

Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler

Abstract

We describe a new operating system scheduling algorithm that improves performance isolation on chip multiprocessors (CMP). Poor performance isolation is the phenomenon where an application's performance is determined by the behavior of other simultaneously running, or co-running, applications. This phenomenon is caused by unfair, co-runner-dependent allocation of shared cache on CMPs, which causes co-runner-dependent performance. Poor performance isolation, interferes with the operating system's control over priority enforcement, complicates per-CPU-hour billing and hinders QoS provisioning. Previous solutions required modifications to the hardware, and thus had limited flexibility and long time-to-market. We present a new software solution. The cache-fair scheduling algorithm reduces the dependency of an application's performance on the co-runners by ensuring that the application always runs as quickly as it would under fair cache allocation, regardless of how the cache is actually allocated. If the application executes fewer instructions per cycle (IPC) than it would under fair cache allocation, the scheduler increases that thread's CPU timeslice. This way, even if the thread's IPC is lower than it would be under fair cache allocation, the overall performance does not suffer because the application is allowed to use the CPU longer. We demonstrate, using our implementation of the algorithm in Solaris 10, that this algorithm significantly improves performance isolation for workloads such as SPEC CPU, SPEC JBB and TPC-C.

1. Introduction

Applications running on chip multiprocessors (CMP) [21] suffer from poor performance isolation [9,10,16]. This is a phenomenon where an application's performance is determined by the behavior of the co-running applications. Such performance dependency is due to inherently unfair, co-runner-dependent allocation of shared caches on CMPs. On CMPs, cache allocation is determined by the co-running threads' relative cache demands, fairness is not considered. A thread "demands" a new 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 affects its co-runners' cache allocations and, as a result, their performance. Accordingly, an application will run more slowly with a high-miss-rate co-runner than with a low-miss rate co-runner.

Poor performance isolation causes problems. One problem is non-deterministic and unpredictable behavior of

the OS scheduler, which leads to a weakened control over priority enforcement. It is difficult for the scheduler to ensure a faster forward progress for a high-priority thread on a CMP processor, because that 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 [1]. If in a given CPU-hour an application runs slowly only due to the misfortune of having a high-miss-rate co-runner, billing that application for the full hour is unfair. Yet, existing systems have no way of detecting or preventing this.

Poor performance isolation hinders QoS provisioning. QoS is provisioned via resource reservations, such as a dedicating a fraction of CPU cycles to a customer's application. Poor performance isolation makes resource reservations less effective, since an application could be unpredictably slowed down 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 [9,10,16]. (Co-runner dependent performance variability is used 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 [10]. 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-runner, as opposed to a low-miss-rate co-runner.

Previous work addressed performance isolation in hardware, via cache partitioning [9,16,23]. While cache partitioning ensures fair cache allocation, it increases 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 the IPC, determines the thread's 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 the IPC, and, finally, co-runner-dependent

variability in the overall performance. The OS cannot control the variability in the IPC (because it cannot control cache allocation), so it 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 (i.e., the *fair IPC*), it increases the thread’s CPU timeslice. Conversely, 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.

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. More specifically, the cache-fair scheduler reduced co-runner dependent performance variability from as much as 28% to under 4% for all the benchmarks. We also found that the scheduler generates only a negligible performance overhead (<1%).

While performance isolation has been addressed before in the context of shared physical memory [7], addressing it in the context of shared caches is more difficult. The allocation of physical memory is directly controlled by the operating system, but cache allocation is not. Since the cache-fair scheduler cannot *enforce* cache allocation to be fair, it needs to *compensate* when the allocation is unfair. To provide just the right amount of compensation, the scheduler needs to determine to what extent 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 under fair cache allocation. To determine the fair IPC in the scheduler, we designed a new low-overhead heuristical cache model.

We compared the effectiveness of the cache-fair scheduler and of cache partitioning (the 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 scheduler accounts for *secondary* performance effects of co-runner-dependent cache allocation, while 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 that effect by modeling memory-bus contention in its fair IPC model.

Another advantage of our algorithm over hardware solutions is that it is implemented in the operating system, and the operating system is a natural place to manage resource allocation. OS has a global knowledge of the entire workload and can thus ensure that the cache-fair policy 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 the evaluation (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 stress the distinction between two performance metrics that we use 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.

