16th International Conference on Parallel Architecture and Compilation Techniques

Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler

Alexandra Fedorova
Simon Fraser University
fedorova@cs.sfu.ca

Margo Seltzer
Harvard University
margo@eecs.harvard.edu

Michael D. Smith
Harvard University
smith@eecs.harvard.edu

Abstract

We describe a new operating system scheduling algorithm that improves performance isolation on chip multiprocessors (CMP). Poor performance isolation occurs when an application's performance is determined by the behaviour of its co-runners, i.e., other applications simultaneously running with it. This performance dependency is caused by unfair, co-runner-dependent cache allocation on CMPs. Poor performance isolation interferes with the operating system's control over priority enforcement and hinders QoS provisioning. Previous solutions required modifications to the hardware. We present a new software solution. Our cache-fair algorithm ensures that the application runs as quickly as it would under fair cache allocation, regardless of how the cache is actually allocated. If the thread executes fewer instructions per cycle than it would under fair cache allocation, the scheduler increases that thread's CPU timeslice. This way, the thread's overall performance does not suffer because it is allowed to use the CPU longer. We describe our implementation of the algorithm in Solaris™ 10, and show that it significantly improves performance isolation for SPEC CPU, SPEC JBB and TPC-C.

1. Introduction

Applications running on chip multiprocessors (CMP) [21] suffer from poor performance isolation [8,9,17]. Poor performance isolation is a phenomenon where an application's performance is determined by the behaviour of its co-runners. Such performance dependency is due to inherently unfair, co-runner-dependent allocation of shared caches on CMPs. On CMPs, cache allocation is determined largely by the co-running threads' relative cache demands; fairness is not considered. A thread "demands" a cache allocation by generating a cache miss. The cache miss is satisfied after evicting an existing line from the cache. The evicted line may belong to *any* thread, not necessarily the thread responsible for the cache miss. Consequently, the thread responsible for the miss may affect its co-runners' cache allocations and, as a result, their performance. Accordingly, an application will run more slowly with high-miss-rate co-runners than with low-miss rate co-runners.

Poor performance isolation causes problems. One problem is OS scheduler's weakened control over priority enforcement. It is difficult for the scheduler to ensure that a high-priority thread makes consistently greater forward progress than a low priority-thread on a CMP processor, because the high-priority thread's performance could be arbitrarily decreased by a high-miss-rate co-runner.

Poor performance isolation complicates per-CPU-hour billing in shared computing facilities [2]. If an application runs slowly in a given CPU-hour only due to the misfortune of having a high-miss-rate co-runner, billing that application for the full hour is unfair. Existing systems have no means of detecting or preventing this.

Poor performance isolation hinders QoS provisioning. QoS is provisioned via a reservation of resources, such as a fraction of CPU cycles, for a customer's application. Poor performance isolation makes resource reservations less effective, since an application could be slowed down unpredictably by a high-miss-rate co-runner despite having dedicated resources.

To demonstrate the extent of poor performance isolation on CMPs, we draw upon data on *co-runner-dependent performance variability,* i.e., the difference between the application's running time with co-runner A and co-runner B [8,9,17]. (We use co-runner dependent performance variability as a metric for performance isolation: high variability implies poor performance isolation and vice versa). Previous work has shown that an application may take up to 65% longer to complete when it runs with a high-miss-rate co-runner than with a low-miss-rate co-runner [9]. Such dramatic slowdowns were attributed to significant increases in the second-level cache miss rates (up to 4x) experienced with a high-miss-rate co-

1089-795X/07 $25.00 © 2007 IEEE
DOI 10.1109/PACT.2007.18

IEEE computer society

runner, as opposed to a low-miss-rate co-runner.

Previous work addressed performance isolation in hardware, via cache partitioning [8,17,24,25]. While cache partitioning ensures fair cache allocation, it increases the cost and complexity of the hardware, has limited flexibility and long time-to-market. Our software solution avoids these shortcomings.

Our solution is a new operating system scheduling algorithm, the *cache-fair* algorithm. This algorithm reduces co-runner-dependent variability in an application's performance by ensuring that the application always runs as quickly as it would under fair cache allocation, *regardless* of how the cache is actually allocated. The cache-fair algorithm accomplishes that objective by regulating threads' CPU timeslices. A thread's CPU timeslice, as well as its IPC, determines its overall performance. (The IPC determines *how quickly* the thread executes instructions on CPU, while the timeslice determines *how much time* the thread gets to run on CPU.) Co-runner-dependent cache allocation creates co-runner-dependent variability in IPC and, hence, co-runner-dependent variability in overall performance. Since the OS cannot control the variability in the IPC (because it cannot control cache allocation), the cache-fair algorithm instead offsets the variability in the IPC by adjusting the CPU timeslice. The scheduler monitors the thread's IPC, and if it detects that the thread's actual IPC is lower than its IPC under fair cache allocation (we call this the *fair IPC*), it increases the thread's CPU timeslice. Likewise, if the thread's IPC is above its fair IPC, the scheduler decreases that thread's CPU timeslice. In this fashion, the scheduler compensates for the effects of unfair cache allocation on *overall* performance without requiring changes to the hardware.

While performance isolation has been addressed in the past in the context of shared physical memory [6], addressing it in the context of shared caches is more difficult. The allocation of physical memory is directly controlled by the operating system, whereas cache allocation is not. Since the cache-fair scheduler cannot *enforce* fair cache allocation, it must *compensate* when the allocation is unfair. To provide the right amount of compensation, the scheduler needs to determine the extent to which a thread's actual IPC differs from its fair IPC. Unfortunately, the fair IPC is not trivial to obtain: it cannot be measured, because one cannot simply "try" running a thread with a fairly allocated cache portion. To determine the fair IPC in our scheduler, we designed a new, low-overhead, heuristic cache model.

We implemented the cache-fair algorithm in a commercial operating system, Solaris 10, and showed (using a simulated CMP) that it significantly improves performance isolation for workloads ranging from SPEC CPU2000 to SPEC JBB and TPC-C. Co-runner dependent performance variability was reduced from as much as 28% to under 4% for all the benchmarks. Performance overhead generated by the algorithm was negligible (<1%).

We compared the effectiveness of the cache-fair scheduler and of cache partitioning (an alternative hardware solution) and found that the cache-fair scheduler reduces co-runner-dependent performance variability to a greater degree than cache partitioning. The cache-fair algorithm accounts for *secondary* performance effects of co-runner-dependent cache allocation, whereas cache partitioning does not. A significant secondary effect is co-runner dependent contention for the memory bus (a high-miss-rate co-runner will get relatively more bus bandwidth). The cache-fair scheduler accounts for performance effects of bus contention its fair IPC model.

