CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Harvard University - DASH

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Cache-Fair Thread Scheduling for Multicore Processors

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

Citation	Fedorova, Alexandra, Margo Seltzer, and Michael D. Smith. 2006. Cache-Fair Thread Scheduling for Multicore Processors. Harvard Computer Science Group Technical Report TR-17-06.
Citable link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:25686821
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Cache-Fair Thread Scheduling for Multicore Processors

Alexandra Fedorova, Margo Seltzer
and Michael D. Smith

TR-17-06

Computer Science Group
Harvard University
Cambridge, Massachusetts

Cache-Fair Thread Scheduling for Multicore Processors

Alexandra Fedorova†‡, Margo Seltzer† and Michael D. Smith†
†*Harvard University*, ‡*Sun Microsystems*

ABSTRACT

We present a new operating system scheduling algorithm for multicore processors. Our algorithm reduces the effects of unequal CPU cache sharing that occur on these processors and cause unfair CPU sharing, priority inversion, and inadequate CPU accounting. We describe the implementation of our algorithm in the Solaris operating system and demonstrate that it produces fairer schedules enabling better priority enforcement and improved performance stability for applications. With conventional scheduling algorithms, application performance on multicore processors varies by up to 36% depending on the runtime characteristics of concurrent processes. We reduce this variability by up to a factor of seven.

1. INTRODUCTION

In the recent years, the hardware industry has moved from discussion to full-scale production of *multicore processors* – processors that have multiple processing cores inside them [18]. Multiple processing cores enable multiple application threads to run simultaneously on a single processor. Multicore technology, with its promise of improved power efficiency and increased hardware utilization, has been embraced by the industry: AMD, Fujitsu, IBM, Intel and Sun Microsystems are shipping multicore systems and have announced plans to release future models [1-6].

Conventional CPU schedulers make assumptions that do not apply on multicore processors: they assume that the CPU is a single, indivisible resource and that if threads are granted equal time slices, those threads will share the CPU equally. On multicore processors, concurrently running threads, or *co-runners*, often share a single second-level (L2) cache, and cache allocation is controlled by the hardware [1-4]. Cache sharing depends solely on the cache needs of the co-runner(s), and unfair cache sharing occurs often (see Figure 1). A thread's cache occupancy affects its cache miss rate, and, as a result, impacts the rate at which the thread retires instructions. Therefore, a thread's CPU performance significantly varies depending on its co-runner. This *co-runner-dependent performance variability* can create the following problems:

(1) Unfair CPU sharing: Conventional schedulers ensure that equal-priority threads get equal shares of the CPU. On multicore processors a thread's share of the CPU, and thus its forward progress, is dependent both upon its time slice and the cache behavior of its co-runners. Kim et al showed that a SPEC CPU2000 [12] benchmark *gzip* runs 42% slower with a co-runner *art*, than with *apsi*, even though *gzip* executes the same number of cycles in both cases [7].

(2) Poor priority enforcement: A priority-based scheduler on a conventional processor ensures that elevating a job's priority results in greater forward progress for this job. On a multicore processor, if the high-priority job is scheduled with 'bad' co-runners, it will experience inferior rather than superior performance.

(3) Inadequate CPU accounting: On grid-like systems where users are charged for CPU hours [37] conventional scheduling ensures that processes are billed proportionally to the amount of computation accomplished by a job. On multicore processors, the amount of computation performed in a CPU hour varies depending on the co-runners (we observed variability of up to 36%, and other studies reported similar numbers [7,16]), so charging for CPU hours is not appropriate.

Clearly then, co-runner-dependent performance variability requires that we rethink fair sharing and

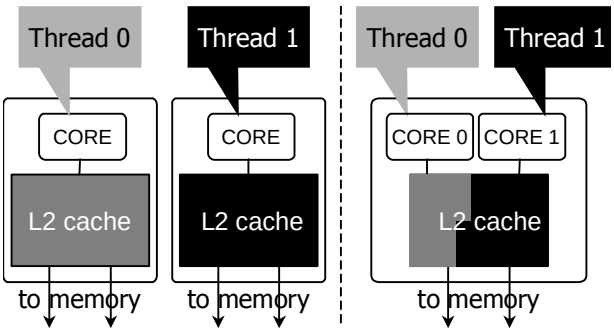

Figure 1. Conventional processor (left) and multicore processor (right). Cache coloring shows the thread's working set. Each thread's working set is large enough to populate the entire L2 cache, as shown on the left. However, when the threads become co-runners on a multicore processor (right), the cache is not equally allocated among the threads.

timeslicing for multicore processors. To achieve fair sharing on these processors, L2 cache allocation must be considered. This problem is actually more difficult than fair sharing in a shared-memory multiprocessor, where a thread's performance similarly depends on how much of the shared memory it gets [25]. The fundamental difference is that the operating system can observe and control multiprocessor memory allocation while L2 cache allocation is completely opaque to the operating system.

We present a new scheduling algorithm, the *cache-fair algorithm*, that addresses unequal cache allocation and reduces co-runner-dependent performance variability. This algorithm redistributes CPU time to threads to account for unequal cache sharing: if a thread's performance decreases due to unequal cache sharing it gets more time, and vice versa. The challenge in implementing this algorithm is determining how a thread's performance is affected by unequal cache sharing using limited information from the hardware. Our solution uses runtime statistics and analytical models and does not require new hardware structures or operating system control over cache allocation.

We implemented our algorithm as an extension to the Solaris™ 10 operating system and evaluated it on a full-system hardware simulator [11] of the UltraSPARC® T1 [4]. We demonstrate that our algorithm significantly reduces co-runner-dependent performance variability: by at least a factor of two and by as much a factor of seven in the cases where the variability is significant. Co-runner-dependent performance is the result of unequal cache sharing, and by eliminating it, we address the problems caused by unequal cache sharing:

(1) Unfair CPU sharing: With our algorithm, an application achieves predictable forward progress regardless of its co-runners. The effects of unfair cache sharing are negated.

(2) Poor priority enforcement: Our algorithm ensures that a thread makes predictable forward progress regardless of its co-runner. Therefore, elevating a thread's priority results in greater forward progress, and vice versa, just like on conventional processors. Priorities are properly enforced.