A thread’s IPC is affected by the amount of cache allocated to that thread: a larger cache allocation results in a higher IPC, and vice versa. Therefore, co-runner-dependent variability in cache allocation causes co-runner-dependent variability on the IPC. The 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, there is a box corresponding to each 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. Thread boxes stacked on top of one another 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 thus 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 due to the high-miss-rate co-runner Thread B, Thread A’s cache allocation is below the fair level. 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 experiences the lower-than-fair IPC due to Thread B, but the cache-fair algorithm compensates for that reduced IPC by increasing Thread A’s CPU timeslice. This allows Thread A to achieve the fair performance overall.

We showed an example where 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* that thread's CPU timeslice.

We note that the cache-fair algorithm does not establish a new scheduling policy but 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 has 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 as a corollary (and vice versa). The question is: how can the scheduler ensure that those corollary CPU share adjustments do not disrupt the primary timeslice adjustments required by the algorithm? In some situations, the scheduler can ensure that the corollary adjustment carries over to the thread whose timeslice has to be adjusted by that amount anyway (i.e., if a scheduler increases the timeslice of some low-miss-rate thread, there could be a high-miss rate thread whose timeslice needs to be adjusted by the corresponding opposite amount). However, one cannot guarantee that those situations will always present themselves when needed.

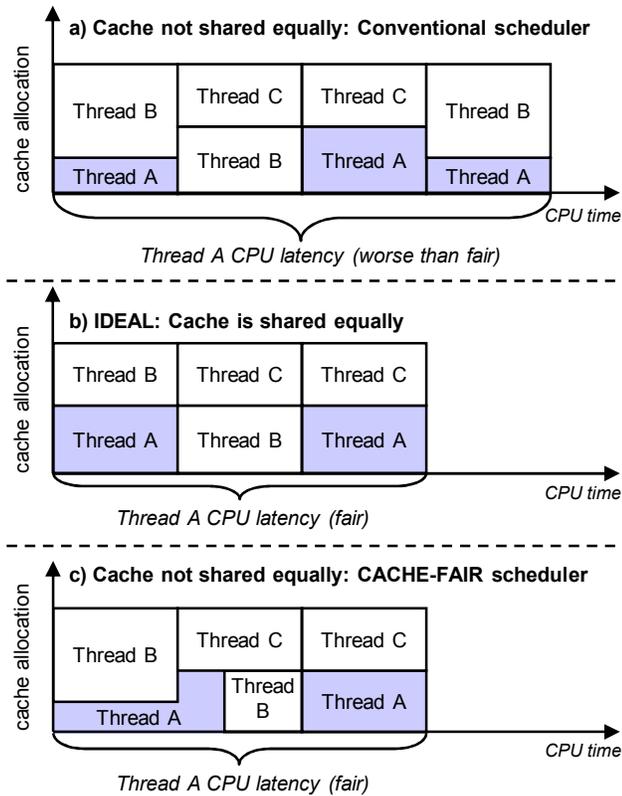


Figure 1. Illustration of the cache-fair algorithm

To ensure that the scheduler works well in all circumstances, we define a new thread class: the *best-effort class*. Threads in the best-effort class are not managed for improved performance isolation. These could be background threads or other threads, where performance isolation is not important¹. (In contrast, threads that *are* managed for improved performance isolation are in the *cache-fair* class.) In an event that a corollary CPU timeslice adjustment cannot be carried over to a cache-fair thread that anyhow requires such an adjustment, the scheduler adjusts the timeslice of a best-effort thread. The adjustment can be both positive as well as negative and the scheduler is designed to avoid significant negative adjustments: (1) the adjustments are spread among several best effort threads (if possible), and (2) there is a threshold on the adjustment to a single best effort thread's timeslice. As we demonstrate in the evaluation section, the performance effect on best-effort threads is small (<1% on average).