Another advantage of our algorithm over hardware solutions is that it is implemented in the operating system – a natural place to manage resource allocation. The OS has a global knowledge of the entire workload and thus can ensure that the cache-fair algorithm "plays well" with other resource management policies.

In the rest of the paper we describe the cache-fair algorithm (Section 2), the fair IPC model (Section 3), the implementation of the algorithm in Solaris 10 (Section 4), and our evaluation of it (Section 5). We discuss related work in Section 6, and conclude in Section 7.

2. Overview of the Algorithm

In this section we explain how the cache-fair algorithm improves performance isolation via adjustments to threads' CPU timeslices. In our examples we will assume a dual-core system with a shared second-level (L2) cache. We identify two performance metrics used in this paper:

Overall performance (or simply performance) is the thread's overall *CPU latency*: the time it takes to complete a logical unit of work (say, 500 million instructions). ***Fair performance*** refers to performance under fair cache allocation.

IPC is the thread's instructions per cycle rate. ***Fair IPC*** refers to IPC under fair cache allocation.

A thread's IPC is affected by the amount of cache allocated to that thread: a larger cache allocation usually results in a higher IPC, and vice versa. Therefore, co-runner-dependent variability in cache allocation causes co-runner-dependent variability in

IPC. The cache-fair scheduler offsets that variability by increasing or decreasing the thread's CPU timeslice.

Figure 1 illustrates this concept. There are three threads (A though C) running on a dual-core CMP with a shared cache. In the figure, each box corresponds to a thread. The height of the box indicates the amount of cache allocated to the thread. The width of the box indicates the thread's CPU timeslice. The area of the box is proportional to the amount of work completed by the thread. Stacked thread boxes indicate co-runners.

We show three scenarios resulting in different levels of performance isolation for Thread A: In Figure 1(a) Thread A runs with other threads on a conventional CMP with a *conventional* scheduler and experiences poor performance isolation. In Figure 1(b) it runs on a hypothetical CMP that enforces fair cache and memory bus allocation and thus experiences good performance isolation. In Figure 1(c) Thread A runs on a conventional CMP *with the cache-fair scheduler* and thus experiences good performance isolation.

In Figure 1(a) Thread A's IPC is below its fair IPC, because its cache allocation is below the fair level due to a high-miss-rate co-runner Thread B. As a result, Thread A's overall performance (shown on the X-axis as the CPU latency) is worse than its fair performance (achieved in Figure 1(b)). In Figure 1(c), Thread A still exhibits lower-than-fair IPC due to Thread B, but the cache-fair algorithm compensates for the reduced IPC by increasing Thread A's CPU timeslice. This allows Thread A to achieve fair performance overall.

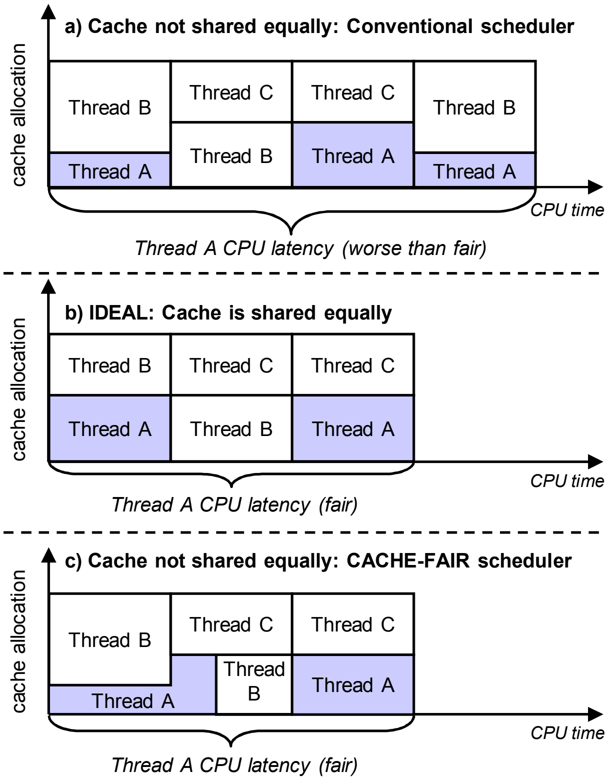


Figure 1. Illustration of the cache-fair algorithm

In Figure 1, Thread A's CPU timeslice was increased because its IPC was lower than its fair IPC. If, conversely, Thread A's IPC had been *higher* than its fair IPC, the scheduler would have *decreased* its CPU timeslice.

We note that the cache-fair algorithm does not establish a new scheduling policy, but instead helps enforce *existing* policies. For example, if the system is using a fixed priority policy, the cache-fair algorithm will make the threads run as quickly as they would if the cache were shared equally *given* the fixed priority policy.

Referring again to Figure 1(c), we note that as the scheduler increased the timeslice of Thread A, the timeslice of Thread B correspondingly *decreased*. This is how CPU time sharing works: if the CPU share of one thread is increased, the CPU share of some other thread will be decreased to compensate (and vice versa). We must ensure that those *compensatory* timeslice adjustments do not work against any previous adjustments made to ensure fairness.

To that end, we define two thread classes, a *cache-fair* class and a *best-effort* class. Threads in the cache-fair class are managed for improved performance isolation: the scheduler makes adjustments to those threads' timeslices to counter the effects of unfair cache allocation. Threads in the best-effort class are *not* managed for improved performance isolation: these could be background threads for which performance isolation is not important[1]. The scheduler performs compensatory adjustments to best-effort threads' timeslices as needed. In Figure 1(c), for example, Thread B is in the best-effort class, while Thread A is in the cache-fair class.

With this design, we must ensure that best-effort threads do not suffer significant performance penalties. In our experiments, the performance penalty experienced by best-effort threads was small, less than 1% on average. To prevent large performance penalties, the scheduler spreads compensatory adjustments among multiple best-effort threads whenever possible.