(3) Inaccurate CPU accounting: Our algorithm reduces dependency of a thread's performance on its co-runner. Charging for CPU hours is appropriate, because the amount of work accomplished by a thread is proportional to the CPU hours paid for by the customer and is not affected by the co-runner.

Multicore processors offer advantages of power efficiency, improved hardware utilization and reduced cost due to component duplication [2,4,5,17]. The exploding popularity of these processors suggests that they will become the dominant CPU type. The performance effects of unequal L2 cache sharing will increase in the future: Unequal cache sharing affects the cache miss rate, an increased cache miss rate causes more memory accesses, and the cost of memory access in terms of processor cycles grows at the rate of 50% per year [38]. Therefore, it is critical to address this problem now, so we can enjoy the benefits of multicore processors without losing good systems properties developed through years of research and practice.

The rest of the paper is organized as follows: In Section 2 we describe the algorithm; in Section 3 we describe the analytical models used in our algorithm; in Section 4 we describe our implementation; and in Section 5 we evaluate it. We discuss related work and alternative solutions in Section 6, and conclude in Section 7.

2. ALGORITHMIC OVERVIEW

Our scheduling algorithm reduces co-runner-dependent performance variability, diminishing the effect of unfair cache allocation. We reduce co-runner-dependent performance variability by redistributing CPU time such that a thread runs as quickly as it would with an equally shared cache, regardless of its co-runners.

More specifically, we make the thread's CPU latency, i.e. the time to complete a logical unit of work (such as 10 million instructions) equal to its CPU latency under equal cache sharing. A thread's CPU latency is the product of its *cycles per instruction* (CPI) (how efficiency it uses the CPU cycles it's given) and its share of CPU time (how long it runs on the CPU). Co-runner-dependence affects a thread's CPI, because, for example, a cache-starved thread incurs more memory stalls, exhibiting a higher CPI. An OS scheduler, on the other hand, influences the thread's CPU latency by adjusting its share of CPU time.

Figure 2a) illustrates co-runner dependency. There are three threads (A though C) running on a dual-core processor. Thread A is cache-starved when it runs with Thread B and suffers worse performance than when it runs with Thread C. In the figure, a box corresponds to each thread. The height of the box indicates the amount of cache allocated to the thread. The width of the box indicates the CPU quantum allocated to the thread. 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.

Figure 2b) depicts the imaginary ideal scenario: the cache is shared equally, as indicated by the equal heights of all the boxes. Our goal is to complete the same amount of work (the shaded area) in the same time (the length along the X-axis) it takes in the ideal scenario.

In Figure 2a) Thread A completes less work per unit of time than it does in Figure 2b), because it does not get an equal share of the cache when running with Thread B. As a result, it takes longer to complete the same amount of work; its latency is longer than its latency under equal cache sharing.

Figure 2c) shows how the cache-fair algorithm eliminates dependency of Thread A's performance on Thread B. By giving Thread A more CPU cycles, it makes Thread A retain the same latency as under equal cache sharing (Figure 2b). Note that the adjustment to Thread A's CPU quantum is *temporary*. Once the thread catches up with its work, its quantum is restored to its initial value.

Giving more CPU time to one thread takes away time from another thread (or other threads in general),

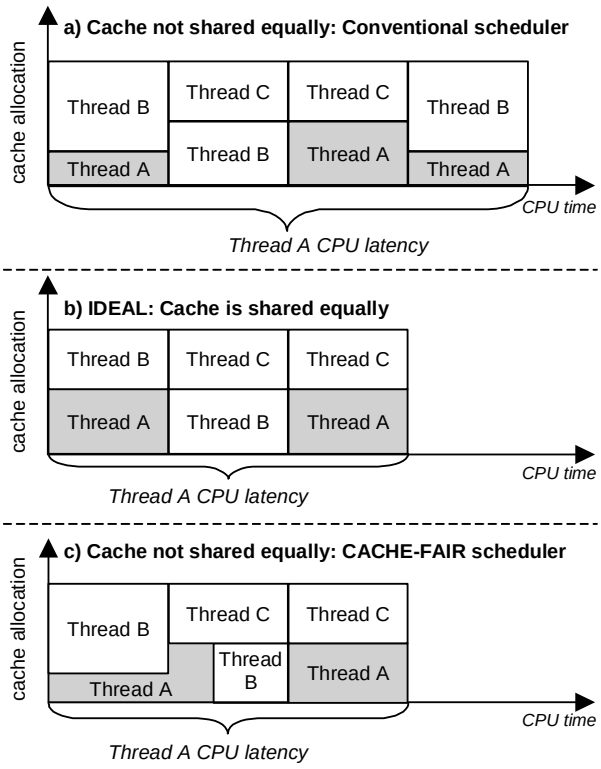

Figure 2. Thread A's CPU latency when the cache is not shared equally and a conventional scheduler is used (a), when the cache is ideally shared equally and a conventional scheduler is used (b), and when the cache is not shared equally and the cache-fair scheduler is used (c). Thread A takes longer to complete the shaded work in (a) than in (b) or (c).

and vice versa. Figure 2c) illustrates that our algorithm gives more time to Thread A at the expense of Thread B. Our algorithm requires, therefore, that there are two classes of threads in the system: the *cache-fair* class and the *best-effort* class. In our example, Thread A is a cache-fair thread, and Thread B is a best-effort thread. The scheduler reduces co-runner-dependent performance variability for threads in the cache-fair class, but not for threads in the best-effort class. Our algorithm, however, avoids imposing significant performance penalties on best-effort threads.

The user specifies the job's class in the same way she specifies a job's priority. We expect that user jobs will be in the cache-fair class by default and that background system jobs will fall into the best-effort class. Because modern systems typically run dozens of background threads (running a single-threaded "hello world" program in Java requires nine threads!), the requirement that best-effort threads exist in the system is easily satisfied.

The cache-fair scheduling algorithm does not establish a new CPU sharing policy but helps *enforce* existing policies. For example, if the system is using a fair-share policy, the cache-fair algorithm will make the cache-fair threads run as quickly as they would if the cache were shared equally, given the number of CPU cycles they are entitled to under the fair-share policy.

