

CASC: A Cache-Aware Scheduling Algorithm For Multithreaded Chip Multiprocessors

Alexandra Fedorova^{†‡}, Margo Seltzer[†], Michael D. Smith[†] and Christopher Small[‡]
[†]*Harvard University*, [‡]*Sun Microsystems*

ABSTRACT

In this paper we propose CASC, a cache-aware operating system scheduling algorithm for multithreaded chip multiprocessors (CMT). CMT is emerging as a popular architecture for server platforms, and most major hardware manufacturers plan or already have released CMT processors. It is the job of the operating system to manage the shared resources of the hardware, and the second-level (L2) cache is a critical shared resource in CMT processors. We propose an operating system scheduling algorithm, CASC, that improves management of the L2 cache. CASC works by co-scheduling threads that collectively achieve a low L2 miss rate and by giving priority to threads that do not require much space in the L2 cache. In this paper we describe the design of CASC and its implementation in Solaris™ 10. We show that CASC reduces L2 miss rates by 15-46% and achieves improvement in processor throughput of 28-50%.

1. INTRODUCTION

This paper describes and evaluates a practical implementation of CASC, a Cache-Aware operating system Scheduling algorithm for multithreaded chip multiprocessors (CMT). CMT processors are emerging as a popular platform for future server systems [7,19-23,29]. These processors share different resources and have different bottlenecks than traditional processors. In particular the second-level (L2) cache has been found to be a critical shared resource in CMT [1-3,7]. It is the job of the operating system to manage the shared resources of the underlying hardware, because the OS can often determine how these resources are used by the applications. Because of the novelty of this architecture, operating systems are not yet tuned for CMT processors.

We present CASC, an operating system scheduling algorithm that improves management of the L2 cache. CASC reduces contention for the L2 cache by co-

scheduling threads that require a lot of L2 cache with threads that require little L2 cache and by giving priority to threads that achieve low L2 cache miss rates. CASC uses a novel online algorithm that efficiently estimates L2 cache miss rates achieved by groups of threads sharing the cache. CASC reduces L2 miss rates by 15-46% and improves instruction throughput by 28-50%.

This work builds on a previous study [2], where we proposed a scheduling algorithm with similar goals. In that work we evaluated the performance potential of the earlier algorithm via a combination of analytical study and simulation. The prior algorithm required estimating L2 cache miss rates achieved by groups of threads, and we showed that, if it were possible to estimate them at runtime with no performance penalty, and to determine, without substantial runtime overhead, which subsets of threads produced the best cache performance, this algorithm would reduce L2 cache miss rates by 19-37%.

In fact, obtaining this information at runtime proved to have high overhead. In order to construct a working cache-aware scheduling algorithm, we had to address the following challenges:

- How can we estimate miss rates for groups of threads at runtime with little performance penalty? (The only online method of which we are aware [38] introduces about 20-40% of runtime overhead.)
- How can we make scheduling decisions based on estimated miss rates without performing exhaustive search through all possible subsets of threads and without requiring synchronization among processors.

CASC solves these problems. We have implemented CASC in a new scheduler for Solaris 10. Our implementation of CASC efficiently estimates miss rates for groups of threads, and obtains the performance benefits comparable to what we predicted. The contributions of this work are:

- The design and implementation of a low-overhead online algorithm for estimating miss rates for groups of threads.
- The design, implementation, and evaluation of a new thread-scheduling algorithm that makes scheduling decisions based on estimates of L2 miss rates. Our method does not require global synchronization among processors, which is an important requirement for multiprocessor scheduling algorithms, because cross-processor synchronization can limit scalability of such systems [25].

CASC is a software algorithm that achieves significant reduction in L2 miss rates, equivalent to doubling the size of the L2 cache. In other words, for the workloads that we ran, CASC created the illusion of having an L2 cache twice as large as what was actually available.

The rest of the paper is organized as follows. In Section 2 we provide background on CMT processors and motivate our research. In Section 3 we give an overview of our scheduling algorithm. In Section 4 we describe the design and implementation of CASC. In Section 5 we present experimental results, discuss limitations of CASC, talk about the impact of CASC’s scheduling policy on fairness, and project what implications our results will have

on the scalability of CMT processors. In Section 6 we discuss related work, and in Section 7 we conclude.

2. CMT PROCESSORS AND L2 CACHE

CASC is targeted at multithreaded chip multiprocessors (CMTs). CMT processors combine chip multiprocessing (CMP) [32] and hardware multithreading (MT) [26]: they are built with several multithreaded processor cores on a single chip.

Multithreaded and multi-core processor designs are becoming increasingly popular: Most new processors released today by AMD, Fujitsu, IBM, Intel and Sun Microsystems are multithread and/or multi-core [19-23]. IBM released its Power 5 CMT processor in July 2004 [21]. AMD announced plans to launch its dual-core Opteron in mid-2005 [22]. Sun Microsystems’ Niagara® CMT processor is now available in the labs and should be released to the general public in 2006 [19].