In addition to determining which threads' timeslices to adjust, the scheduler must also compute the amount by which to adjust each timeslice. Determining the right amount of adjustment is trivial as long as the scheduler knows by how much the

[1] A user specifies a thread's class in the same way as she specifies a thread's priority.

thread's actual IPC deviates from its fair IPC: The scheduler (1) computes how many instructions the thread would have completed if its IPC had been fair, (2) compares it with the number of instructions actually completed by the thread, and (3) computes the adjustment to the next CPU timeslice so that by the end of that timeslice the thread completes as many instructions as it would if its IPC were fair. Unfortunately, fair IPC values are not trivial to determine, because they cannot be measured directly. We estimate them using a new performance model, which we describe in the next section.

3. Fair IPC Model

Our model for fair IPC is comprised of two parts: we first estimate the *fair cache miss rate*, and then the fair IPC *given* the fair miss rate.

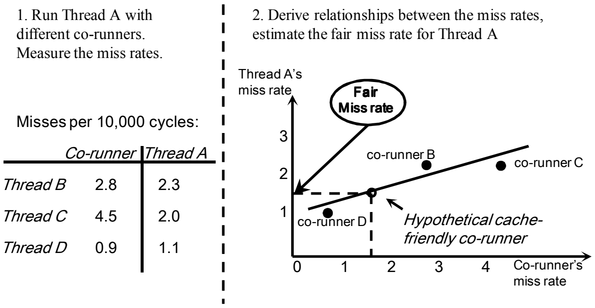
Fair cache miss rate is the miss rate experienced by the thread when it is allocated its fair cache share.

The novelty of our model is in techniques for estimating the fair cache miss rate; fair IPC (given the fair miss rate) can be estimated using existing techniques [30]. Therefore, we discuss the fair miss rate model and refer the reader to our other publication describing the entire fair IPC model [14]; details on how our fair IPC model accounts for memory bus contention can also be found in this work.

3.1. Overview of the model

Models for cache miss rates have been designed in the past [4,5,7,9,13,17,26,32], but those models were either too complex and high-overhead to use inside an OS scheduler, or required inputs that could not be easily obtained at runtime. We designed a simple online model.

For the purposes of this section we define the miss rate as the number of *misses per cycle* (MPC). Our approach for estimating the fair miss rate is based on an intuitive and empirically verified observation: if the co-runners have similar cache miss rates, they end up with equal cache allocations. Recall that the shared cache is allocated based on demand; intuitively, if the threads have similar demands (i.e., similar miss rates), they will have similar cache allocations.



1. Run Thread A with different co-runners. Measure the miss rates.

Misses per 10,000 cycles:

	Co-runner	Thread A
Thread B	2.8	2.3
Thread C	4.5	2.0
Thread D	0.9	1.1

2. Derive relationships between the miss rates, estimate the fair miss rate for Thread A

Figure 2. Estimating the fair cache miss rate for Thread A

Considering this assumption more formally, if we assume that a thread's cache accesses are uniformly distributed in the cache (validity of this assumption is discussed below), we can model cache replacement as a simple case of the *balls and bins* abstraction [10]. For two co-runners A and B, let their cache requests correspond to black and white balls respectively. We toss black and white balls into a bin. Each time a ball enters the bin, another ball is evicted from the bin. If we toss black and white balls at the same rate, then after enough tosses the number of black balls in the bin will form a multinomial distribution centered around one-half. Thus, two threads with equal L2 cache miss rates (balls being tossed at the same rate) will share the cache equally, or fairly. This result generalizes to any number of different coloured balls being tossed at the same rate [10]. Thus any *N* threads with the same cache miss rate will share the cache fairly.

We say that A and B are *cache-friendly* if they experience similar miss rates when running together (and, by our assumption, A and B share the cache fairly). Therefore, fair miss rate of A can be observed when A's miss rate equals its co-runner's miss rate. Based on that, to estimate the fair miss rate for Thread A (on a dual-core system) one could run it with different co-runners until detecting its cache-friendly co-runner (and recording the corresponding miss rate). That approach is not practical, however, since $O\binom{n}{m}$ tests (where *n* is the number of threads and *m* is the number of processors) are required to find a cache-friendly co-runner or to determine that none exists.

Instead we run Thread A with several different co-runners, derive a relationship between the miss rates of Thread A and its co-runners, and use that relationship to estimate Thread A's fair miss rate. Our goal is to find the miss rate that *would* be observed if Thread A and its co-runner had same miss rates. We use the derived relationship to estimate that miss rate. Figure 2 illustrates this process. We express the relationship between the co-runners' miss rates using a linear function. We experimentally found that a linear function approximated that relationship better than other simple functions. The resulting equation has the form:

$$MissRate(A) = a * \sum_{i=1}^{n} MissRate(C_i) + b \qquad (1),$$

where *n* is the number of co-runners, C_i is the *i*th co-runner, and a and b are the linear equation coefficients. By our definition:

$$FairMissRate(A) = MissRate(A) = MissRate(C_i),$$

for all i. Equation (1) can be expressed as:

$$FairMissRate(A) = a * n * FairMissRate(A) + b,$$

and: $$FairMissRate(A) = \frac{b}{1 - a * n} \quad (2).$$

3.2. Model evaluation

We evaluated the accuracy of our model by comparing the fair miss rates estimated by our model with the actual fair miss rates. We used nine SPEC CPU2000 benchmarks as our experimental workload. We computed the estimated fair miss rate by running each of the selected benchmarks with several different co-runners (also from the SPEC CPU2000 suite) on a simulated dual-core CMP, deriving the coefficients for Equation 1 via linear regression analysis, and then using Equation 2. We measured the actual fair miss rates in an experiment where the benchmarks ran on our experimental CMP with an equally partitioned cache (we implemented cache-partitioning in our simulator for this purpose).

Figure 3 shows how the estimated fair miss rates compare to the actual miss rates. The X-axis shows the names of the SPEC CPU2000 benchmarks we ran; the Y-axis shows the actual and estimated fair miss rates for each benchmark. The estimated miss rates closely approximate the actual miss rates. The difference between the measured and estimated values is within 8% for six out of nine benchmarks, within 25% for eight out of nine benchmarks.

We observed that our estimates were less accurate for benchmarks with relatively low miss rates than for benchmarks with relatively high miss rates (for *crafty*, we overestimated the fair cache miss rate by almost a factor of two). We hypothesize that because low-miss-rate benchmarks actively reuse their working sets, there is little variation in the miss rate when those benchmarks run with different co-runners; low variation in the miss rates used for regression analysis results in a low-fidelity linear equation.