The key part of our algorithm is correctly computing the adjustment to the thread's CPU quantum. We follow a four-step process to compute the adjustment:

1. Determine a thread's *fair L2 cache miss rate* – a miss rate that the thread would experience under equal cache sharing. We designed a new efficient, online analytical model to estimate this rate (Section 3.1).
2. Compute the thread's *fair CPI rate* – the cycles per instruction under the fair cache miss rate. We use an existing analytical model to perform this computation (Section 3.2).
3. Estimate the *fair number of instructions* – the number of instructions the thread would have completed under the existing scheduling policy if it ran at its fair CPI rate (divide the number of cycles by the fair CPI). Then measure the actual number of instructions completed.
4. Estimate how many CPU cycles to give or take away to compensate for the difference between the actual and the fair number of instructions. Adjust the thread's CPU quantum accordingly.

The algorithm works in two phases:

Reconnaissance phase: The scheduler computes the fair L2 cache miss rate for each thread.

Calibration phase: A single calibration consists of computing the adjustment to the thread's CPU quantum and then selecting a thread from the best-effort class whose CPU quantum is adjusted to offset the adjustment to the cache-fair thread's quantum. Calibrations are repeated periodically. We explain how we select best-efforts thread in Section 4.

New threads in the system are assigned CPU quanta according to the default scheduling policy. For new threads in the cache-fair class, the scheduler performs the reconnaissance phase and then the calibration phase. We present the implementation details in Section 4.

The challenge in implementing this algorithm is that in order to correctly compute adjustments to the CPU quanta we need to determine a thread's fair CPI ratio using only limited information from hardware counters. Hardware counters allow measuring threads' runtime statistics, such as the number of cache miss and retired instructions, but they do not tell us how the cache is actually shared and how the thread's CPI is affected by unequal cache sharing. We estimate the fair CPI ratio using both information from hardware counters and analytical models. These models are the subject of the next section.

3. ANALYTICAL MODELS

The key requirement for analytical models we employ is that they can be efficiently used at runtime and not require any pre-processing of the workload or any advance knowledge about it. We now present the analytical models used to estimate a) the fair L2 cache miss rate and b) fair CPI ratio.

3.1 Fair L2 cache miss rate

The fair L2 cache miss rate is the number of *misses per cycle* (MPC) that would be generated by a thread if the cache were shared equally. We need to determine a thread's fair cache miss rate in order to estimate its fair CPI ratio. Modeling cache miss rates is a well-studied area [7,13,20,26,27,30-32,36], but the existing models require inputs that are expensive to obtain at runtime. Our model is not a general-purpose cache-model, but it produces accurate estimates of the fair cache miss rate and can be used efficiently at runtime, and is thus well suited for our needs.

Our approach is based on an empirically derived observation that if the co-runners have similar cache miss rates they share the cache roughly equally. So if co-runners A and B experience similar miss rates, they

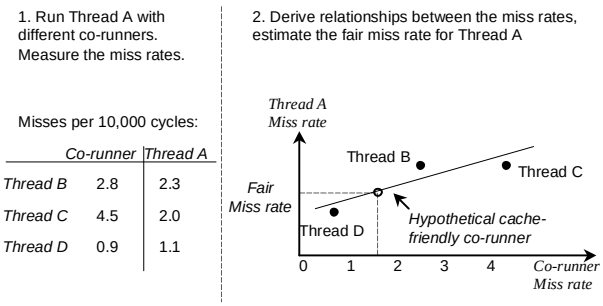

Figure 3. Estimating the fair cache miss rate for Thread A.

share the cache equally and they each experience their fair miss rate. In this case we say that A and B are *cache-friendly* co-runners. To estimate the fair cache miss rate, for example, for Thread A on a dual-core CPU, one could run Thread A with different co-runners until finding its cache-friendly co-runner. This is not practical, however, because this may take an unbounded amount of time. Instead we run Thread A with several different co-runners and derive the relationship between Thread A's miss rate and its co-runner. We then use this relationship to estimate the miss rate Thread A would experience with a "hypothetical" cache-friendly co-runner; this miss rate is Thread A's fair miss rate.

Figure 3 illustrates this idea. Step 1 shows the miss rates measured as Thread A runs with different co-runners. Step 2 shows that we derive a linear relationship between the miss rate of Thread A and its co-runners. We use the corresponding linear equation to compute Thread A's miss rate when running with a hypothetical cache-friendly co-runner – its fair miss rate.

We now justify the assumptions in this approach: (1) Cache-friendly co-runners have similar cache miss rates, and (2) the relationship between co-runners' miss rates is linear.

Co-runners with similar miss rates share cache equally: If we assume that a thread's L2 cache accesses are uniformly distributed in the cache, then we can model cache replacement as a simple case of the balls in bins problem [41]. 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, a 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 [40]. Thus, two threads with the same L2

cache miss rate (balls being tossed at the same rate) will share the cache equally. We verified this empirically by analyzing how co-runners' cache miss rates correspond to their cache allocations (we instrumented our simulator to count per-thread cache allocations). This theory applies if cache requests are distributed uniformly across the cache. We measured how cache request made by SPEC CPU2000 benchmarks are distributed among the cache banks on a simulated machine with a four-banked L2 cache (we instrumented the simulator to collect distribution data). We found that for most benchmarks, the distribution was close to uniform: the difference between the access frequencies of two different cache banks was at most 15%. There were rare exceptions, *gzip*, *twolf*, and *vpr*, where such difference reached 50%, but even in those cases, the "hot-bank" effect was not significant enough to affect our model's accuracy, as we will demonstrate in Figure 4.

Relationship between co-runners' miss rates is linear. We chose nine benchmarks from the SPEC CPU2000 suite with different cache access patterns and ran them in pairs on our simulated dual-core processor. We ran each benchmark in several pairs. We analyzed the relationship between the miss rate of each benchmark and the miss rates of its co-runners and found that a linear equation approximated these relationships better than other simple functions.