The popularity of these processors is due to the inability of conventional state-of-the-art processors to meet the needs of memory-intensive modern workloads. Web services, application servers, and on-line transaction processing systems are notorious for high processor cache miss rates, which cause the processor to block on memory [3,14-18]. These workloads have processor pipeline utilizations of less than 20% on conventional processors [18]. This means that 80% of the time the processor is blocked while application cache miss rates are being serviced, and its processing resources are wasted. State-of-the-art architectural methods for reducing instruction-stream memory latency, such as prefetching and out-of-order execution, are not effective for these types of workloads, because the average memory access times these workloads experience are large: these techniques are not able to overcome the growing gap between processor and memory performance [9].

Multithreaded architectures use an alternative technique for hiding long memory latencies. By running several threads in parallel they mask latency of individual threads, just like software multithreading masks latency of disk I/O (See Figure 1). As a result, if one thread blocks on a memory access, other threads can make forward progress. This latency-hiding property is fundamental to hardware multithreading. Studies have shown that even single-core multithreaded processors improve performance by 9-65% [26,33]. By combining several multithreaded processor cores on a single chip in CMT processors, performance improvements can be even larger.

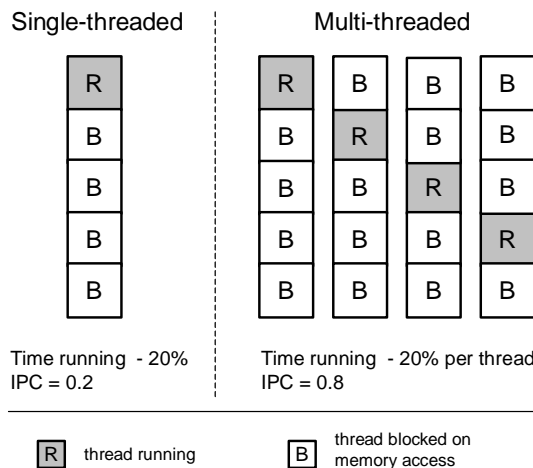


Figure 1. Each box denotes the state of the processor pipeline for a single cycle. For a single-threaded processor, if a thread spends 20% of its time running and the remainder of its time blocked handling cache misses, the processor is blocked 80% of the time and completes only one instruction in five cycles, yielding 0.2 instructions per cycle (IPC). A multithreaded processor hides memory latency. Although each thread spends 80% of the time in the blocked state, overall, the processor is blocked only 20% of time, yielding 0.8 IPC.

The speed-up from MT comes at relatively low cost in terms of transistors. To enable hyper-threading on the Pentium 4, the number of transistors had to be increased by only 5% [39]. Now that squeezing performance improvement out of extra transistors is becoming increasingly more difficult [20,29], the industry is betting on MT and CMT to continue improving performance of computer systems in the future [7,19-23,29].

The industry’s unanimous attention to these

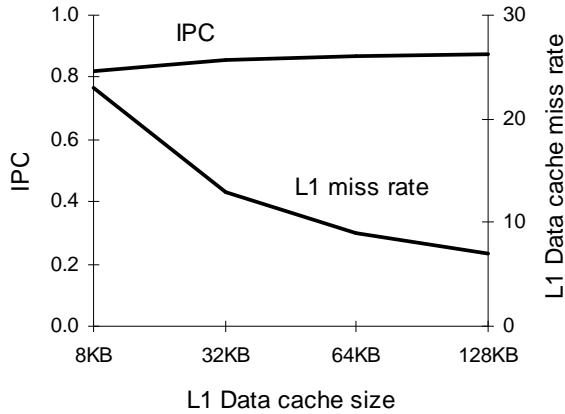


Figure 2. L1 data cache miss rate in misses per 100 ref. (right-hand Y-axis) and processor instructions per cycle (IPC) (left-hand Y-axis) for a multithreaded SPEC CPU2000 workload for various cache sizes on a single core of a CMT processor. Note the relationship between the shapes of the curves: while the miss rate varies dramatically with the cache size, the IPC stays about the same. This shows that virtually all latency resulting from L1 cache misses is hidden by hardware multithreading.

