 25% of the instruction misses, on a randomly selected job mix of SPEC 2000 applications. Although thread thrashing can be mitigated by increasing cache associativity, it does so at the cost of increased hardware complexity and cache access time. As such, minimizing the amount of inter-thread conflict using performance-aware multithreaded schedulers proves to be an effective

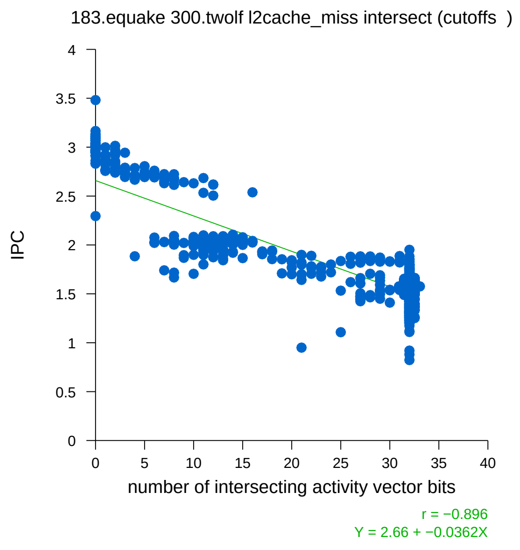

Figure 2. IPC correlation to activity confluence of two paired threads.

Benchmark Mix	% Diff.
164.gzip, 164.gzip, 181.mcf, 183.equake	0.0%
164.gzip, 164.gzip, 188.ammp, 300.twolf	12.0%
164.gzip, 177.mesa, 181.mcf, 183.equake	0.0%
164.gzip, 177.mesa, 183.equake, 183.equake	0.0%
164.gzip, 197.parser, 253.perlbmk, 300.twolf	44.4%
177.mesa, 177.mesa, 197.parser, 300.twolf	11.1%
177.mesa, 181.mcf, 253.perlbmk, 256.bzip2	0.0%
177.mesa, 188.ammp, 253.perlbmk, 300.twolf	59.5%
177.mesa, 197.parser, 197.parser, 256.bzip2	96.2%
181.mcf, 181.mcf, 256.bzip2, 256.bzip2	0.0%
181.mcf, 183.equake, 253.perlbmk, 300.twolf	4.0%
181.mcf, 253.perlbmk, 253.perlbmk, 256.bzip2	0.0%
183.equake, 188.ammp, 188.ammp, 256.bzip2	11.1%
188.ammp, 188.ammp, 197.parser, 197.parser	96.2%
188.ammp, 300.twolf, 300.twolf, 300.twolf	8.0%
197.parser, 197.parser, 253.perlbmk, 256.bzip2	0.0%
Average	22.5%

Table 1. Percentage of scheduling decisions that are different using 128-super-set activity vector versus raw miss count for various benchmark mixes

and most likely necessary technique.

Figure 2 shows the consequence to the number of executed instructions per cycle (IPC) on an SMT machine when monitoring the number of groups of cache sets that had high activity for both threads in an SMT system executing two threads. Thirty-two super sets were considered, and the forty-five pairings of nine SPEC CPU2000 benchmarks (including two instances of the same benchmark) were evaluated to derive this data. Overall, the results show that IPC is strongly correlated with the confluence of cache set activity of the combined threads. When two threads have zero-overlap (below an average threshold) in their cache activity, IPC is generally 1.5 higher than when two threads overlap across all 32 cache super sets.

The decisions of operating systems schedulers can be driven by fine-grained run-time information. To test this concept, sixteen random job mixes of SPEC CPU2000 benchmarks, consisting of four applications per job, were examined using an SMT architecture simulator. At each scheduling interval, the operating system could choose one of three threads to schedule with a resident (fourth) thread. Scheduling decision were based on the level two cache miss counter that tracks overall cache misses and compared against the scheduling decisions based on monitoring a 128-super-set activity vector of the cache accesses. The results are summarized in table 1. Although there is a great deal of variance between the individual tests, 22.5% of the decisions were different on average between the two methods. In the tests where very few or none of the decisions differ typically involve benchmarks with very high (such as *181.mcf*) or very low levels of memory demand (such as *164.gzip*). In these cases, the activity is consistently very high or low across the cache, so little new information is derived from the activity vector. Unfortunately, it is extremely difficult to test which is the better decision because it is challenging to replicate the exact initial conditions of each decision. However, coupled with the correlation data from Figure 5 the case is strong that the inter-thread interference is more closely correlated to activity vector score than the overall activity counts, so decisions based on the activity vector are probably better.