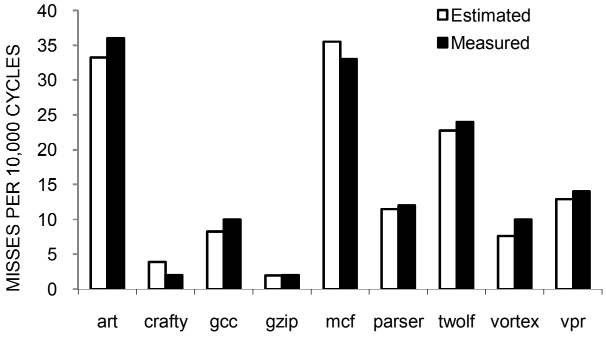


Figure 3. Estimated vs. the actual fair cache miss rate

A limitation of our model is that it requires running a thread with many co-runners that have diverse cache access patterns. If the workload has only a few threads, or if all threads have similar cache-access patterns, the linear equation will have low fidelity. In those situations, we could use one of the alternative, although more limiting, methods, such as a compiler-based model [7], a hardware-based model if the appropriate hardware becomes available [32], or we could co-schedule a thread with a synthetically generated benchmark sized to use exactly its fair share of the cache [12] and measure the thread's (fair) miss rate.

Our model assumes that cache requests are distributed uniformly across the cache, while this is not the case for many workloads [27]. We do not view this as a serious limitation, however: existing work on cache models that operated under the same assumption of uniformity showed that the assumption does not significantly affect model accuracy [5,9,15,23,32]. Relaxing this assumption is difficult in an online model, because cache access distribution cannot be obtained online via hardware, and obtaining it in software is costly [5].

4. Implementation

We implemented the cache-fair algorithm as a loadable module for Solaris 10. Module-based implementation makes our solution flexible: a user can enable the cache-fair scheduler only if needed and the scheduler can be tuned and upgraded independently of the core kernel.

To have a thread managed by the cache-fair scheduler the user invokes the Solaris system call `priocntl` and specifies the name of the cache-fair module as one of the arguments. The user specifies the thread's class, cache-fair or best-effort, also via `priocntl`.

Each cache-fair thread goes through two phases: *sampling* and *scheduling*. During the sampling phase, the scheduler gathers performance data and uses it to estimate the thread's fair miss rate. During the scheduling phase, the scheduler periodically monitors the thread's performance and adjusts the thread's CPU timeslice if its actual performance deviates from its fair performance.

When a thread is in the sampling phase the scheduler monitors the cache miss rates for it and its

Table 1. Configuration of the simulated machine

CPU cores	Two single-threaded processing cores, each running at 992 MHz.
L1 caches	Each core has a 16KB instruction cache and an 8KB data cache, both four-way set associative
L2 cache	Unified, shared, 1MB four-way banked, eight-way set associative.
Memory bus	4 GB/s peak bandwidth

co-runners via performance counters. We rely on performance counters commonly available on modern processors. We define a *run* as the contiguous time interval when a group of co-runners runs simultaneously. A run terminates when any of the co-runners gives up the CPU. At the end of the run we record the observed miss rates. Measurements recorded at the end of the run correspond to one data sample. We collect at least ten data samples for each cache-fair thread. The scheduler discards measurements from runs where the cache-fair thread executed fewer than 10 million instructions (to eliminate cold-start effects on cache miss rates [11]). Therefore, the sampling phase ends once the cache-fair thread has completed at least 100 million instructions in valid runs. At the end of the sampling phase, the scheduler estimates the fair miss rate using linear regression analysis.

The per-thread runtime overhead of performing regression analysis is determined by the number of samples we obtain during the sampling phase; this quantity is set to ten in our implementation. Since the quantity is fixed, the per-thread overhead does not grow with the number of cores or the number of co-runners. Therefore, the model's performance will likely scale well as the number of cores and threads increases.

The sampling phase needs to be repeated every time a thread changes its cache access patterns. An online phase detection algorithm would detect such a change [19,28], but unfortunately we are not aware of a phase-detection algorithm that works well on CMPs. Instead, we repeat the sampling phase every time a thread has completed one billion instructions. Infrequent repetitions of the sampling phase limit the overhead produced by linear regression.

After the sampling phase, the thread enters the scheduling phase. In this phase, the scheduler periodically monitors the thread's IPC (again via hardware counters), compares it to the thread's fair IPC (estimated using the thread's fair miss rate), and based on the difference between the two, adjusts the thread's CPU timeslice. The scheduler also performs the corresponding compensatory adjustment to a best-effort thread. It tries to spread compensatory adjustments evenly among all best-effort threads, to limit the penalty on any particular thread.

Table 2. Schedules for each benchmark

Principal	*Fast Schedule*	*Slow Schedule*
art	*art,crafty,crafty,crafty*	*art,mcf,mcf,mcf*
crafty	*crafty,vpr,vpr,vpr*	*crafty,mcf,mcf,mcf*
gcc	*gcc,vpr,vpr,vpr*	*gcc,mcf,mcf,mcf*
gzip	*gzip,crafty,crafty,crafty*	*gzip,mcf,mcf,mcf*
mcf	*mcf,gzip,gzip,gzip*	*mcf,crafty,crafty,crafty*
parser	*parser,crafty,crafty,crafty*	*parser,mcf,mcf,mcf*
twolf	*twolf,crafty,crafty,crafty*	*twolf,mcf,mcf,mcf*
vortex	*vortex,crafty,crafty,crafty*	*vortex,mcf,mcf,mcf*
vpr	*vpr,crafty,crafty,crafty*	*vpr,mcf,mcf,mcf*

Performance monitoring and timeslice adjustment is performed for each cache-fair thread every 50 million instructions. We determined experimentally that this frequency was sufficiently high to allow the threads to achieve fair performance within less than a half-second of the beginning of the scheduling phase, while keeping the scheduler overhead low (the cache-fair scheduler generated less than a 1% overhead as compared to the default scheduler).

5. Evaluation

We evaluated our implementation of the cache-fair scheduler using a multiprogram workload of SPEC CPU2000 benchmarks (Section 5.1) and database workloads: SPEC JBB and TPC-C benchmarks (Section 5.2). We compare performance isolation under the cache-fair scheduler and the Solaris fixed-priority scheduler, to which we refer as the *default* scheduler.

Our experimental hardware is a simulator of a dual-core CMP, based on the UltraSPARC® T1 architecture [18] and implemented as a set of Simics [20] modules. Table 1 summarizes its configuration parameters. This is a full-system simulator that executes the complete operating system and applications unmodified. Therefore, the operating system scheduler is *not* simulated and works the same way it would on real hardware.