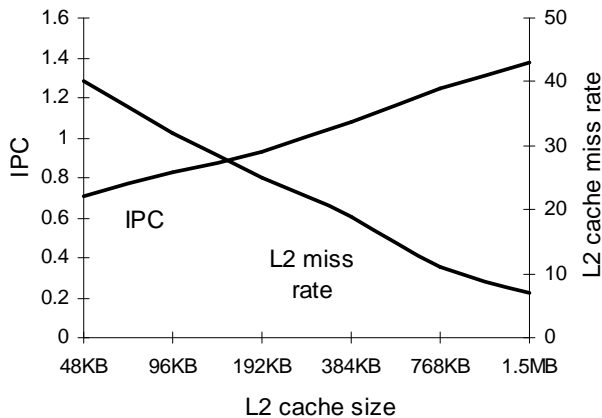


Figure 3. L2 cache miss rate in misses per 100 ref. (right-hand Y-axis) and processor instructions per cycle (IPC) (left-hand Y-axis) for a multithreaded SPEC CPU2000 workload for various L2 cache sizes on a dual-core CMT processor. Processor IPC degrades as the L2 miss ratio increases, because the L2-miss latency (i.e. the DRAM latency) cannot be masked entirely by CMT processors.

multithreaded architectures indicates that a large number of applications will be run on CMT processors in the near future. The operating system has a unique opportunity to increase the benefit that applications will get from CMT processors because it manages resource allocation of the thread contexts. While the subject of tuning OS resource management policies for conventional processors has been extensively addressed in the past, and the results of these studies have been incorporated in modern operating systems (e.g., scheduling policies for improved cache affinity, selecting the right duration of scheduling time slice, etc.), the same has not yet been done for CMT processors. This subject needs to be addressed if we want applications to get the most out of CMT processors.

The scheduling algorithm presented here improves performance of CMT processors by improving management of the L2 cache. L2 cache is typically shared among all thread contexts on CMT processors. We now explain why the L2 cache is a critical shared resource. Recall that CMT processors hide the memory latency of individual instruction streams. The amount of memory latency that can be hidden depends on the magnitude of the memory latency experienced by the application. We have shown (and other studies have shown this independently) that while CMT (and simple multithreaded) processors typically do a good job of hiding first-level (L1) cache miss latency, they are more limited in hiding L2-cache miss latencies [1,2]. Figure 2 shows that on a CMT processor, throughput in instructions per cycle (IPC) is fairly independent of the L1 miss rate, even when the size of the L1 cache is tiny. Figure 3, on the contrary, shows that processor throughput is vulnerable to high miss rates in the L2 cache.

When a workload misses in the L2 cache, the processor has to fetch data from the off-chip memory (DRAM). Accessing DRAM may take ten times as long as accessing an on-chip cache; and the gap between processor and memory performance is continuously widening [9]. Additionally, off-chip bandwidth may become a bottleneck [1,7]. Therefore, if the L2 cache miss rate is high, combined memory latencies may become too large to be hidden, and the processor will achieve poor utilization of its computing resources and correspondingly poor performance.

While good performance in the L2 cache is important for all multithreaded architectures, cache contention is more likely to be an issue on CMT processors, where the number of threads sharing the cache is larger (e.g., Sun Microsystems’ Niagara processor is equipped with 32 thread contexts). This is the reason why many studies,

including ours, have concluded that good performance in the L2 cache is critical in CMT processors [1-3,7], and this is why we were motivated to create a scheduling algorithm that improves management of this resource. In the rest of the paper we describe the design and implementation of such a scheduling algorithm, CASC.

3. OVERVIEW OF THE ALGORITHM

Before diving into the implementation in Section 4, we present a high-level overview of CASC and highlight the challenges in designing a feasible implementation.

Intuitively, our algorithm is quite simple: observe which threads *collectively* achieve a low L2 cache miss rate when run in parallel, and then schedule threads in these “cache-friendly” subsets.

This algorithm is inspired by balance-set scheduling, which was introduced to improve performance of virtual memory [36]. A *balance set* is a subset of processes whose combined working set fits in memory. Scheduling processes in such balance sets assures that the system achieves good performance in the VM cache. While this approach may be well suited for virtual memory systems, we found that for CPU caches, working set size is not a good indicator of a workload’s cache behavior [37]; instead, we use the L2 miss rate as an indicator of threads’ performance in the L2 cache.

The key challenge for this algorithm then is to estimate L2 miss rates for groups of threads at runtime with little to no performance penalty.

In our previous evaluation study [2] we used a Berg-Hagersten cache model [38] that accurately predicted miss rates for a single thread and adapted this model to work for multiple threads [37]. Although this model had good accuracy and, according to our estimates, a scheduler using it would be able to effect significant performance improvement, we found that implementing this method in a real system produced a significant runtime penalty. In order to predict miss rates with the Berg-Hagersten model, it was necessary to monitor threads’ memory access patterns at runtime. Such monitoring is expensive: the user-level tool designed by Berg and Hagersten incurs 20-40% runtime overhead. The tool monitors threads’ memory access patterns by read-protecting the pages that contain the locations being monitored. It then handles traps that occur when the memory locations of interest are accessed. If the monitoring is implemented in the kernel, handling these traps does not require multiple context switches between the kernel and user-level, and this led to our initial

optimism that a kernel implementation would have sufficiently low overhead to be practical. However, we found that this was not the case: the overhead was still significant. So we developed an alternate approach in CASC.