The expression for the relationship between the co-runners' miss rates for a processor with *n+1* cores is:

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

where T is a thread for which we compute the fair miss rate, Ci is the ith co-runner, n is the number of co-runners, and a and b are the linear equation coefficients. Thread T experiences its fair cache miss rate *FairMissRate(T)* when all concurrent threads experience the same miss rate:

$FairMissRate(T) = MissRate(T) = MissRate(Ci)$, for all i. Equation (1) can be expressed as:

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

and the expression for *FairMissRate(T)* is:

$$FareMissRate(T) = \frac{b}{1 - a * n} \qquad (2).$$

The cache-fair scheduler dynamically derives coefficients for Equation 1 for each cache-fair thread at runtime and then estimates its fair cache miss rate. Section 4 provides implementation details.

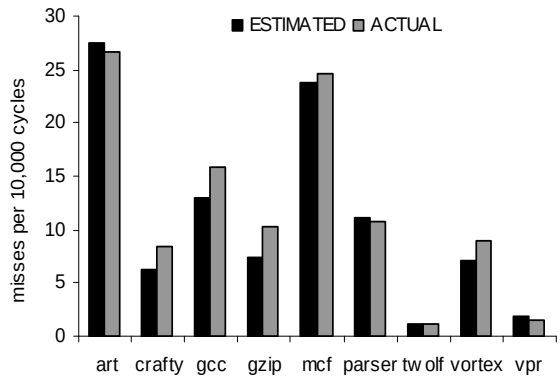

Figure 4. Estimated vs. the actual fair cache miss rate.

We evaluate the accuracy of the estimated fair miss rates produced using our model by comparing it with the actual fair miss rates. We compute the actual fair miss rate for a thread by running this thread and a co-runner on a simulated dual-core machine with an equally partitioned cache (an equally partitioned cache ensures equal cache sharing). We compute the estimated fair miss rate by running each thread with several different co-runners, deriving the coefficients for Equation 1 by means of linear regression, and then using Equation 2. We validated the model for workloads where there was no data sharing among the co-runners. Studying whether data sharing affects the model is the subject of ongoing work.

Figure 4 shows how the estimated fair miss rate compares to the actual miss rate. The names of the SPEC CPU2000 benchmarks are on the X-axis; the Y-axis shows the cache miss rate for the actual and estimated fair miss rates for each benchmark. The estimated miss rates closely approximate the actual miss rates. The source of errors in the model is the implicit assumption that cache misses occur due to insufficient capacity, rather than changes in the working set or mapping conflicts. To account for these effects one would need to know about the workload's memory access patterns; this information is expensive to obtain at runtime. Besides, on multicore processors, where cache is scarce, capacity misses dominate the miss rate. Our model provides a good combination of accuracy and efficiency, which makes it practical to use inside an operating system. We describe its implementation in Section 4.

3.2 Fair CPI ratio

The fair CPI ratio is the number of cycles per

instruction achieved by a thread under equal cache sharing. It is used to compute the adjustment to cache-fair threads' CPU quanta. To estimate it, we use an existing analytical model [29] adapted to our CPU architecture [30]. We describe the general form of the model and omit the details. The model has the form:

$$CPI = IdealCPI + L2CacheStalls \quad (3),$$

where *IdealCPI* is the CPI when there are no L2 cache misses, and *L2CacheStalls* is the per-instruction stall time due to handling L2 cache misses.

The expression for fair CPI is:

$$FairCPI = IdealCPI + FairL2CacheStalls \quad (4),$$

where *FairCPI* is the CPI when the thread experiences its fair cache miss rate. To estimate *FairCPI*, we need to determine (1) *IdealCPI* and (2) *FairL2CacheStalls:*

Computing IdealCPI: We compute it using Equation 3. At runtime, we measure the thread's actual *CPI* and the statistics needed to compute the thread's actual *L2CacheStalls*. Subtracting *L2CacheStalls* from *CPI* gives us the thread's *IdealCPI*.

Computing FairL2CacheStalls: L2 cache stall time is a function of the cache miss rate, the per-miss memory stall time *MemoryStalls* (including memory latency and memory-bus delay), and the store buffer stall time[1] *StoreBufferStalls*:

$$L2CacheStals = F(MissRate, MemoryStalls, StoreBufferStalls) \quad (5).$$

FairL2CacheStalls is computed using the fair cache miss rate *FairMissRate*:

$$FairL2CacheStalls = F(FairMissRate, MemoryStalls, StoreBufferStalls) \quad (6).$$

FairMissRate is estimated as described in Section 3.1. We delay discussing estimation of *MemoryStalls* and *StoreBufferStalls* until Section 4, which describes how the models for fair cache miss rates and fair CPI are used inside the cache-fair scheduler.

[1] A store buffer is the queue at the processor that allows non-blocking writes: a writing thread places a value in the store buffer and continues without waiting for the write to propagate down the memory hierarchy. If the store buffer becomes full, the thread stalls until space becomes available in the store buffer.

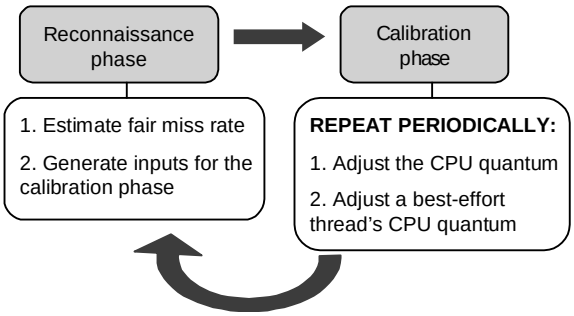

Figure 5. The structure of the cache-fair algorithm.

4. IMPLEMENTATION

Recall that the cache-fair scheduler enforces an existing scheduling policy. We implemented our cache-fair scheduler in the Solaris 10 operating system on top of its fixed-priority scheduling policy (each thread has a fixed priority). Like conventional schedulers, our scheduler runs on every system clock tick for each running thread. It performs the housekeeping needed by the baseline fixed-priority policy for that thread. It also performs cache-fair-specific processing, depending on the phase that the thread is undergoing.

Each cache-fair thread goes through two phases, the reconnaissance phase and the calibration phase. The logical structure of these phases was described in Section 2 and is summarized in Figure 5. In this section, we describe how the phases are implemented, how often they are repeated and, in particular, how we obtain the inputs for the analytical models used inside the algorithm. Most inputs are obtained using hardware performance counters, which allow measuring thread-specific runtime statistics [22]; such counters are typically available on modern processors [23,24], so our implementation is not limited to a specific hardware architecture.