5.1. Multiprogram workload experiment

We picked nine benchmarks from the SPEC CPU2000 suite, so as to represent a variety of cache access patterns. We run each benchmark, which we call the *principal* benchmark in two experiments, or *schedules.* In the first experiment, the principal benchmark runs with high-miss-rate threads – we call this the *slow schedule.* In the second experiment, the principal benchmark runs with low-miss-rate threads – we call this the *fast schedule.* Table 2 shows the benchmarks and the schedules. We assign the principal benchmark to the cache-fair class. We assign one of the three remaining threads to the best-effort class. The two remaining threads were not managed by the cache-fair scheduler. We run each schedule until the principal benchmark completes 500 million instructions in the scheduling phase. And the end, we measure the principal benchmark's performance isolation, i.e., the difference between its runtime in the fast and slow schedules.

We constructed this experiment such that the principal benchmark runs with three identical co-runners, to ensure that any performance differences between the cache-fair and default schedulers are due to differences in the scheduling algorithms, not to co-runner pairings. However because of the limited number of co-runners, it is not feasible to estimate the fair miss rate for a principal thread: there would not be enough different samples for the linear regression. Therefore, we estimate all principal benchmarks' fair miss rates in a separate experiment that includes all nine benchmarks.

5.1.1. Effect on performance isolation. For each principal benchmark, we computed performance variability (our metric for performance isolation) as the *percent slowdown in the slow schedule vs. the fast schedule*. We measured the time it takes the principal benchmark to complete 500 million instructions in the slow schedule, in the fast schedule, and computed the difference relative to the time in the fast schedule.

Figure 4 shows performance variability for each benchmark with the two schedulers. With the default scheduler (black bars) performance variability is substantial: it ranges from 5% to 28%. With the cache-fair scheduler, performance variability is negligible: below 4% for all benchmarks.

Performance variability in our experiments was caused by unfair L2 cache sharing (for example, *vpr'*s 19% slowdown in the slow schedule is explained by a 46% increase in its L2 miss rate over the fast schedule); but since the cache-fair scheduler accurately modeled the effects of unfair cache allocation on IPC, it was able to successfully eliminate the variability in the *overall* performance.

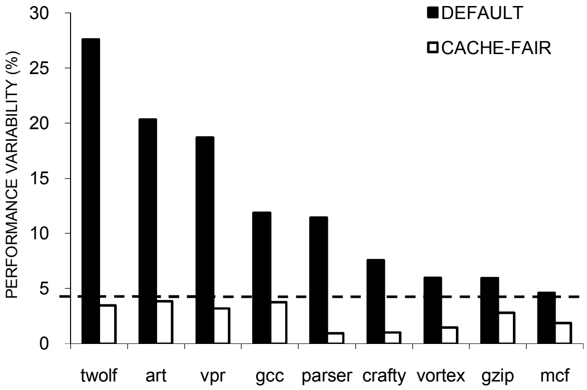


Figure 4. Performance variability with default and cache-fair scheduler. The dotted line is at 4%.

5.1.2. Effect on absolute performance. The cache-fair scheduler is expected to affect the absolute performance of applications it manages. Applications that did not get their fair share of the cache are expected to take less time to complete under the cache-fair scheduler, while applications that got more than their fair share are expected to take more time to complete.

Figure 5 shows completion times for each principal benchmark under the two schedulers. Completion times are shown as ranges. Ranges denoted by pairs of white circles correspond to the default scheduler, ranges denoted by black boxes correspond to the cache-fair scheduler. The top of the range boundary (either a circle centre or a box edge) indicates the completion time in the slow schedule (longer completion time); the lower range boundary is the time in the fast schedule (shorter completion time). The times for each benchmark are normalized to its completion time in the fast schedule with the default scheduler. Note that in this experiment five benchmarks completed more quickly with the cache-fair scheduler (box-delimited ranges are below circle-delimited ranges), while three benchmarks completed more slowly.

In this figure, we arrange the benchmarks along the X-axis in descending order of IPC. Thus high-IPC benchmarks appear on the left side in the figure, and low-IPC benchmarks appear on the right. Note that high-IPC benchmarks usually experience shorter completion times under the cache-fair scheduler than under the default scheduler (indicated by black boxes appearing below the circles). This is expected: high-IPC threads are usually less memory-bound, so they "fight" for cache allocation less aggressively and get less than their fair cache share as a result. This forces the cache-fair scheduler to increase the length of their

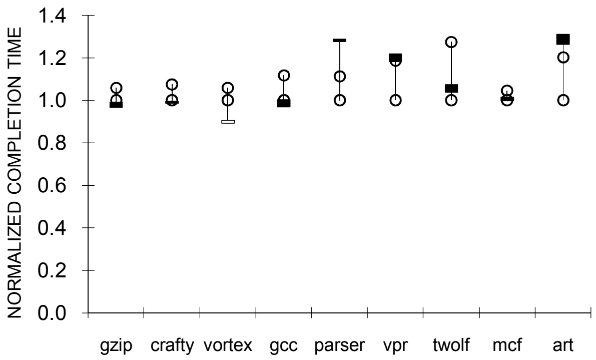


Figure 5. Ranges of normalized completion times with the two schedulers

CPU timeslice, which reduces their overall completion time.

We emphasize that the goal of the cache-fair scheduler is to provide performance comparable to that under fair cache sharing, not to increase absolute performance. Therefore, it is not surprising that the scheduler does not have a clear advantage over the default scheduler in terms of performance. Those applications that get less than their fair cache share will experience an increase in absolute performance, those that get more than their fair share will experience a decrease. Performance isolation, not absolute performance, is the focus of the scheduler.

5.1.3. Effect on overall throughput. We now evaluate the effect of the cache-fair scheduling algorithm on the instructions-per-cycle completed by the entire workload (i.e. the aggregate IPC). An alternative metric for throughput used on CMP architectures is weighted speedup [29]. We do not use weighted speedup, as it would not be affected by the cache-fair scheduler. Weighted speedup is determined by threads' individual IPCs, so it would only be affected if those IPCs change. The cache-fair scheduler, as we explained, does not change threads' individual IPCs, only the overall runtime.

We present aggregate IPCs of slow schedules in Figure 6. Each schedule is identified by the name of the principal benchmark. Each schedule's IPC is normalized to its IPC under the default scheduler. Black bars correspond to the default scheduler, white bars – to the cache-fair scheduler. We omit the figure for the fast schedules, but summarize the results.