The approach we use in CASC is based on the assumption that the overall L2 miss rate incurred by a set of threads running concurrently can be expressed as a linear combination of the L2 miss rates that the threads in the set would experience if they were each running alone, with a smaller, dedicated L2 cache. (The validity of this assumption is challenged by interactions between threads’ memory access patterns, as we discuss later.) The algorithm would thus observe the L2 miss rates achieved by sets of running threads and use these data to estimate each thread’s L2 miss rate with a smaller cache. It would then use these estimated L2 miss rates to predict L2 miss rates achieved collectively by sets of threads.

4. CASC DESIGN AND IMPLEMENTATION

We now describe the design and implementation of CASC in detail. We implemented CASC in Solaris 10, so we begin by explaining how it fits into the Solaris scheduling infrastructure (Section 4.1). We then describe how we estimate threads’ miss rates (Section 4.2). Finally, we explain how CASC makes scheduling decisions (Section 4.3).

4.1 CASC scheduling class

The CASC scheduler is implemented as a loadable scheduling class module in Solaris 10. The Solaris scheduler consists of two parts: the *dispatcher* and a *scheduling class* [12]. The dispatcher implements the mechanism: it distributes threads among scheduling queues (there is one scheduling queue per processor), and dispatches threads on processors. The scheduling class implements a policy: it decides how long each thread gets to stay on a processor and the priority assigned to the thread. Solaris currently includes four different scheduling classes: time-sharing (implements a multi-level feedback queue scheduling policy), fair-share, fixed-priority and real-time. By default all user programs use the time-sharing class. To begin using an alternative scheduling class, the program has to make a `pricnt1()` system call and specify the name of the new scheduling class (“CA” for CASC).

Each scheduling class implements the scheduler interface. The rest of the kernel interacts with the

scheduling class through this interface. Here are some example functions included in this interface:

```

cl_enterclass called when a thread enters the scheduling
                class;
cl_tick       called on every clock tick;
cl_sleep     called when a thread goes to sleep;
cl_preempt   called when a thread is preempted;
cl_wakeup    called when a thread wakes up;
cl_exit      called when a thread exits.

```

CASC re-uses most of the time-sharing class functions; in some functions it does extra processing of its own. This means that CASC adheres to the time-sharing class' scheduling policy as much as possible.

As will become clear in the next section, we needed the kernel to call back to our scheduler when a thread was going off a processor (OFFPROC) and on a processor (ONPROC). For this, we added two functions, `cl_onproc` and `cl_offproc` to the scheduling class interface and inserted calls to these functions when the processor switches among threads.

4.2 Estimating miss rates

We express L2 miss rate as the *number of L2 misses per 100 references*. We assume that the overall L2 miss rate ($L2MR_{group}$) incurred by a set of n threads running concurrently with a cache of size C can be expressed as a linear combination of the threads' *partial-cache miss rates* (PCMR), the L2 miss rates that the individual threads in the set would incur if they were each running with a dedicated L2 cache of size C/n . Specifically, we assume that $L2MR_{group}$ is simply the average of individual threads' PCMRs:

$$L2MR_{group}(C) = \frac{1}{n} \sum_{i=1}^n PCMR_i(C/n) \quad (1)$$

It should be apparent that this method cannot be highly accurate, because it assumes that there is no interference among threads in the cache. In our previous study we employed this method to estimate miss rates for groups of threads; to estimate individual threads' PCMRs we used the Berg-Hagersten model for single-threaded workloads [38]. The resulting estimated group L2 miss rates were on average within 17% of actual values. Although the accuracy was not high, our prior work demonstrated that it would be

sufficient for the algorithm to achieve significant performance improvement [2].

In CASC, we cannot use the Berg-Hagersten model to estimate individual PCMRs, because this would induce too high an overhead. Therefore, in CASC we use the reverse of the above method: we observe the miss rates achieved by groups of threads, and from that we estimate the PCMRs of threads in those groups. Once we know the PCMR of each thread, we can estimate the miss rate for an arbitrary group of threads. We will now explain this in more detail.

Let's say that we have a CMT processor with n total thread contexts (i.e. virtual processors) running a workload of $n + 1$ threads. There are $n + 1$ different ways to assign these threads to n virtual processors if the order of threads on processors does not matter. These thread-to-processor assignments can be represented in an $(n + 1) \times (n + 1)$ matrix, where each row i represents a particular assignment a_i , and each column j represents a thread t_j . Each entry ij in the matrix denotes whether thread t_j is included in assignment a_i (the entry is equal to "1" if the thread is included). Consider the following sample matrix for $n = 4$.

	t_0	t_1	t_2	t_3	t_4
row0	0	1	1	1	1
row1	1	0	1	1	1
row2	1	1	0	1	1
row3	1	1	1	0	1
row4	1	1	1	1	0

Row 0 denotes that threads t_1 through t_4 are running simultaneously on a system's four virtual CPUs; row 1 denotes running threads t_0 and t_2 through t_4 .