Finally, we discuss how the scheduler computes the amount by which to adjust the CPU timeslices. Determining right the amount of adjustment is trivial as long as the scheduler knows by how much the thread's actual IPC deviates from its fair IPC: the scheduler (1) computes how many instructions the thread would have completed under fair IPC, (2) compares it with the number of instructions completed in actuality, and (3) computes the adjustment to the next CPU timeslice such that by the time when that timeslice expires the thread will have completed the number of instructions corresponding to the fair IPC. Unfortunately, the fair IPC is not trivial to determine, because it cannot be measured directly. We estimate the fair IPC using a new performance model. We describe this model 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*, i.e., the miss rate under fair cache allocation, and then the fair IPC *given* the fair miss rate. The novelty of our model is in techniques for estimating the fair cache miss rate. Given that miss rate, we estimate the corresponding IPC using by and large existing techniques[19,28,31]. Therefore, in this paper we discuss the details of the fair cache miss rate model only. For details on the entire model, we refer the reader to another publication [anon].

3.1. Overview of the model

Models for cache miss rates have been designed before [4,6,8,10,13,16,24,30], but previous models were either too complex to use inside an OS scheduler or required inputs that could not be easily obtained online.

For the purposes of this section we define the miss rate

¹The user specifies the job's class in the same way she specifies a job's priority.

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 that if the co-runners have similar cache miss rates they end up with equal cache allocations. Intuitively, since the shared cache is allocated based on demand, if the threads have similar demands (i.e., similar miss rates), they will have similar cache allocations.

To back this assumption more formally, if we assume that a thread's cache accesses are uniformly distributed in the cache, we can model cache replacement as a simple case of the *balls and bins problem*[11]. Assume two co-runners, whose 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 throw the black and white balls at the same rate, then the number of black balls in the bin after many tosses will form a multinomial distribution centered around one-half. This result generalizes to any number of different colored balls being tossed at the same rate [14]. Thus, two threads with the same L2 cache miss rates (balls being tossed at the same rate) will share the cache equally, or fairly.

We say that co-runners A and B are *cache-friendly* if they experience similar miss rates when running together. By our assumption, A and B would be sharing the cache fairly and experiencing the fair cache miss rate. To find the fair miss rate for Thread A (on a dual-core system) at runtime, one could observe Thread A's miss rates with different co-runners until detecting its cache-friendly co-runner. (We identify a cache-friendly co-runner upon detecting a situation that Thread A and that co-runner have similar miss rates). That approach is not practical, however, because finding a cache-friendly co-runner requires $O(n^m)$ tests (where n is the number of threads and m is the number of processors), and may still be unsuccessful if none of the threads happens to be cache-friendly with Thread A.

Instead we run Thread A with several different co-runners, derive a relationship between the miss rates of Thread A's 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

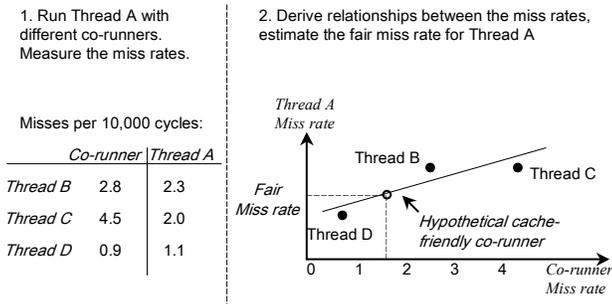


Figure 2. Estimating the fair cache miss rate for Thread A

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:

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

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

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

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 evaluate the accuracy of the fair miss rates estimated using our model by comparing them with the actual fair miss rates. We derived the actual fair miss rate for a thread by measuring the miss rate when this thread runs with a co-runner on a simulated dual-core machine with an equally partitioned cache. We computed the estimated fair miss rate by running each thread with several different co-runners, deriving the coefficients for Equation 1 via linear regression analysis, and then using Equation 2. We validated the model for multiprogram workloads of SPEC CPU2000 benchmarks.

Figure 3 shows how the estimated fair miss rates compare to the actual miss rates. The names of the SPEC CPU2000 benchmarks are on the X-axis; 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

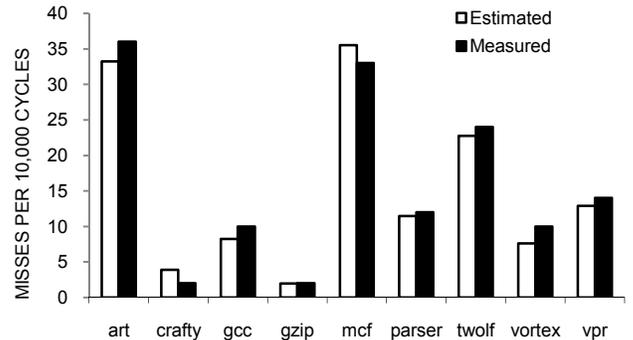


Figure 3. Estimated vs. the actual fair cache miss rate

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 resulted in the linear equation with low fidelity.