In *parser*'s schedule, the IPC was 12% lower under the cache-fair scheduler than under the default scheduler. It turned out that *parser*'s fair miss rate was overestimated, so *parser*'s CPU timeslice was reduced more than necessary. As a result, that schedule's best-effort thread *mcf*, a low-IPC thread, occupied a larger fraction of CPU time under the cache-fair scheduler. The aggregate IPC decreased as a result. Improving the accuracy of the fair miss rate model would address this problem.

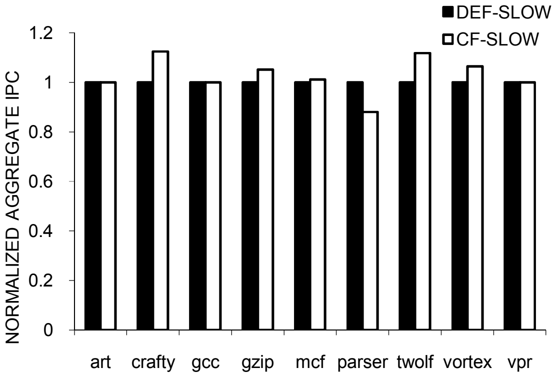


Figure 6. Aggregate IPC for slow schedules with the default scheduler and cache-fair scheduler

For the rest of the schedules, on the other hand, the aggregate IPC either increased (by 1-12% for five out of nine schedules) or remained roughly unchanged (for three out of nine schedules). The largest IPC increase (12%) was in the schedule with *crafty* as the principal benchmark. *crafty* failed to achieve its fair IPC when running with a cache-demanding co-runner *mcf*. As a result, *crafty*'s share of CPU time was increased relatively to *mcf*'s. *crafty* is a relatively high-IPC thread, so the aggregate IPC increased as a result.

For fast schedules, the aggregate IPCs remained largely unchanged. For eight out of nine schedules, the IPC changed by at most +/- 3% in comparison with the default scheduler. In the schedule with *art* as the principal benchmark, the throughput increased by 8%. *art* is a high-miss-rate and low-IPC application; it occupied more than its fair cache share, which forced the cache-fair scheduler to reduce its CPU share. As a result, the system executed fewer instructions from low-IPC *art* and more instructions from *art*'s high-IPC co-runner *crafty*. That led to the increase in the aggregate IPC.

We saw that the effect on aggregate IPC depends on the relative IPCs of the threads whose timeslices are being adjusted. In a workload with a balanced mix of high-IPC and low-IPC threads, we will typically see that high-IPC threads (that typically get less than their fair cache share) will have their CPU shares increased, while low-IPC threads (that typically get more than their fair cache share) will have their CPU shares decreased. In such a workload, we expect that the aggregate IPC will increase under cache-fair scheduler.

5.1.4. Effect on best-effort threads. We now evaluate the cache-fair scheduler's performance effect on best-effort threads. Recall the schedules in Table 2 and note that there is only one best-effort thread in the schedule. Therefore, our experiment permits evaluating the *worst-case* performance penalties on best-effort threads: when there is only one best-effort thread, the scheduler is unable to spread compensatory adjustments among many threads.

Our evaluation led us to the following conclusions: (1) in general, performance penalties for best-effort threads are small; (2) to avoid large penalties it is important to distribute the penalties among multiple best-effort threads.

Table 3 shows the slowdown (vs. the default scheduler) experienced by the best-effort thread in each schedule. The first and third columns identify the slow and fast schedules respectively (by the principal benchmark and the corresponding best effort thread).

Table 3. Percent slowdown for the best-effort threads. Negative values indicate speedup.

SLOW SCHEDULES	Best-effort slowdown	FAST SCHEDULES	Best-effort slowdown
art-mcf	-8.07%	*art-crafty*	-23.38%
crafty-mcf	4.58%	*crafty-vpr*	2.69%
gcc-mcf	**26.70%**	*gcc-vpr*	11.23%
gzip-mcf	7.14%	*gzip-crafty*	1.40%
mcf-crafty	-0.63%	*mcf-gzip*	6.90%
parser-mcf	-11.34%	*parser-crafty*	**-23.84%**
twolf-mcf	15.79%	*twolf-crafty*	-3.11%
vortex-mcf	8.51%	*vortex-crafty*	5.84%
vpr-mcf	2.32%	*vpr-crafty*	-13.43%
SLOWDOWN SUMMARY:			
Mean:	0.52%	**Max:**	26.70%
Median:	2.50%	**Min:**	-23.84%

The second and fourth columns show how much less CPU time (in percent) the best-effort thread received under the cache-fair scheduler vs. the default scheduler. Positive values indicate slowdown, negative values indicate speed-up.

On average, the cache-fair algorithm resulted in negligible (less than 1%) performance penalties for best-effort threads. In one third of the schedules the threads experienced slowdown of more than 6%, and in one schedule (*gcc-mcf)* of as much 27%. Had there been multiple best-effort threads, compensatory adjustments would have been distributed among them, reducing the penalty on a single thread. We note that in about one third of the schedules, the best-effort threads experienced a speed-up (of 12% on average).

We suppose that having multiple cache-fair threads in a workload, would further help to soften the performance penalty on best-effort threads: Compensatory adjustments caused by different cache-fair threads will likely cancel out. This can be explained with an example:

Suppose there are two cache-fair threads in the system: *Tcache_hungry* and *Tcache_moderate*. *Tcache_hungry* uses more than its fair cache share, while *Tcache_moderate* is forced to use less than its fair share. Therefore, the scheduler will increase the timeslice of *Tcache_moderate* by some amount X, and decrease the timeslice of *Tcache_hungry* by some amount Y. After increasing the timeslice of *Tcache_moderate* the scheduler will pick a best-effort thread, we will call it *Tbest_effort*, to apply the compensatory adjustment of –X to its timeslice. Similarly, after decreasing the timeslice of Tcache_hungry by Y, the scheduler will need to pick a best-effort thread whose timeslice it will increase by Y. The scheduler will pick *Tbest_effort* in order to offset the previous penalties on *Tbest_effort*'s timeslice. As a result, *Tbest_effort*'s timeslice will be increased by Y, cancelling in part or in full the penalty of –X imposed by the compensatory adjustment of *Tcache_moderate*. Therefore, the overall performance penalty on *Tbest_effort* is reduced due to there being multiple cache-fair threads in the system.