Now, if we measure the L2 miss rate for each of these thread combinations, (i.e. rows in the matrix), we can write down the following system of linear equations:

$$\begin{aligned}
 (x_1 + x_2 + x_3 + x_4)/n &= L2MR(0) \\
 (x_0 + x_2 + x_3 + x_4)/n &= L2MR(1) \\
 (x_0 + x_1 + x_3 + x_4)/n &= L2MR(2) \\
 (x_0 + x_1 + x_2 + x_4)/n &= L2MR(3) \\
 (x_0 + x_1 + x_2 + x_3)/n &= L2MR(4)
 \end{aligned}$$

where the x_i are threads' PCMRs, and $L2MR(i)$ is the measured L2 miss rate for row i . Assuming that the L2 miss rate of a group of threads can be approximated by the average of PCMRs of the threads in that group (recall

Formula 1), solving this system of equations¹ gives us the threads' estimated PCMRs.

We stress that the PCMRs obtained in this manner are not necessarily accurate estimates of thread miss rates when running with a smaller dedicated cache, and this method should not be used in a general-purpose cache model. Rather, we treat an estimated PCMR as a *locality score*: a thread with poor locality of reference will receive a high PCMR, a thread with good locality of reference will receive a low PCMR. We found that with this approach CASC is able to categorize threads according to their degrees of locality correctly in most cases. We discuss this in more detail in Section 5.3.2.

Note that solving the linear system is only necessary on start-up, when the PCMRs of zero or very few threads are known. If there are at least $(n - 1)$ threads in the system whose PCMR is known, we can use the following simplification to estimate PCMRs of newly arrived threads. We co-schedule each thread whose PCMR is unknown with $(n - 1)$ threads whose PCMRs are known. We measure the L2 miss rate produced by this schedule, and solve a single linear equation to compute the PCMR of the new thread:

$$(PCMR_i + PCMR_j + PCMR_k + x)/n = L2MR, \quad (2)$$

where $PCMR_i$, $PCMR_j$ and $PCMR_k$ are previously estimated PCMRs of some threads i , j and k , x is the PCMR of the new thread, and $L2MR$ is the L2 miss rate observed during this run.

We call the process of estimating PCMRs using the matrix the *Initial Measurement*. We call the process of estimating a single PCMR the *Secondary Measurement*.

After we estimate the PCMRs of all threads running in the system we can estimate the L2 miss rate for any arbitrary group of threads by averaging the PCMRs of the threads in the group. With this approach we are able to estimate the L2 miss rate for a group of threads to within 30% of the actual value in the worst case. Although the accuracy of this method is not perfect and would be too low for a general-purpose cache model, as we show in Section 5 this method helps us achieve significant performance improvement.

¹ Solving a system of linear equations requires floating-point operations, which are not permitted in the Solaris kernel. We modified our linear equation solver to use only integer operations, which introduces negligible error.

4.2.1 Re-measuring PCMRs

Since thread L2 miss rates vary over time, PCMR estimation should be done periodically. Program execution goes through phases. A phase is a period in a program execution that has stable or slow-changing characteristics inside the phase but disruptive transition periods between phases [10,11]. Since program's behavior with respect to the L2 cache may change from phase to phase, CASC should refresh the PCMR estimates for every phase of the program.

An online phase-detection algorithm proposed by Balasubramonian *et al.* detects phase changes even for workloads with inconsistent phase lengths [11]. This algorithm collects program statistics every time interval t , and if any of the statistics at interval $(t+1)$ differ significantly from interval t , it signals a change of phase.

CASC could use a similar approach: it could monitor effective group miss rates and if the observed values significantly deviate from the predicted, it could re-measure threads' PCMRs. Implementing re-measurement is the subject of future work; here we limit ourselves to workloads with stable behavior.

4.2.2 Implementation of measurement phases

4.2.2.1 Initial measurement

The most challenging part of implementing the initial measurement is orchestrating a simultaneous run of each matrix row².

To accomplish this, we increase priorities of all threads participating in the initial measurement. When all threads of some matrix row are ONPROC, we record the time when the row went ONPROC and reset the hardware counters. When any thread of that row goes OFFPROC, the row is no longer running all of its member threads simultaneously, so we terminate the measurement interval. We record the duration of the interval and the L2 miss rate read from the hardware counters. We continue this process until each row accumulates a total duration of measurement intervals greater than or equal to some value `MIN_RUNLENGTH` (we discuss how we set `MIN_RUNLENGTH` in Section 4.2.2.3). For each row, we weigh the L2 miss rate measured at each interval by the

² We assume that the number of threads in the system is at least $n+1$, where n is the number of processors. If this is not the case, the initial measurement could be adjusted to work with fewer threads. If, during the initial measurement, there are more than $n+1$ threads, only the first $n+1$ threads that entered the system participate in the initial measurement.

fraction that that interval constitutes of the entire measurement, and compute the weighted average.