3. Hardware support

Modern microprocessors have support for a number of different performance counters for tracking events in both the microarchitecture pipeline and memory system [2][1]. For example, the Intel Pentium4 and Itanium2 and the IBM Power4 processors each support a large number of possible counters of which a few can be monitored at any given time (4 in Intel CPU's and 8 in the Power4). Additionally, the Itanium has the ability to sample data cache miss addresses using Event Address Registers and can be configured to ignore misses that return under a certain threshold number of cycles (which theoretically extends the capability to lower level caches if a proper threshold is chosen). Such support can be useful in validating microarchitecture behavior, performance analysis, and performance tuning. However, these counters have limitations in their flexibility to support adaptable run-time optimization and operating system scheduling. Although existing processor counters facilitate coarse-grained analysis, collecting aggregate counter values of an architecture component over an interval of time can be generally lacking in detail about the exact location of exploitable workload characteristics.

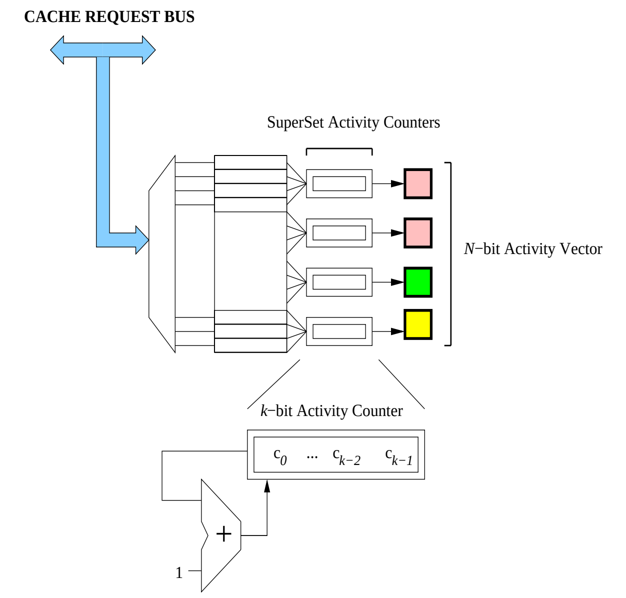

Figure 3. Activity counters and vectors

Figure 3 illustrates the hardware structures needed to support the proposed performance-aware scheduling. The address bits from the cache bus are snooped and used to index the superset activity counters and increment the appropriate counter. Similar hardware was proposed in [9]. The activity counters are then each reduced to a few bits and combined into a N-dimensional *Activity Vector* (where N is the number of super sets). In the experimental results section, we evaluate the effects of the number of super sets, the precision of the activity vector, and how the activity vector thresholds between activity vector thresholds are chosen. Generally, the design theme for not transporting a large number of activity counters (one for each cache set, or even one for each cache super set) to the operating system for analysis in scheduling decisions is to reduce the processing overhead of the performance-aware scheduler.

Although overall a thread's cache behavior varies over time, it demonstrates periodicity and hence is very predictable. In [5] it was shown that simple models could be used to predict cache activity within a small margin with 89% to 95% accuracy. This is important as scheduling decisions are made in order to minimize interference in a

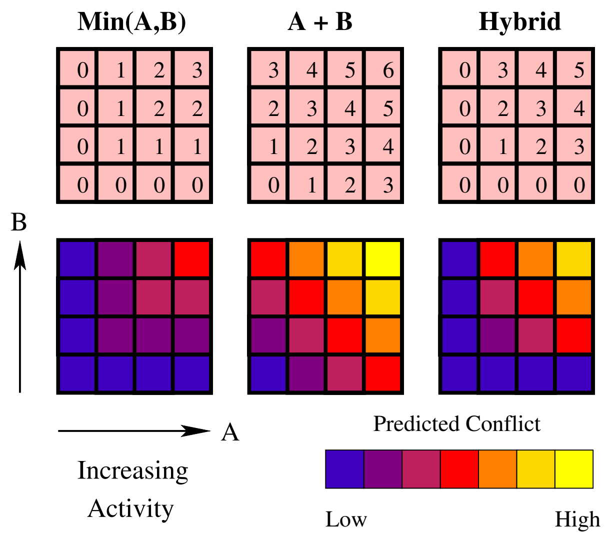

Figure 4. Various methods of quantifying predicted intersection from cache usage

scheduling period before that period occurs.