A limitation of our model is that to estimate a miss rate for a thread one must run that thread with many different co-runners. If the workload has only a few co-runners or if all co-runners have similar cache-access patterns, the linear equation will have a lower fidelity.

Another weakness is the model's assumption of uniform distributions of cache request across the cache. Many benchmarks were shown to have non-uniform distributions [25]. Relaxing this assumption would be very difficult in an online model, because access distribution cannot be obtained online on existing hardware. Given that our model had good accuracy in spite of that assumption we decided that not relaxing that assumption was an acceptable option.

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.

The user asks that a thread is managed by the cache-fair scheduler via the Solaris' `pricntl` system call. Similarly, the user specifies the thread's class: the *cache-fair* class (the threads for which the scheduler provides performance isolation), or the best-effort class.

Each cache-fair thread managed by the cache-fair scheduler goes through to two phases, *preparation* phase and *scheduling* phase. During the preparation phase, the scheduler gathers performance data needed to estimate the fair cache miss rate for the thread, and estimates the fair miss rate. During the scheduling phase, the scheduler periodically monitors the thread's performance and adjusts the thread's CPU timeslice if the actual performance deviates from the fair performance.

When a thread is in the preparation phase the scheduler monitors cache miss rates for that thread and its co-runners. For the monitoring, the scheduler uses performance counters available on modern processors. Accessing performance counters requires little performance overhead. The preparation phase ends once the cache-fair thread has completed at least 100 million instructions. Once the phase is complete, the miss rates are run through the linear regression analysis to estimate the fair miss rate. The scheduler discards miss rates from runs where the cache-fair thread executed less than 10 million instructions: this eliminates cold-start effects [12].

The model's running time for a given thread is determined by the number of observations we want to

obtain for the linear regression (we set it to ten). That quantity is fixed, so the model's running time does not grow with the number of cores or with the number of co-runners. This implies that the model will scale well on systems with large number of cores and threads.

The preparation phase needs to be repeated every time a thread changes its cache access patterns. An online phase detection algorithm would detect such a change [5,18,26], but unfortunately we are not aware of a phase-detection algorithm that works well on CMPs. We repeat the preparation phase every time a thread has completed one billion instructions. Infrequent repetitions of the preparation phase limit the overhead from running linear regression.

After the preparation phase, the thread enters the scheduling phase. In that phase, the scheduler periodically monitors the thread's IPC, compares it to its 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 keeps track of positive and negative CPU timeslice adjustments that have been performed, and if they do not cancel out, selects a best-effort thread whose timeslice it adjusts to make up for the difference.

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

5. EVALUATION

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

Our experimental hardware is a simulator dual-core CMP, based on the UltraSPARC T1 architecture [17] and implemented as a set of Simics [20] modules. Table 1 summarizes the system configuration parameters. This is a full-system simulator that executes the complete operating system and applications unmodified. Therefore, the operating system scheduler *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, ensuring that they represent a variety of cache access patterns. We run each benchmark with high-miss-rate threads (*slow schedule*) and with low-miss-rate threads (*fast schedule*). Table 2 shows the benchmarks and the

Table 1. Configuration of the simulated machine

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

schedules. The *principal* benchmark is the benchmark whose performance isolation we measure; it belongs to the cache-fair class. One of the other three threads (arbitrarily selected) is in the best-effort class.

We constructed this experiment such that the principal benchmark runs with the threads that are identical. This ensures that any performance differences between the cache-fair and default schedulers are due to differences in the scheduling algorithms, not to co-runner pairings. Since the benchmark runs with identical threads, it would not make sense to estimate its fair miss rate in that setting: we would not have enough different data points for the linear regression. Therefore, we estimate all benchmarks' fair miss rates in a separate run, where all principal benchmarks run together. We run each schedule until the principal benchmark completes 500 million instructions in the scheduling phase.