On a simulated CMT processor with eight thread contexts and 18 threads, we observed an average duration for the initial measurement of 9.8 seconds, and a median value of 11.5 seconds (MIN_RUNLENGTH was set to 2 milliseconds). We stress that although during the initial measurement the system is not running the best thread schedules, there is no penalty to the applications: we observe no measurable performance degradation during the initial measurement, because the threads participating in the measurement are running as they would with the default scheduler (except that their priorities are elevated); CASC simply observes the L2 miss rates achieved by various subsets. Extra processing is performed only when solving the system of linear equations. And since this is done only once at the end of the initial measurement, the overhead is not significant. (We provide more discussion on CASC overhead and present concrete data in Section 5.3.4.) As our target application domain is long-running server applications which may persist for days or weeks, it is acceptable for the system to take 10-15 seconds to reach stable state; it would also be acceptable for other applications, whose lifetimes can be measured in minutes (e.g., desktop user applications).

4.2.2.2 Secondary measurement

Secondary measurement proceeds as follows. When a new thread enters the system we mark its PCMR as unknown. Whenever this thread goes ONPROC along with peer threads whose PCMRs are known, we reset the hardware counters and start the measurement interval. When this thread or one of the peers goes OFFPROC, the group is no longer running in parallel, so we terminate the measurement interval, read the hardware counters and compute the thread's PCMR using Formula 2. We proceed in this manner until the combined duration of all measurement intervals reaches MIN_RUNLENGTH. (Note that at each interval the thread may be running with a different set of peers.) We weigh the PCMRs estimated at each measurement interval by the fraction that that interval constitutes of the entire measurement, and compute the weighted average.

To ensure that threads with unknown PCMRs always run with peers whose PCMRs have already been estimated, we force new threads to run on a specific CPU until we can estimate their PCMR. In our implementation of CASC we designate CPU 0 for this purpose. This accelerates

secondary measurement by ensuring that threads with unknown PCMRs never run simultaneously, and avoids missing a potential measurement opportunity.

The secondary measurement is short: in our experiments it took about 0.5 seconds per thread.

4.2.2.3 Selecting MIN_RUNLENGTH

MIN_RUNLENGTH is the minimum duration of the measurement phase, comprised of one or more shorter intervals. MIN_RUNLENGTH should be selected such that the L2 miss rate measured during the run is representative of long-term behavior of the threads. Programs usually go through various execution phases, and it is difficult to put an exact number on how long one has to run to capture the essential properties of the workload [10,11].

Balasubramonian et al. [11] found that the length of the interval representative of program behavior is highly workload-dependent and ranges from 10K to 320K instructions for most programs. However, there are outliers, such as 197.parser from the SPEC CPU2000 benchmark suite whose optimal measurement interval was found to be 40M instructions. The authors of this study describe an online algorithm that dynamically determines the optimal interval for the workload and adds only 1% performance overhead [11]. Incorporating this algorithm into CASC is outside the scope of the current work. In our current implementation we set MIN_RUNLENGTH to 2 milliseconds. This is longer than the maximum interval found in the work by Balasubramonian, and we expect that this would be long enough to capture the essential properties of most workloads.

4.3 Making scheduling decisions

Recall that our objective is to schedule threads in subsets that produce low L2 miss rates. A straightforward way to determine such subsets is to estimate cache miss rates for all possible assignments of threads to processors (by averaging the PCMRs of the subset members) and then to perform exhaustive search to find the ones that are projected to perform best. On a system with hundreds of threads and a dozen of virtual processors (i.e. thread contexts) there are over 10^9 ways to assign threads to processors, and estimating miss rates for all possible assignments is impractical. Furthermore, ensuring that only certain threads run in parallel, as a group, is difficult on a multiprocessor system where each processor dispatches threads independently of other processors and on- and off-processor transitions are not synchronized among

processors. Requiring such global synchronization could limit scalability of such systems [25].

In CASC we use the following approach. We associate with each virtual CPU a *bucket*, representing the maximum PCMR (MAX_PCMR) that will be assigned to the processor. Then we assign threads to buckets, according to their PCMRs. A thread is assigned to the bucket with the minimum MAX_PCMR that is greater than the thread’s PCMR. To make this algorithm more concrete, refer to Figure 4. Thread t0 with PCMR 1 will be assigned to bucket 0, thread t2 with PCMR 3 will get assigned to bucket 1, and so on. A thread is allowed to run only on CPUs whose MAX_PCMR is above that thread’s PCMR. So thread t2 with PCMR 3 will be allowed to run on CPUs 1 through 3. Thread t5 with PCMR 12 will be allowed to run only on CPU 3.

MAX_PCMR assignments of the buckets determine the maximum miss rate that the system will achieve. Recall that we estimate the L2 miss rate that a system achieves by averaging the PCMRs of threads that run in parallel. Refer again to Figure 4. CPU 0 will never run a thread whose PCMR is above 1, CPU 1 will never run a thread whose PCMR is above 5, and so on. Therefore, assuming that our linearity assumption holds, the system as a whole will never achieve a miss rate that exceeds the average of MAX_PCMRs of all the CPU buckets – 8.5 in the example from Figure 4.