4. Experimental Results

4.1. Experimental Methodology

Our simulated environment was designed to simulate a common SMT processor, the Intel Pentium 4 model 2 with Hyperthreading [4]. Our model has separate level one data and instruction caches with an access time of 4 cycles (the Pentium 4 has a two cycle access time for integers and 9 cycles for floating point data and unspecified access time to the trace cache). Level two is a unified 512KB cache with an access time of 7 cycles. Finally the Level three cache is 2MB with an access time of 14 cycles. Main memory access is modeled as 300 cycles. Our simulator is a modified version of the SMTSIM simulator extended to support our thread scheduling algorithms. Tests were performed over one billion simulated cycles on all possible static combinations of nine SPEC2000 benchmarks.

Although it is possible to simply save the counters from each super set and use that raw data in the scheduling algorithm, it is probably unnecessary. A simplified version of the data is sufficient because the noisy nature of the data makes high precision meaningless. Additionally, only a summary of the fine-grained usage is needed to predict what the level of interference will be. The predicted usage for each thread in each super set is given a score of between zero for low usage and some pre-defined maximum. The predicted use in each super set for each thread is scored against the predicted use in that super set of the other threads in potential. The overall goodness of a potential grouping is the sum of

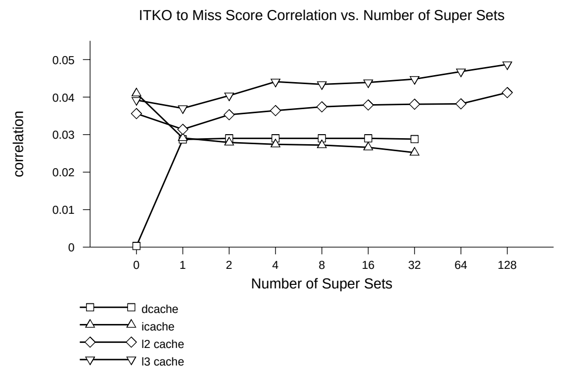

Figure 5. Effect of the number of super sets.

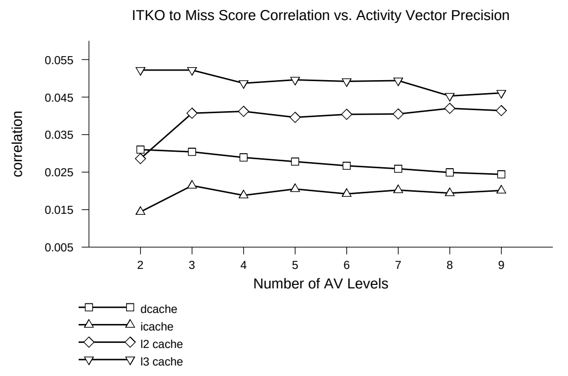

Figure 6. Effect of activity vector precision.

the scores from all super sets. The question is then how to best predict interference from predicted use. Using a simple sum of the scores is weak because it gives the same number for two heavily imbalanced scores (where there is unlikely have high interference) as is does for two evenly weighted scores (which would presumably have high interference). The weakness of using the minimum of the scores is that if one thread has light usage, it will determine the interference score. If the other thread also has light usage, this is not a concern, but the same score will be given even if the second thread has very high usage (and will thus probably cause higher interference).

The interference score used in this study is a hybrid of the sum and minimum. The score given is the sum of the threads' usage scores minus one unless one score is zero. If one score is a zero the interference score is made zero. This special case is made because if one thread has very low usage, it is unlikely to interfere with other scheduled threads. We feel that this is a good compromise between the other two methods considered although ongoing research indicates that something more heavily weighted toward the minimum score may be optimal. The scoring systems are illustrated in Figure 4, each being developed with a four by four matrix that indicates the prediction of conflict at the particular superset position given thread **A** and thread **B** quantized thread activity (0-3). The first row of the figure has the three interference prediction matrices for the three scoring methods considered. Three methods are considered: *Min*, *Sum*, and *Hybrid*. The second row is a color-coded representation of the same data, where brighter colors indicate higher predicted interference. The overall interference score is the sum of the interference scores from each super set.

4.2. Cache Monitor Granularity

Many modern processors include the ability to monitor cache activity at a course-grained level in real time, which could be used by operating systems in scheduling decisions. However, because demand varies across the cache, having finer grained information allows better characterization of cache behavior. Figure 5 illustrates this concept. The correlation between the score given to the intersection and total number of inter-thread kickouts is given for various numbers of super sets. The data for zero super sets is the correlation between the total number of misses in the cache and the number of ITKO and demonstrates the effects of counter quantization and the scoring scheme. The strong upward trend in the lower level caches is the important feature to note because these are the caches that have the greatest effect on overall performance. A miss which requires a main memory access is equivalent to dozens of misses in a higher-level cache.