4.1 Reconnaissance phase

The goal of the reconnaissance phase is to estimate a thread's fair cache miss rate and generate inputs needed by the fair CPI model.

Assume that a cache-fair thread *T* is in its reconnaissance phase. The cache-fair scheduler first measures the miss rates of *T* and the different co-runners with which it runs. The scheduler does not force *T* to run with specific co-runners, but observes any co-runner combinations that appear on the processor.

Whenever *T* appears on the CPU with a new group of co-runners, we reset the hardware counters.

When any of the co-runners goes off the processor, we record the miss rate of *T* and the overall miss rate of its co-runners, generating a *data point*. By the end of the reconnaissance phase, we have approximately ten such data points. We perform linear regression analysis on these data points, as explained in Section 3.1, and generate linear equation coefficients for Equation 1. Then, using Equation 2, we estimate the fair cache miss rate for thread *T*. Implementations of linear regression analysis usually require floating-point operations, which are not permitted inside the Solaris kernel, so we implemented linear regression using only integer operations.

During the reconnaissance phase, we also generate values for *MemoryStalls* and *StoreBufferStalls* needed as inputs for the fair CPU model used in the calibration phase. Recall that these values are used to compute *FairL2CacheStalls* (Section 3.2, Equation 6) and correspond to the memory and store buffer stall times that a thread would attain if it experienced its fair cache miss rate. Since we cannot measure these values directly we estimate them: we measure *T*'s actual memory and store buffer stall times as *T* runs in the reconnaissance phase, and then express them as linear functions of *T*'s cache miss rate. We substitute *T*'s fair cache miss rate into these equations to compute the store buffer and memory stall times for the *FairL2CacheStalls* model.

4.2 Calibration phase

Upon completion of the reconnaissance phase thread *T* enters the calibration phase, where the scheduler periodically re-distributes, or *calibrates*, the CPU time. A single calibration involves adjusting *T*'s CPU quantum, based on how its actual CPI ratio differs from it fair CPI ratio, and selecting a best-effort thread whose quantum is adjusted correspondingly.

Calculating the adjustment to the CPU quantum, described in Section 2, requires knowing *T*'s fair CPI ratio. In Section 3.2, we explained how to estimate it. The required inputs are: (1) the actual *CPI*, (2) the actual L2 cache stall time (*L2CacheStalls*), and (3) the L2 cache stall time corresponding to *T*'s fair cache miss rate (*FairL2CacheStalls*). Here is how we obtain these inputs:

1. Actual CPI is computed by taking a ratio of the number of CPU cycles *T* has executed and the number of instructions *T* retired.
2. The actual L2 cache stall time is estimated using the model expressed by Equation 5. The required inputs, i.e., the cache miss rate and the memory and store buffer stall times, are obtained from the hardware counters
3. *FairL2CacheStalls* is estimated using the model expressed by Equation 6. The inputs are obtained using the inputs from the reconnaissance phase.

Once *T*'s fair CPI ratio is known, the scheduler compares it to *T*'s actual CPI and estimates the temporary adjustment to *T*'s CPU quantum, so that by the end of its next quantum, *T* completes the number of instructions corresponding to its fair CPI. If *T* has been running faster than it would at its fair CPI, its quantum is decreased, and the quantum of some best-effort thread is increased correspondingly. In this case, we compensate the best-effort thread that has suffered the most performance penalty by increasing its quantum. If *T* has been running slower than it would at its fair CPI, its quantum is increased, and a best-effort thread's quantum is decreased. In this case, we select the best-effort thread that has suffered the least performance penalty. The adjustment to the best-effort thread's CPU quantum is temporary: once the thread has run with the adjusted quantum, its quantum is reset to its original value. Section 5 evaluates the performance effects on best-effort threads.

4.3 Phase duration and repetition

The reconnaissance phase needs to be sufficiently long to capture the long-term properties of the workload's cache access patterns. Upon analysis of temporal variation of L2 cache access patterns for nine SPEC CPU2000 benchmarks, we found that most workloads (eight out of nine) had stable cache access patterns over time, and though there was occasional short-term variability, any window of 100 million instructions captured the long-term properties of the workload. Accordingly, we set the duration of the reconnaissance phase to 100 million instructions.

After an initial reconnaissance phase, a cache-fair thread enters the calibration phase. In the calibration phase, the adjustments to the thread's CPU quantum are performed every ten million instructions (roughly every clock tick). Frequent adjustments allow the cache-fair algorithm to be more responsive.

A reconnaissance phase is repeated occasionally to account for changes in the thread's cache access patterns, which would necessitate the need to re-compute its fair cache miss rate. From our analysis of the benchmarks' temporal behavior, we found that most workloads change their L2 cache access patterns gradually and infrequently, so it is appropriate to choose a fixed-sized repetition interval for the reconnaissance phase. We set it to one billion instructions.

Statically setting phase repetition intervals works

Processing 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	256KB, four-way banked, eight-way set associative, unified instruction and data.
Memory bus	4 GB/s peak bandwidth

Table 1. Architectural parameters of the simulated machine

for most workloads, but for less-common workloads with frequently changing cache-access patterns, the frequency of reconnaissance phase needs to be determined dynamically. Standard techniques for phase detection [19] can be used to detect when the workload has changed its cache access patterns, and the reconnaissance phase can be performed every time such a change is detected.

5. EVALUATION

We evaluate our algorithm by demonstrating that it significantly reduces co-runner-dependent performance variability for the cache-fair threads. Co-runner-dependent performance variability is the effect of unfair cache sharing, and by eliminating it, we address the problems resulting from unfair cache sharing. We also show that our algorithm has only a small performance penalty on best-effort threads. We begin by introducing our experimental setup and the workload.

5.1 Experimental setup

We use Sun Microsystems' multicore processor simulator [11] for the UltraSPARC T1 architecture [4]. The number of cores, the type of core (single-threaded or multithreaded), and the cache size are configurable. We configure our simulated machine with two single-threaded cores and a shared L2 cache. Table 1 summarizes the configuration parameters. This is a full-system simulator: it runs the complete operating system and applications unmodified.