Recall that our goal is to schedule threads in subsets that achieve a low L2 miss rate. Instead of coordinating threads to run in certain subsets we achieve this goal by restricting each thread to run on a subset of CPUs whose MAX_PCMRs are higher than the thread’s PCMR. As a result, cache-greedy threads will be allowed to run on only a handful of CPUs, making it impossible that they all run at the same time. Referring again to Figure 4, threads t4, t5 and t6, for example, will only be able to run on CPU 3. This

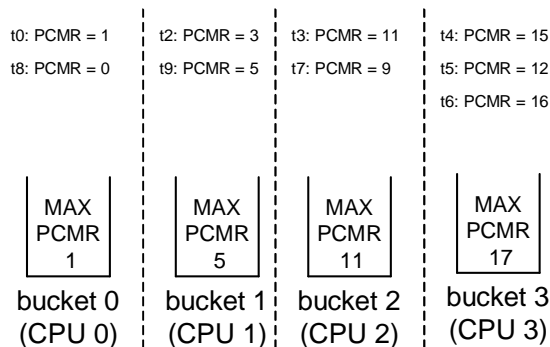


Figure 4. An example assignment of MAX_PCMRs.

assures that only a single cache-greedy thread runs at any time. As a result, the overall miss rate is reduced.

Restricting the subset of CPUs on which the thread is allowed to run (we refer to this as *thread CPU preferences*) will inevitably affect fairness: cache-greedy threads with more restrictive CPU preferences will get scheduled less often than cache-frugal threads with less limited preferences. We discuss the effect of CASC scheduling policy on fairness in Section 5.3.3.

We next explain how we assign MAX_PCMRs to CPU buckets. We refer to the process of assigning MAX_PCMRs to CPU buckets as the *bucketization algorithm*. Achieving a good assignment is tricky. A good assignment ensures that the system never goes above a specified miss rate threshold (we describe how we set the threshold in section 4.3.3), but should never restrict threads’ CPU preferences to the point that some processors are left idle. We now describe how we achieve these goals.

4.3.1 Bucketization algorithm

The pseudo-code for our algorithm is presented below. We begin by assigning to each CPU bucket the maximum possible MAX_PCMR (this would result in no restrictions on any thread’s CPU preferences). Then we compute the projected L2 miss rate by averaging the MAX_PCMRs across all CPU buckets. If the projected miss rate is higher than the threshold, we decrease the MAX_PCMR for one of the CPUs. A side-effect of decreasing the MAX_PCMR is that the threads whose PCMRs are greater than the new MAX_PCMR of the CPU will not be allowed to run on that CPU. To minimize the number of threads that are restricted from running on that CPU, we attempt to pick the CPU such that decreasing its MAX_PCMR will result in restricting the smallest number of threads. We call such a CPU a *minimum penalty CPU*. We proceed in this fashion, decreasing the MAX_PCMR on the minimum penalty CPU until the miss rate threshold is satisfied. As on each iteration we decrement the MAX_PCMR of some CPU, this algorithm will terminate in at most (INITIAL_MAX_PCMR x NUM_CPUS) steps.

Main loop of the algorithm:

```

projected_MR = compute_projected_miss_rate()

while(projected_MR > miss_rate_threshold)
    CPU = get_min_penalty_CPU()
    CPU->MAX_PCMR = CPU->MAX_PCMR - 1
    /* Restrict the threads whose PCMRs are
    greater than CPU->MAX_PCMR from running on
    CPU */

```

```
projected_MR = compute_projected_miss_rate()
```

Helper routines:

```
compute_projected_miss_rate():  
    total_PCMR = 0;  
    for(i = 0; i < NUM_CPUS; i++)  
        total_PCMR += CPU[i].MAX_PCMR;  
    return total_PCMR / NUM_CPUS;
```

```
get_min_penalty_CPU():
```

Find the CPU, such that decreasing its MAX_PCMR will result in restricting the smallest number of threads from running on that CPU. For example, if CPU 1 has a MAX_PCMR of 5, and the number of threads in that CPU bucket is 2, then the penalty for decreasing the MAX_PCMR of CPU 1 is 2. Do not select any CPU such that decreasing its MAX_PCMR will restrict some threads from running on any CPU at all.

There are situations when the miss rate threshold cannot be satisfied. For example, referring back to Figure 4, if the miss rate threshold is 3, the only way to satisfy it is to decrease the MAX_PCMR for bucket 3. But this may result in a situation where threads t4, t5 and t6 will not be able to run on any CPU at all. If this is the case, the bucketization algorithm announces that it cannot satisfy the desired miss rate threshold and terminates with the best MAX_PCMR assignment it can achieve.