4.3. Activity Vector Precision

An experiment similar to that performed for the number of super sets was performed on the number of usage levels and the results are shown in Figure 6. An interesting result is that that correlation is strongest for very few usage levels and correlation then flattens out or even goes down with finer granularity. Part of this effect is due to the simple method of determining the intersection score from the predicted usage, but the overall trend is most likely real. One weakness of the scoring system is that very heavy usage in both threads in linked to heavy interference, when in reality high use can only be achieved if there is little interference. The results indicate that only a few bits for each super set counter can be used to make effective scheduling decisions.

4.4. Determination of Activity Vector Cutoff

If the counter data is going to be quantized to one of a small number of values, some decision must be made as to

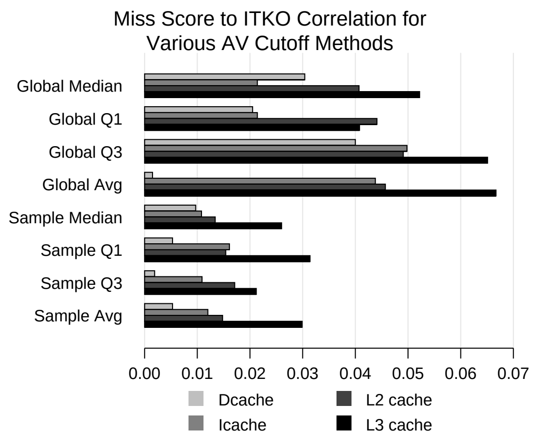

Figure 7. Effect of the activity vector cutoff method.

where to differentiate between those values. If only two values are available, only one decision point is needed. The question is where to put that decision point. Figure 7 summarizes the correlation between miss scores and ITKO for each cache and a variety of decision points on a two-level scoring system. The first set of tests are based on global statistics. All of the data was consolidated across all super sets, samples, and tests. The median, the first and third quartiles, and the arithmetic mean across all of this data were tested as the cutoff value and the average correlation was compared. The strongest correlations are with the third quartile used as the cutoff. This means that very high numbers of misses tend to best indicate a higher ITKO.

Another set of data that was tested used local statistics. Median, quartiles, and average were calculated for each activity vector for each sample. For example, when the median was used, exactly one half of the values in each activity vector were high and one half were low. This required much less calculation as the data did not have to be consolidated across all of the test and samples, but the correlation values were significantly lower. For tests with a higher number of activity vector values, the cutoff between values was linearly scaled from the two-level test. For instance a four-level test would have cutoffs at half the two-layer cutoff, the two-layer cutoff, and one and a half times the two-layer cutoff. Further analysis has to be done on where the ideal cutoff values are and how those can best be translated into expected interference values.

5. Conclusions

The advantages of multithreading extend only to the point at which threads begin to interfere and adversely impact system performance. The challenge in scheduling in a multithreaded environment is choosing threads in such a way that minimizes inter-thread interference. Fine-grained cache information is an excellent tool in choosing threads such that inter-thread interference in the memory hierarchy is minimized. In this paper, we have investigated to what granularity this information should be obtained and it's respective utility in guiding thread scheduling decisions.

6. Acknowledgments

The authors would like to thank John Liu, Tipp Moseley, Vijay Janapa Reddi, and Joshua Stone for their help in revising this paper. We also thank the anonymous reviewers for their helpful insights and suggestions.

References

[1] K. W. Cameron. *Empirical and Statistical Application Modeling Using On-chip Performance Monitors*. PhD thesis, Louisiana State University, Aug. 2000.

[2] L. DeRose. *Hardware Performance Monitor (HPM) Toolkit*. IBM Research Advanced Computing Technology Center, Version 2.5.1 edition, June 2003.

[3] S. Hily and A. Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report PI-1086, IRISA, 1997.

[4] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*. Santa Clara, CA, 2004.

[5] J. Kihm, A. Janiszewski, and D. Connors. Predictable fine-grained cache behavior for enhanced simultaneous multithreading (smt) scheduling. In *Proceedings of International Conference on Computing, Communications and Control Technologies*, 2004.

[6] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural support for enhanced smt scheduling. In *Proceedings of the 13th Annual Parallel Architectures and Compilation Techniques*, September 2004.

[7] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 17th annual international conference on Supercomputing*. ACM Press, 2003.

[8] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[9] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *The 9th International Symposium on High-Performance Computer Architecture*, Santa Clara, CA, 2002.

[10] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 54–58, 2000.

[11] M. VanBeisbrouk, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.