5.1.1. Effect on performance isolation

We compare performance variability with the default and cache-fair schedulers. (Recall that co-runner-dependent performance variability is our metric for performance isolation.) We compute performance variability for the principal benchmark as *percent slowdown in the slow schedule*: measure the time it takes the principal benchmark to complete the 500 million instructions in the slow schedule, measure that time in the fast schedule, compute the difference as percent of the time in the fast schedule.

Figure 4 shows performance variability with the two schedulers. With the default scheduler (black bars) performance variability is substantial, ranging from 5% to

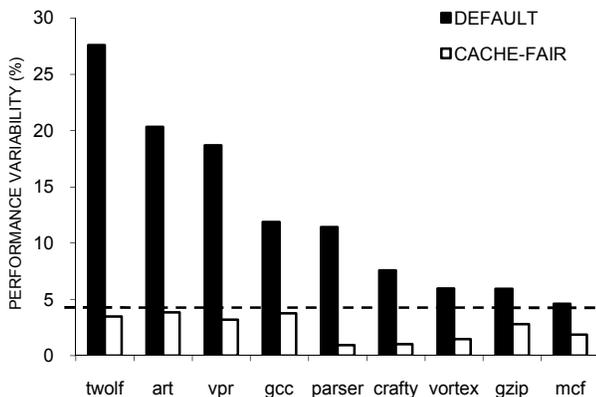


Figure 4. Performance variability with default and cache-fair scheduler. The dotted line is at 4%.

Table 2. Schedules for each benchmark

<i>Principal</i>	<i>Fast Schedule</i>	<i>Slow Schedule</i>
<i>art</i>	<i>art,crafty,crafty,crafty</i>	<i>art,mcf,mcf,mcf</i>
<i>crafty</i>	<i>crafty,vpr,vpr,vpr</i>	<i>crafty,mcf,mcf,mcf</i>
<i>gcc</i>	<i>gcc,vpr,vpr,vpr</i>	<i>gcc,mcf,mcf,mcf</i>
<i>gzip</i>	<i>gzip,crafty,crafty,crafty</i>	<i>gzip,mcf,mcf,mcf</i>
<i>mcf</i>	<i>mcf,gzip,gzip,gzip</i>	<i>mcf,crafty,crafty,crafty</i>
<i>parser</i>	<i>parser,crafty,crafty,crafty</i>	<i>parser,mcf,mcf,mcf</i>
<i>twolf</i>	<i>twolf,crafty,crafty,crafty</i>	<i>twolf,mcf,mcf,mcf</i>
<i>vortex</i>	<i>vortex,crafty,crafty,crafty</i>	<i>vortex,mcf,mcf,mcf</i>
<i>vpr</i>	<i>vpr,crafty,crafty,crafty</i>	<i>vpr,mcf,mcf,mcf</i>

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 allocation (e.g., *vpr*'s 19% slowdown in the slow schedule is explained by a 46% increase in the L2 miss rate, as compared to the fast schedule); but since the cache-fair scheduler accurately modeled the effects of unfair cache allocation on the IPC, it was able to successfully eliminate the variability in the *overall* performance by adjusting the principal thread's timeslice.

We stress that the scheduler does not change *how* the cache is shared and thus does not affect threads' individual IPCs. It eliminates performance variability by altering threads' CPU timeslices.

These results indicate that co-runner-dependent performance variability due to unfair cache allocation on CMPs can be effectively eliminated using system software, and thus costly hardware solutions may be avoided.

5.1.2. Effect on absolute performance

The cache-fair scheduler is expected to deliver different absolute performance for applications than the default scheduler. Applications that did not get their fair cache share are expected to take less time to complete with 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 with the two schedulers. Completion times are shown as ranges. Ranges denoted by white circles correspond to the default scheduler, ranges denoted by the