Table 4. SPEC JBB with the two schedulers

Schedule	Default sched.	Cache-fair sched.
Slow	2497 txn/sec	2435 txn/sec
Fast	2728 txn/sec	2448 txn/sec
Difference	**9%**	**1%**

This effect occurs only if the system has roughly as many threads that use more than their fair cache share as the threads that use less than their fair share. This is reasonable to expect: if a thread has not gotten its fair cache share there must have been another thread that has caused that by using more than its fair share.

5.2. Experiments with database workloads

We describe our experiments with two database benchmarks: SPEC JBB and TPC-C. (We built our own implementation of TPC-C on top of Oracle Berkeley DB [1]). We ran two sets of experiments: one where SPEC JBB is the principal benchmark, and another one where TPC-C is the principal benchmark. We run each principal benchmark in the slow and in the fast schedule, just as in our SPEC CPU2000 experiments. We evaluate performance isolation with respect to *transactions per second*: we measure performance variability as the difference in transaction rates between the two schedules. This application-level performance metric is often more meaningful for users than IPC.

SPEC JBB and TPC-C emulate database activities of an order-processing warehouse. These benchmarks can be run with databases of various sizes. The size is determined by the number of warehouses, and the standard number of warehouses is ten. Because our simulator had a low upper limit for physical memory (only 4GB) we were forced to use a smaller database size (and hence fewer warehouses), to avoid physical memory paging. Because we used a reduced database size, we configured the simulator with a smaller L2 cache: 512KB. The number of threads used by the benchmark is also configurable – we use one thread in the principal benchmark, as this simplified measurement.

5.2.1. SPEC JBB. In the slow schedule, SPEC JBB's co-runners were TPC-C configured with five warehouses (TPC-C_5WH) and *twolf* (used as the best-effort thread). In the fast schedule, SPEC JBB's co-runners were TPC-C configured with one warehouse

(TPC-C_1WH) and *twolf* (used as the best-effort thread). We pinned threads to CPU cores as follows:

Schedule	Core 0	Core 1
Slow	SPEC JBB, *twolf*	TPC-C_5WH
Fast	SPEC JBB, *twolf*	TPC-C_1WH

Pinning threads to CPU cores in this fashion prevents any performance effects due to changing co-runners (the principal always runs with the same co-runner when the threads are pinned).

Table 4 reports SPEC JBB's transactions per second (txn/sec) for each schedule with the two schedulers. The bottom row shows the difference. With the default scheduler, the co-runner-dependent difference in the transaction rate is 9%. With cache-fair scheduler, the difference is only 1%.

In the fast schedule, SPEC JBB completed fewer transactions per second with the cache-fair scheduler than with the default scheduler. This happened because SPEC JBB occupied more than its fair cache (61% as measured by our simulator), and so the cache-fair scheduler reduced its CPU share.

In the slow schedule, SPEC JBB achieved roughly the same transaction rate under the cache-fair scheduler as under the default scheduler. When SPEC JBB ran with TPC-C_5WH, it used roughly half the cache (in fact, the measured cache share was exactly 50%). The cache-fair scheduler, therefore, did not adjust SPEC-JBB's CPU share, and so SPEC JBB achieved the same performance as under the default scheduler.

5.2.2. TPC-C. In this experiment the principal benchmark was TPC-C configured with two warehouses (TPC-C_2WH). In the slow schedule we ran it with SPEC JBB and *twolf*. In the fast schedule, we ran it with Sphinx [3] and *twolf*. (Sphinx is a speech recognition benchmark, representative of the workload used in online voice recognition servers). *twolf* was the best-effort thread in both schedules. The assignment of threads to CPUs was as follows:

Schedule	Core 0	Core 1
Slow	TPC-C_2WH, *twolf*	SPEC JBB
Fast	TPC-C_2WH, *twolf*	Sphinx

Table 5 shows the variability in TPC-C's transaction rate with the two schedulers. Under the default scheduler, the difference in transaction rate was 13%, while under the cache-fair scheduler it was only 1%.

In the slow schedule TPC-C ran more quickly under the cache-fair scheduler than under the default scheduler. In the slow schedule, TPC-C occupied only 38% of the cache, indicating that its high-miss-rate co-runner, SPEC JBB, reduced TPC-C's fair cache share.

Table 5. TPC-C with default and cache-fair schedulers

Schedule	Default sched.	Cache-fair sched.
Slow	902 txn/sec	1028 txn/sec
Fast	1018 txn/sec	1035 txn/sec
Difference	**13%**	**1%**

The cache-fair scheduler gave TPC-C an extra 14% of CPU time to compensate for unfair cache allocation.

5.3. Comparison with cache partitioning

We compared the cache-fair scheduler with cache partitioning in terms of their ability to improve performance isolation. Cache partitioning eliminates unfair cache allocation and can also improve cache utilization by allocating cache in a more efficient manner [23]. However, we found that when it came to reducing co-runner-dependent performance variability, the cache-fair scheduler did significantly better than cache partitioning.

We configured our simulator to equally partition the L2 cache among the two cores using way-partitioning, and ran the slow and fast schedules presented in Table 2. Partitioning reduced co-runner-dependent performance variability for only three out of nine benchmarks and made no difference for the remaining six. The reason is that cache partitioning does not eliminate co-runner-dependent contention for the memory bus. To confirm this, we ran another experiment where the simulator was configured with infinite memory bus bandwidth (to eliminate the variability in the bus contention); in that experiment cache partitioning did eliminate performance variability. Memory bus is a highly contended resource on CMPs [31], so taking that contention into account is necessary in order to reduce co-runner-dependent performance variability. The cache-fair algorithm accounts for the memory-bus contention and thus improves performance isolation more effectively than cache partitioning.

5.4. Comparison with OS-level page allocation

OS-level page allocation can be used to enforce equal cache sharing directly, by controlling where data is allocated in the cache [33]. In contrast, our approach *compensates* for unequal sharing. While a thorough study would be needed to compare the two approaches in detail, one difference between them is that OS-level page allocation, unlike our approach, could cause inefficient cache use. Suppose the system allocated equal cache portions to threads T_a and T_b (via OS-level page allocation), but T_a did not use its entire cache share. A part of T_a's cache share would be wasted.

Detecting this inefficiency is not trivial without adequate hardware support. In contrast, our algorithm does not alter cache allocation, it only compensates applications when the allocation is unfair.

6. Related work

We compare and contrast our work with existing solutions for improving performance isolation on CMPs.