Our workload consists of benchmarks from the SPEC CPU2000 suite [12]. This benchmark suite is commonly used for CPU studies and has been recently updated to include programs with large cache working sets, to reflect trends in modern commercial applications. This suite is built of real applications, with different cache access patterns [21]. For our experiments, we picked nine benchmarks that represent a variety of popular workloads:

art – image recognition
crafty – game playing: chess
gcc – a compiler
gzip – a compression utility
mcf – combinatorial optimization
parser – word processing
twolf – place and route simulator
vortex – object-oriented database
vpr – FPGA circuit placement and routing

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

Table 2. The schedules for each benchmark

We run each benchmark in two scenarios: in the *slow schedule* case, the benchmark's co-runners have high cache requirements; and in the *fast schedule* case, the co-runners have low cache requirements. In a slow schedule, a thread is likely to run more slowly than in the fast schedule, hence the naming of the two cases. In each of the schedules, the *principal* benchmark belongs to the cache-fair class and runs with three other application threads, one of which is in the best-effort class. Table 2 shows the schedules for each benchmark.

We run each schedule until the principal benchmark completes a segment of one hundred million instructions in the calibration phase; running to completion would take weeks on a simulator. We fast-forward the simulation to the point where all benchmarks enter the main processing loop, and then perform the detailed simulation. All benchmarks are CPU-bound.

5.2 Co-runner-dependent performance variability

We compare the time that it takes the principal benchmark to complete its work segments in the fast and slow schedules. We refer to this quantity as *completion time*. When running with a conventional fixed-priority scheduler, the difference between completion times in the fast and in the slow schedules is large, but when running with our cache fair scheduler, it is significantly smaller.

Figure 6 demonstrates normalized completion times with the conventional scheduler. The co-runner-

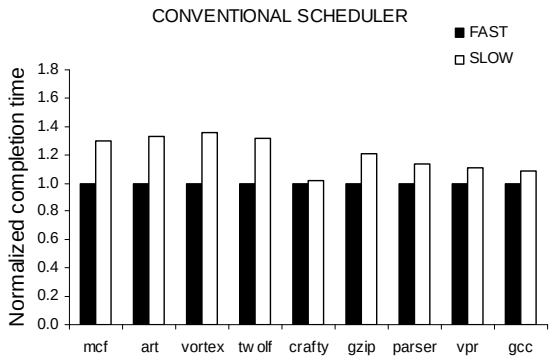

Figure 6. Co-runner-dependent performance variability with the conventional scheduler.

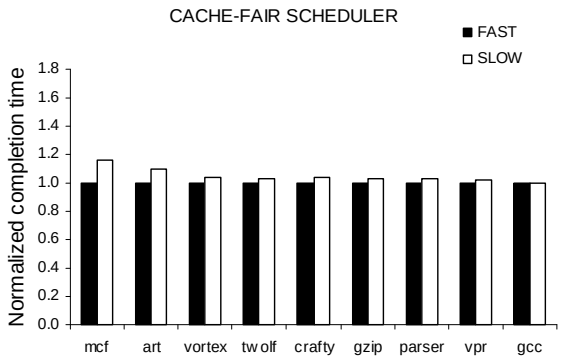

Figure 7. Co-runner-dependent performance variability with the cache-fair scheduler.

dependent performance variability is evident for most benchmarks, and reaches as much as 36% for *vortex*. *Vortex* runs 36% slower in the slow schedule, because its L2 cache miss rate is 85% higher than in the fast schedule.

Figure 7 shows normalized completion times with the cache-fair scheduler. The variability is significantly smaller. For *vortex*, the difference in completion times was 7%, reduced by more than a factor of five. For *gzip*, the difference was reduced from 21% to 3% – a factor of seven. One benchmark, *crafty*, that experienced only a small performance variability (2%) with the conventional scheduler, experienced a slightly higher variability with the cache-fair scheduler (4%); this was due to small errors in the model. The cache-fair scheduler reduced performance variability by at least a factor of two for the remaining benchmarks, which all experienced a significant variability (at least 8%) with the conventional scheduler.

5.3 Effect on performance

We now evaluate the effect of our scheduling algorithm on absolute performance. As expected, applications with high cache requirements may experience longer completion times with the cache fair scheduler than with the conventional scheduler, because the cache-fair scheduler would reduce their CPU quantum to compensate them for occupying more than their fair share of the cache. Conversely, applications with low cache requirements may experience shorter completion times with the cache-fair scheduler.

Figure 7 shows raw completion times for each benchmark in each schedule. There are four bars for each benchmark. The first two bars show the completion times for the fast and slow schedules with the cache-fair scheduler, the second two – with the conventional scheduler. Smaller bars indicate shorter completion times.

Applications with high cache requirements, especially *mcf* and *art*, have longer average completion times with the cache-fair scheduler. On the contrary, applications with low cache requirements, such as *gzip* and *vpr*, achieve shorter completion times with the cache-fair scheduler.

Such effect on absolute performance is expected. The goal of the cache-fair scheduler is to reduce the effects of unequal cache sharing on performance. Those applications that are hurt by unequal cache sharing experience improved performance, and vice versa. Nevertheless, all applications experience improved performance stability and predictability.

5.4 Effect on best-effort threads

In our experiments, best-effort threads occasionally experienced a small performance degradation, and in some cases – a performance boost. The largest observed degradation was an 8% slowdown relative to a best-effort thread's performance with the conventional scheduler. Best-effort threads

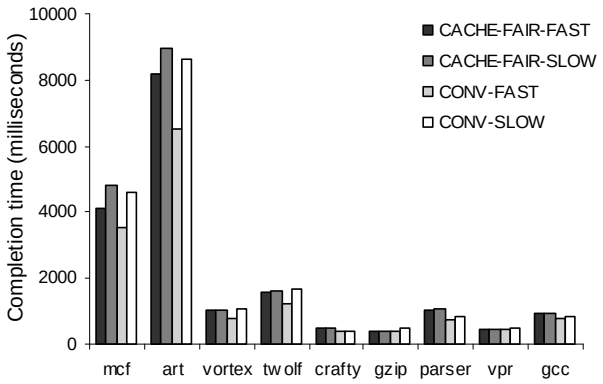

Figure 8. Raw completion times in the fast and slow schedule with cache-fair and conventional scheduler.

experienced performance boost when their CPU shares increased in response to decrease in cache-fair threads' CPU shares. Although we did not observe significant degradation in best-effort thread performance in our experiments, it is important to prevent it in general. One way to do this is to limit how much the scheduler can reduce a best-effort thread's CPU share.