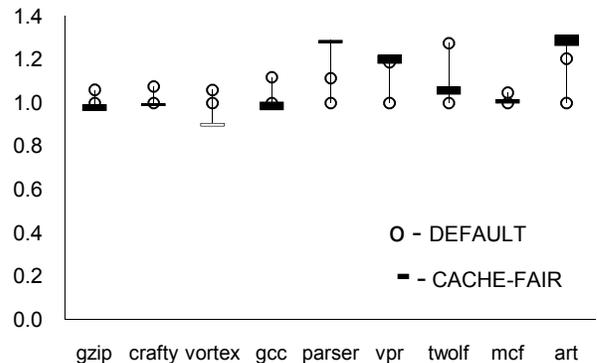


Figure 5. Ranges of normalized completion times with the two schedulers

black boxes correspond to the cache-fair scheduler. The upper the range boundary (either a circle or a box edge) indicates the completion time in the slow schedule (longer completion time), the lower boundary is the fast schedule completion (shorter completion time). The numbers are normalized to the completion time in the fast schedule with the default scheduler.

In that figure, we sort the benchmarks along the x-axis by their IPC in descending order. So high-IPC benchmarks appear on the left side of the figure, and low-IPC benchmarks on the right side. We see that five benchmarks completed more quickly with the cache-fair scheduler, while three completed more slowly. Note that high-IPC threads usually complete more quickly with the cache-fair scheduler than with the default scheduler (this is indicated by lower positions of black boxes in relation to white-dots). This is expected: high-IPC threads are usually low-miss-rate, so they get less than their fair share of the cache, forcing the cache-fair scheduler to increase their CPU timeslices and taking less time to complete, as a result.

5.1.3. Effect on the 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[27]. We do not use weighted speed-up, because that metric would not be affected by the cache-fair scheduler. Weighted speed-up is comprised of 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 compare the aggregate IPC for the two schedulers in Figure 6 (the IPC is normalized to the default scheduler). We show only the slow schedules; we omit the figure for the fast schedules, but summarize the results. For slow schedules, the aggregate IPC either increases (by 1-13% for 5 out of 9 schedules) or remains roughly unchanged (for 3 out of 9 schedules). The largest IPC increase (12%) was in the schedule with *crafty* as the principal benchmark. *Crafty* is a low-miss-rate thread, so it failed to achieve its fair IPC

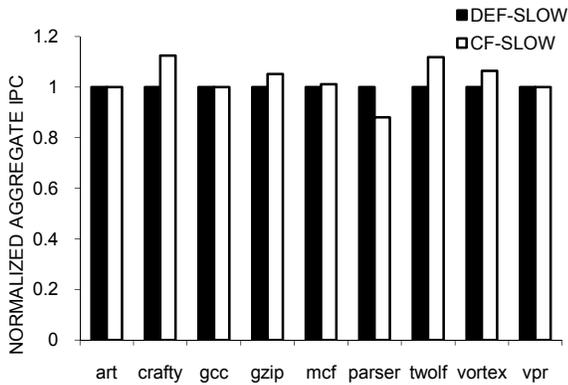


Figure 6. Aggregate IPC for slow schedules with the default scheduler and cache-fair scheduler

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

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

when running with *mcf*. As a result, *crafty*'s share of CPU time was increased relatively to *mcf*'s, so *crafty* ran more often than *mcf*. Since *crafty*'s individual IPC was higher than *mcf*'s, the aggregate IPC has increased.

In the schedule with *parser* as a principal benchmark the IPC with the cache-fair scheduler was 12% lower than with the default scheduler. We found, however, that *parser*'s fair miss rate was overestimated, so *parser*'s CPU timeslice was reduced more than necessary. Had the model estimated *parser*'s miss rate accurately, the aggregate IPC would not have been decreased. This stresses the importance of developing accurate techniques for estimating the fair cache miss rate.

For fast schedules, the aggregate IPC remained largely unchanged. For 8 out of 9 schedules, the IPC changed by at most +/- 3% in comparison with the default scheduler. In the schedule where *art* is 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.

In general, we saw that the effect on aggregate IPC depends on relative IPCs of the threads whose timeslices are being adjusted. Typically we will see that low-miss-rate (and high-IPC) threads will have their CPU shares increased, while high-miss-rate (and low-IPC) threads will have their CPU shares decreased, so the aggregate IPC will more likely increase than decrease.