Hardware solutions employ changes to the CMP processor that either enforce fair resource allocation or expose control over resource allocation to the operating system [8,9,11,24,25,32]. The advantage of hardware solutions is that they can address performance isolation *directly*, and thus require few or no modifications to the operating system. However, as we have shown, simple hardware solutions, such as cache partitioning, do not address the problem effectively; at the same time complex modifications can make the hardware prohibitively costly. In addition, hardware solutions are usually less flexible and require longer time to market than software solutions. To the best of our knowledge, none of the previously proposed hardware solutions has been made commercially available. The cache-fair scheduling algorithm, on the other hand, can be used on systems that exist today.

Software solutions related to resource allocation on CMPs usually employ co-scheduling, i.e., scheduling a thread with the *optimal* co-runner. Co-scheduling has been used to improve performance [22,29] and performance isolation [16]. The key difference of co-scheduling is that it may be able to actually *force* cache allocation to be fair by selecting the "right" co-runner for the thread. On the other hand, if the right co-runner cannot be found, co-scheduling cannot be used. The cache-fair scheduling algorithm does not have that limitation. Better scalability is another potential advantage of the cache-fair scheduler. Co-scheduling requires co-ordination of scheduling decisions among the processor's cores and may thus limit the scheduler's scalability if the number of cores is large. The cache-fair scheduler, on the other hand, does not require inter-core coordination. Cache-fair algorithm is, to the best of our knowledge, the first CMP scheduling algorithm that does not use co-scheduling. On future CMP systems with dozens of cores this design may exhibit significant scalability advantages.

7. Summary

We presented the cache-fair scheduling algorithm, a new algorithm that improves performance isolation on CMPs. We evaluated it using our implementation in a commercial operating system. We showed that this algorithm almost entirely eliminates co-runner-dependent performance variability, and as such, significantly improves performance isolation. The cache-fair algorithm is more effective, less costly, and more flexible than hardware cache partitioning. It is also potentially more robust and scalable than existing software solutions.

8. Acknowledgements

We would like to thank Sun Microsystems for supporting this work. We also thank Andreas Moestedt of Virtutech for providing excellent technical support for Simics.

9. References

[1] Berkeley DB. Oracle Embedded Database. *http://www.oracle.com/database/berkeley-db.html*

[2] Consolidation and Virtualization. *http://www.sun.com/datacenter/consolidation/index.jsp*

[3] Sphinx-4. *http://cmusphinx.sourceforge.net/sphinx4/*

[4] A. Agarwal, J. Hennessey, and M. Horowitz. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184-215, 1989

[5] Erik Berg and Erik Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS),* 2004

[6] E. Berger, S. Kaplan, B. Urgaonkar, P. Sharma, A. Chandra, and P. Shenoy. Scheduler-Aware Virtual Memory Management. *Poster, Symposium on Operating Systems Principles (SOSP),* 2003

[7] C. Cascaval, L. DeRose, D. A. Padua, and D. Reed. Compile-Time Based Performance Prediction. In *Proceedings of the 12th Intl.Workshop on Languages and Compilers for Parallel Computing,* 1999

[8] F. J. Cazorla, Peter M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable Performance in SMT Processors. In *Proceedings of the 1st Conference on Computing Frontiers,* 2004

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Multi-Processor Architecture. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture,* 2005

[10] Richard Cole, Alan M. Frieze, Bruce M. Maggs, Michael Mitzenmacher, Andrea W. Richa, Ramesh K. Sitaraman, and Eli Upfal. On Balls and Bins with Deletions. In *Proceedings of the Second International Workshop on*

Randomization and Approximation Techniques in Computer Science, 1998

[11] G. Dorai and D. Yeung. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT),* 2002

[12] Daniel Doucette and Alexandra Fedorova. Base Vectors: A Potential Technique for Microarchitectural Classification of Applications. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-34,* 2007

[13] P. K Dubey, A. Krishna, and Mark S. Squillante. Analytic Performance Modeling for a Spectrum of Multithreaded Processor Architectures. In *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems,* 1995

[14] Fedorova, Alexandra. Operating System Scheduling for Chip Multithreaded Processors. Harvard University, Division of Engineering and Applied Sciences. 11-7-2006

[15] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. A Non-Work-Conserving Operating System Scheduler for SMT Processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA-33,* 2006

[16] R. Jain, C. J. Hughes, and S. V. Adve. Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS),* 2002

[17] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT),* 2004

[18] Poonacha Kongetira. A 32-way Multithreaded SPARC(R) Processor. In *Proceedings of the 16th Symposium On High Performance Chips (HOTCHIPS),* 2004

[19] J. Lau, S Schoenmackers, and B. Calder. Transition Phase Classification and Prediction. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture,* February 2005

[20] Daniel Nussbaum, Alexandra Fedorova, and Christopher Small. The Sam CMT Simulator Kit. *Sun Microsystems TR 2004-133,* 2004

[21] K. Olukotun, B Nayfeh, and L. Hammond. The Case for a Single-Chip Multiprocessor. In *Proceedings of the Seventh International Conference On Architectural Support For Programming Languages And Operating Systems (ASPLOS),* 1996

[22] Sujay Parekh, Susan J. Eggers, and Henry M. Levy. Thread-Sensitive Scheduling for SMT Processors. *University of Washington TR 2000-04-02,* 2004

[23] M. K. Qureshi and Yale Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th International Symposium on Microarchitecture,* 2006

[24] S. E. Raasch and S. K. Reinhardt. Applications of Thread Prioritization in SMT Processors. In *Proceedings of the Workshop On Multi-Threaded Execution, Architecture and Compilation,* 1999

[25] N. Rafique, W. T. Lim, and M. Thottethodi. Architectural Support for Operating System-driven CMP Cache Management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques,* 2006

[26] R. Saavedra-Barrera, D. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures,* 1990

[27] Alex Settle, Joshua L. Kihm, Andrew Janiszewski, and Daniel A. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT),* 2004

[28] X. Shen, Y. Zhong, and C. Ding. Locality Phase Prediction. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2004

[29] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2000

[30] Yan Solihin, V. Lam, and Josep Torrellas. Scal-Tool:Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors. In *Proceedings of the 1999 Conference on Supercomputing,* 2006

[31] Lawrence Spracklen and Santosh G. Abraham. Chip Multithreading: Opportunities and Challenges. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture,* 2005

[32] G. E. Suh, S Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture,* 2002

[33] D. Tam, Azimi R., Soares L., and Stumm M. Managing Shared L2 Caches on Multicore Systems in Software. *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-34*