5.5 Comparison with cache partitioning

Recent research proposed using dynamic cache partitioning on multicore processors to address unequal cache sharing [7,13,33-35]. Although it has been shown that dynamic cache partitioning can be used to lessen the effects of "bad" co-runners [7], we found that cache partitioning cannot reduce co-runner-dependent performance variability to the extent that cache-fair thread scheduling does.

We implemented dynamic cache partitioning in our simulated processor in a similar fashion as was done in the recent work [7], and ran the nine benchmarks (in the same fast and slow schedules as shown in Table 2) on a dual-core machine with the cache equally partitioned among the cores. Partitioning reduced co-runner-dependent performance variability only for three out of nine benchmarks and made no difference for the remaining six benchmarks. While performance variability due to cache sharing was eliminated, there remained performance variability due to contention for the memory bus. (We confirmed that memory-bus contention was the problem by showing that running with unlimited memory-bus bandwidth eliminated performance variability.) Our cache-fair algorithm incorporates memory-bus delay into the fair CPI model, accounting for performance variability due to sharing of the memory bus.

5.6 Scheduler overhead and scalability

The cache-fair scheduling algorithm was designed to avoid measurable performance overhead. Most of the work done during the reconnaissance phase involves accessing hardware performance counters, which involves only a few processor cycles per access. Our method for estimating the fair cache miss rate requires generating only about ten data points for each cache-fair thread, keeping memory overhead low. A small number of data points also limits the overhead of the linear regression, which runs in $O(N)$ steps, where N is the number of data points. The calibration phase involves simple arithmetic operations and inexpensive access to hardware performance counters.

Scalability was an important design goal because the number of cores on multicore processors is likely to increase, and it is important that our algorithm work for future processor generations. Although we have not yet empirically evaluated the scalability of our algorithm, we believe our design will scale because we avoid inter-processor communication and because the amount of work done in phases of the algorithm is independent of the number of cores.

It has been shown that inter-processor communication can limit scalability of multiprocessor operating systems [10]. Our algorithm makes scheduling decisions using thread-local information, so there is no communication among the cores. This puts our solution at an advantage over algorithms for multicore processors that rely on *co-scheduling*, i.e. making sure that a thread only runs with the "right" co-runner [14-16]. Co-scheduling is difficult to implement without inter-core communication, because scheduling a thread to run on one core requires knowing what threads are running on other cores.

The amount of work done in the reconnaissance and calibration phases of the algorithm is fixed per thread. Although in the reconnaissance phase we measure the miss rates of the thread's co-runners, the hardware allows measuring the aggregate miss rate of all of the co-runners, so the cost of this measurement does not increase with the number of cores.

5.7 Applicability to other hardware architectures

The implementation of our algorithm has two hardware-specific components: the model for estimating fair CPI and the mechanism for accessing hardware performance counters.

Modeling CPI as the function of cache miss rate is well studied, and accurate models for different kinds of processors (single- and multithreaded cores, single-issue and multiple-issue pipelines) exist [28-32,43]. Therefore, the fair CPI model can be easily adapted for most modern processors.

Our algorithm relies on runtime statistics that can be obtained from hardware performance counters. We used performance counters that are available on the UltraSPARC T1 processor [22]; other multicore processors have similar counters [23].

We demonstrated how our algorithm targets performance variability due to sharing of the L2 cache, because studies have shown that the L2 cache is a performance-critical component on multicore processors [8,9]. Our technique also applies to processors with shared L1 [2,4] or L3 caches [1,3].

6. RELATED WORK

We discuss alternative ways to address the effects of unequal cache sharing in hardware and software.

Multicore processor architectures that enforce fair resource sharing or expose control over resource allocation to the operating system have been proposed in the past [7,13,33-35]. Such hardware was used for improved performance predictability [33], fairness [7,34], and performance [13,35]. The advantage of a hardware solution is that it can be used to address fair resource sharing for all resources, not just CPU caches, so it could be used for processors that have shared resources other than caches. The downside of a hardware solution is higher cost and long time-to-market of such hardware. To the best of our knowledge, none of the proposed hardware architectures has been made commercially available. Our scheduling algorithm can be used on systems that exist today, allowing applications to enjoy the benefits of multicore processors without losing attractive properties offered by thread schedulers on conventional processors.

Software solutions based on co-scheduling, i.e. aiming to select the "right" co-runner for a thread, have been used to improve performance [14-15] and predictability [16]. Co-scheduling requires being able to determine how a thread's performance is affected by a particular co-runner. To do this, previous work used performance models and heuristics. Effectiveness of these models and heuristics has been demonstrated on systems running at most four concurrent threads. It is not clear whether these methods will scale for multicore systems with larger degrees of concurrency [2,4]. Another limitation of co-scheduling is that if the right co-runner for a thread cannot be found, the thread's performance remains vulnerable to co-runner-dependent performance variability.

7. SUMMARY

We presented the cache-fair scheduling algorithm, a new operating system scheduling algorithm for multicore processors. We evaluated our implementation of the algorithm in Solaris 10 and showed that it significantly reduces co-runner-dependent performance variability, while imposing little penalty on best-effort threads.

Co-runner-dependent performance is the result of unequal cache sharing, and by eliminating it, we address the problems caused by unequal cache sharing. Using our algorithm, applications are not penalized for with "bad" co-runners. This permits better priority enforcement and results in improved fairness. Our technique enables fair CPU accounting in Grid systems: applications can be charged only for the CPU cycles given to them by the system scheduler, and not for the "compensation" cycles given to them by the cache-fair scheduler. Finally, applications enjoy improved performance stability and predictability, which facilitates performance tuning and forecasting.

We demonstrated that our solution is viable by implementing it in a commercial operating system, relying only on features commonly available on commercial hardware and not requiring any advance knowledge about the workload.