5.1.4. Effect on best-effort threads

We now evaluate the cache-fair scheduler's performance effect on best-effort threads. Our experiment was designed to allow evaluation of the *worst-case* performance effect. Recall the schedules in Table 2 and note that there is only one cache-fair thread. Therefore,

Table 4. SPEC JBB with the two schedulers

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

CPU timeslice adjustments to the cache-fair thread would not be cancelled out by adjustments to other cache-fair threads: all complementary adjustments were done to the best-effort thread. Furthermore, note that there is only one best-effort thread in the schedule. Therefore, the scheduler was unable to balance CPU timeslice adjustments among multiple best-effort threads.

Our evaluation led us to the following conclusions: (1) the worst-case effect on best-effort threads is small; (2) to avoid large performance effects it is important to have more than one best-effort thread in the system.

Table 3 shows the slowdown (in comparison with the default scheduler) experienced by the best-effort thread in each schedule. The first and third columns show the schedules, identified by the principal benchmark, and the type of the schedule (slow or fast). The second and fourth columns show how much less CPU time (in percent) the best-effort thread received with the cache-fair scheduler in comparison with the default scheduler. Positive values indicate slowdown, negative values indicate speed-up.

On average, the cache-fair algorithm has a negligible (less than 1%) performance effect on best-effort threads. In about one third of the schedules, the best-effort thread experienced a speed-up. There is only one case when the best-effort thread was significantly penalized (by 27% in *gcc-slow* schedule). Had there been more than one best-effort thread in the system, however, the slowdown would have been smaller.

5.2. Experiments with database workloads

We describe our experiments with two database benchmarks: SPEC JBB and TPC-C. (We use our own implementation of TPC-C over Oracle Berkeley DB[2]). We compare performance isolation for these benchmarks with the two schedulers. We have two sets of experiments: one where SPEC JBB is the principal, and another where TPC-C is the principal. Like in the previous section we run each principal benchmark in the slow and in the fast schedule. 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 the IPC.

SPEC JBB and TPC-C emulate database activities of an order-processing warehouse. The benchmarks can be run with databases of various sizes; the size is determined by the number of warehouses. The standard number of warehouses is ten. Because our simulator had an upper limit for physical memory (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 the measurement.

5.2.1. SPEC JBB

In the slow schedule, SPEC JBB’s co-runners are 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 are TPC-C configured with one warehouse (TPC-C_1WH) and *twolf* (used as the best-effort thread). We pin threads to CPU cores in the following manner:

Schedule	Core 0	Core 1
<i>Slow</i>	SPEC JBB, <i>twolf</i>	TPC-C_5WH
<i>Fast</i>	SPEC JBB, <i>twolf</i>	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, co-runner-dependent difference in the transaction rate is 9%. With cache-fair scheduler, the difference is reduced to 1%.

In the fast schedule, SPEC JBB completes fewer transactions per second with the cache-fair scheduler than with the default scheduler. This happens because SPEC JBB occupies more than its fair share of the cache (61% as opposed to 50%), and so the cache-fair scheduler reduces its CPU share.

In the slow schedule, SPEC JBB achieves roughly the same transaction rate with the cache-fair scheduler as with the default scheduler. When SPEC JBB runs with TPC-C_5WH, it uses roughly half the cache (in fact, we measured it to be exactly 50% of the cache). The cache-fair scheduler, therefore, does not adjust SPEC-JBB’s CPU share, and as a result SPEC JBB achieves the same performance as with the default scheduler.

5.2.2. TPC-C

In this experiment the principal benchmark is TPC-C configured with two warehouses. We run it with two different co-runners: in the slow schedule with SPEC JBB, and in the fast schedule with Sphinx[3]. (Sphinx is a speech recognition benchmark, representative of the workload used in an online voice recognition server in a call center). The assignment of threads to CPUs looks as follows:

Schedule	Core 0	Core 1
<i>Slow</i>	TPC-C_2WH, <i>twolf</i>	SPEC JBB
<i>Fast</i>	TPC-C_2WH, <i>twolf</i>	Sphinx

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

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

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

In the slow schedule TPC-C runs more quickly with the cache-fair scheduler than with the default scheduler. In the slow schedule, TPC-C occupies only 38% of the cache, indicating that the high-miss-rate co-runner SPEC JBB reduced its fair cache share. The cache fair scheduler gave TPC-C an extra 14% of CPU time to compensate for unfair cache allocation.

5.3. Comparison to cache partitioning

Cache partitioning is a hardware alternative for addressing performance isolation, because it eliminates unfair cache allocation. We compared our solution with cache partitioning and found that our solution reduces co-runner-dependent performance variability to a greater extent than cache partitioning.

We implemented way-partitioning of the L2 cache in our simulated processor, and ran the slow and fast schedules presented in Table 2 on our simulated processor enabled with cache partitioning. The cache was equally partitioned among the co-runners. 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, as we explained earlier. To confirm that co-runner-dependent memory contention was the reason, we simulated infinite memory bus bandwidth to eliminate the variability in the bus contention. In that configuration, partitioning did eliminate performance variability. In reality, memory bandwidth is a highly contended resource on CMPs [29], so taking that contention into account is necessary in order to reduce co-runner-dependent performance variability. The cache-fair algorithm accounts for that contention and thus improves performance isolation more effectively than cache partitioning.

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 [9,10,12,23,30]. The advantage of hardware solutions is that they can address performance isolation *directly*, and thus require fewer or no modifications to the

operating system. As we have shown however, simple hardware solutions, such as cache partitioning, do not address the problem effectively, while 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. Accordingly, to the best of our knowledge, none of the 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., aiming to select the optimal co-runner for a thread. Co-scheduling has been used to improve performance [22,27] and performance isolation [15]. The key advantage of co-scheduling over our method is that co-scheduling can actually *force* cache allocation to be more fair by selecting the “right” co-runner for the thread. On the other hand, if the right co-runner cannot be found, co-scheduling does not achieve its goal. Our cache-fair scheduling algorithm does not have that limitation. Better scalability is another advantage of our method over co-scheduling. Co-scheduling requires coordination of scheduling decisions among the processor’s cores and may thus limit scheduler’s scalability if the number of cores is large. The cache-fair algorithm is, to the best of our knowledge, the first CMP-targeted scheduling algorithm that does not require co-scheduling. This is a valuable feature given that future processors will be built with more and more cores, and inter-core communication could become a serious issue.

7. Summary

We presented the cache-fair scheduling algorithm that improves performance isolation on CMPs. We evaluated it using the 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 does not rely on finding optimal co-schedules, which puts it at an advantage over existing software solutions. Furthermore, it is less costly, more flexible and more effective than state-of-the-art hardware solutions.

8. References

- [1] Consolidation and Virtualization. <http://www.sun.com/datacenter/consolidation/index.jsp>
- [2] Berkeley DB. Oracle Embedded Database. <http://www.oracle.com/database/berkeley-db.html>
- [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] Rajeev Balasubramonian, Sandhya Dwarkadas, and David Albonese. Dynamically managing the communication-

- parallelism trade-off in future clustered processors. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003
- [6] 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
- [7] 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
- [8] 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
- [9] 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
- [10] 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
- [11] 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
- [12] 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
- [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] W. Feller. An Introduction to Probability Theory and Its Applications. *John Wiley and Sons*, 1968
- [15] 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
- [16] 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
- [17] Poonacha Kongetira. A 32-way Multithreaded SPARC(R) Processor. In *Proceedings of the 16th Symposium On High Performance Chips (HOTCHIPS)*, 2004
- [18] 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
- [19] R. Matick, T. J. Heller, and M. Ignatowski. Analytical Analysis of Finite Cache Penalty and Cycles per Instruction of a Multiprocessor Memory Hierarchy Using Miss Rates and Queuing Theory. *IBM Journal Of Research And Development*, 45(6):819-842, 2001
- [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 Technical Report 2000-04-02*, 2004
- [23] 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
- [24] 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
- [25] 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
- [26] 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
- [27] Allan Snaveley 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
- [28] Yan Solihin, V. Lam, and Josep Torrellas. ScalTool: Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors. In *Proceedings of the 1999 Conference on Supercomputing*, 2006
- [29] Lawrence Spracklen and Santosh G. Abraham. Chip Multithreading: Opportunities and Challenges. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005
- [30] 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
- [31] Harvey J. Wasserman, Olaf M. Lubeck, Yong Luo, and Frederico Bassetti. Performance Evaluation of the SGI Origin2000: A Memory-Centric Evaluation of LANL ASCI Applications. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, 1997