Improved power efficiency and hardware utilization, and superior performance-per-watt ratio could make multicore processors the dominant CPU in the future. In order to enjoy these benefits while retaining good systems properties that we have cultivated for years, we must adapt our software for this new hardware.

8. REFERENCES

[1] Sun Microsystems UltraSPARC VI+ Features: http://www.sun.com/processors/UltraSPARC-IVplus/features.xml

[2] M. Funk. Simultaneous Multi-threading (SMT) on eServer iSeries Power5™ Processors, *http://www-03.ibm.com/servers/eserver/iseries/perfmgmt/pdf/SMT.pdf*.

[3] POWER4 System Microarchitecture, *http://www-03.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html*.

[4] P. Kongetira. A 32-way Multithreaded SPARC(R) Processor. http://www.hotchips.org/archives/hc16/, *HOTCHIPS 16*, 2004.

[5] AMD: Multi-core Processors – the Next Evolution in Computing, *http://multicore.amd.com/WhitePapers/Multi-Core_Processors_WhitePaper.pdf*

[6] Intel: White Paper: Superior Performance with Dual-Core, *ftp://download.intel.com/products/processor/xeon/srvrplatformbrief.pdf*.

[7] S. Kim, D. Chandra and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture, In *Intl. Conference on Parallell Architectures and Compilation Techniques (PACT)*, 2004.

[8] A. Fedorova, M. Seltzer, C. Small and D. Nussbaum. Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design, In *USENIX Annual Technical Conference*, 2005.

[9] S. Hily, A. Seznec. Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading. In *MTEAC'98*.

[10] B. Gamsa, O. Krieger, J. Appavoo, M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Symposium on Operating System Design and Implementation (OSDI)*, 1999.

[11] D. Nussbaum, A. Fedorova, C. Small. The Sam CMT Simulator Kit, *Sun Microsystems TR 2004-133*, 2004.

[12] SPEC CPU2000 Web site: http://www.spec.org
[13] G.E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning, In the *8th International Symposium on High-Performance Computer Architecture*, 2002.
[14] A. Snavely and D. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Machine, In the *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.
[15] S. Parekh, S. Eggers, H. Levy, J. Lo. Thread-sensitive Scheduling for SMT Processors, *www.cs.washington.edu/research/smt/*, 2000.
[16] R. Jain, C.J. Huges, S.V. Adve. Soft Real-Time Scheduling on Simultaneous Multithreaded Processors, In *23rd IEEE Real-Time Systems Symposium (RTSS)*, 2002.
[17] J. Li and J. F. Martinez. Power-performance implications of thread-level parallelism on chip multiprocessors. In *Int'l Symp. on Performance Analysis of Systems and Software*, 2005.
[18] L. Barroso et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing, In *27th International Symposium on Computer Architecture (ISCA)*, 2000.
[19] X. Shen, Y. Zhong and C. Ding. Locality Phase Prediction. In the *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
[20] E. Berg, E. Hagersten. StatCache: a probabilistic approach to efficient and accurate data locality analysis, in the *International Symposium on Performance Analysis of Systems and Software*, 2004.
[21] Benjamin Lee. An Architectural Assessment of SPEC CPU Benchmark Relevance, *Harvard University Technical Report TR-02-06*, 2006.
[22] UltraSPARC T1 supplement to UltraSPARC Architecture 2005 Specification (Hyperprivileged), *opensparc-t1.sunsource.net*
[23] IBM PowerPC 970FX RISC Microprocessor User's Manual, *IBM Website*
[24] Pentium 4 Programmer's Manual, *download.intel.com/design/Pentium4/manuals/25366919.pdf*
[25] E. Berger et al. Scheduler-Aware Virtual Memory Management, presented as posted in *Symposium on Operating Systems Principles*, 2003.
[26] C. Cascaval, L. DeRose, D.A. Padua, and D. Reed, Compile-Time Based Performance Prediction, in the *12th Intl. Workshop on Languages and Compilers for Parallel Computing*, 1999.
[27] A. Agarwal, J. Hennessey, M. Horowitz, An Analytical Cache Model, *ACM Transactions on Computer Systems*, vol.7(2), pp. 184-215, 1989.
[28] Daniel J. Sorin, et al., Analytic Evaluation of Shared-memory Systems with ILP Processors. In *International Symposium on Computer Architecture (ISCA)*, 1998.
[29] R. E. 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, Vol. 45 NO. 6*, November 2001.
[30] A. Fedorova, M. Seltzer and M. Smith. Modeling the Effects of Memory Hierarchy Performance on the IPC of Multithreaded Processors, *Technical Report TR-15-05, Division of Engineering and Applied Sciences, Harvard University*, 2005.
[31] R. Saavedra-Barrera, D. Culler and T. von Eicken, Analysis of Multithreaded Architectures for Parallel Computing, *SPAA 1990*.
[32] P. K. Dubey, A. Krishna and M. Squillante, Analytic Performance Modeling for a Spectrum of Multithreaded Processor Architectures, *MASCOTS 1995*.
[33] F.J. Cazorla et al. Predictable Performance in SMT Processors, In *Computing Frontiers*, pp. 433-443, 2004.
[34] S. E. Raasch and S. K. Reinhardt. Applications of Thread Prioritization in SMT Processors. In *Proc. of the Workshop on Multithreaded Execution And Compilation*, 1999.
[35] G. Dorai, D Yeung. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance, In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002.
[36] D. Chandra, F. Guo, S, Kim, Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture, In the *11th International Symposium on High Performance Computer Architecture (HPCA)*, 2005
[37] Sun Grid Frequently Asked Questions (#14) http://www.sun.com/service/sungrid/faq.xml#q14
[38] D. Patterson and K. Yelick, *Final Report 2002-2003 for MICRO Project #02-060*, University of California, Berkeley, CA 2003.
[39] D. Willick and D. Eager, An Analytical Model of Multistage Interconnection Networks, In *Proc. of 1990 ACM SIGMETRICS*, pp. 192-199.
[40] W. Feller, An Introduction to Probability Theory and Its Applications, vol. I, (VI(9), *John Wiley and Sons*, 1968
[41] R. Cole et al., On Balls and Bins with Deletions, *Proceedings of Random '98*, pp. 145-158.