An unfortunate MAX_PCMR assignment could also result in the condition that some CPUs are left idle. For example, referring back to Figure 4, if the bucketization algorithm achieves the assignment of {bucket 0: MAX_PCMR = 1; bucket 1: MAX_PCMR = 1; bucket 2: MAX_PCMR = 1; bucket 3: MAX_PCMR = 17} while striving to satisfy the miss rate threshold of 5, then all threads, except t0 and t8 will be restricted to run on CPU 3. CPUs 0, 1 and 2 will be left to run threads t0 and t8: two threads for three CPUs. As a result, at any given time one CPU will be idle. We ensure that this never happens in the following way: when the bucketization algorithm terminates, we check that the current MAX_PCMR assignment guarantees that each CPU can run at least m threads at all times. Since the workload we use for our experiments is CPU-bound, we set $m = 1$. If the workload is not CPU-bound, m should be set to a larger value. If each CPU cannot run at least m threads (provided there are enough threads in the system), we increase the miss rate threshold and repeat the bucketization algorithm. As we can always schedule all threads with a miss rate threshold of 100 (100%), the algorithm will terminate within a bounded

number of iterations. We imagine that an implementation would increase the miss rate threshold geometrically, rather than arithmetically.

The bucketization algorithm runs in a separate thread. It is performed at the end of the initial measurement. The algorithm is re-run after the secondary measurement if the PCMR of the new thread is larger than the largest PCMR in the system. The algorithm is always re-run after the secondary measurement if the existing MAX_PCMR assignment does not satisfy the miss rate threshold.

4.3.2 Enforcing thread CPU preferences

Once the bucketization algorithm completes, we set threads' CPU preferences. A thread is allowed to run only on CPUs whose MAX_PCMR is larger than the thread's PCMR.

We modified the Solaris dispatcher to respect threads' CPU preferences. We added a data structure, linked to the thread structure, which describes a thread's CPU preferences. This is an array of boolean values, one for each CPU. The value is set to "true" if the thread is allowed to run on the CPU corresponding to that array element. When dispatching threads to CPUs, the dispatcher checks whether the thread is allowed to run on the CPU it chose. If not, it chooses another CPU.

4.3.3 Selecting the initial miss rate threshold

Since the bucketization algorithm will terminate if the miss rate threshold cannot be reasonably satisfied (i.e. without starving some threads or leaving some CPUs unused), it is safe to set the miss rate threshold to some arbitrary small value. If this value is too small, the system will adapt and raise this value. For the experiments presented in this paper, we set the miss rate threshold to 15.

5. EXPERIMENTAL RESULTS

5.1. Simulation environment

For the experiments in this study we use a CMT system simulator [28], built as a set of extensions to the Simics simulation toolkit [30]. Simics provides full-system simulation of several popular hardware platforms. Our simulated architecture is built on top of the SPARC® platform and closely matches Sun Microsystems' Niagara CMT processor [19]. We validated our simulator against a real Niagara machine. The simulated machine can boot the Solaris operating system and run all programs that run on Solaris/SPARC platforms. All the simulations described in

this paper are execution-driven and include both user-level and OS code.

Our simulated CPU core has a single-issue pipeline supporting four threads, as does Niagara. Our simulator accurately simulates pipeline contention, the L1 cache, bandwidth limits on crossbar connections between the L1 and L2 cache, the L2 cache, and bandwidth limits on the path between the L2 cache and memory.

Our simulator is configured with two processor cores. Each core runs at 992 MHz and includes four hardware contexts, an 8KB level 1 (L1) data cache and a 16KB L1 instruction cache (both 4-way set-associative). Cache sizes and architectures were chosen to match those of Niagara. We simulate a unified 12-way set-associative L2 cache, shared among all cores on the chip. Niagara has a 3MB L2 cache shared among eight four-threaded cores. We simulate a processor with two cores – this is a quarter of Niagara’s size. Consequently, we configure our base L2 cache to be a quarter of Niagara’s, 768KB. (Although we do not claim that the relationship among the number of cores and the L2 cache size is linear, without the exact knowledge of the relationship this was the best approximation we could make.) We also perform experiments with several smaller cache sizes (between 48KB and 384KB) to demonstrate how CASC works in a cache-constrained environment.

5.2 Workload

Our goal was to choose a workload with a stable L2 miss rate over time, because we have not yet implemented a mechanism that deals with unstable behavior. We also wanted to run workloads that required no complicated set-up and little initialization time, because performing these tasks on a full-system simulator can require weeks or months of simulation. We also needed to have variability in L2 miss rates among the benchmarks: if all threads are identical, CASC cannot single-out cache-greedy threads and improve performance. (We discuss the viability of this requirement for commercial workloads in Section 5.4.1).

We analyzed L2 miss-rates over time of the SPEC CPU2000 benchmarks as well as BerkeleyDB [24], an embedded database library. Berkeley DB was running a hot-set workload (80% of accesses are to 30% of the database) on a 100MB database. We selected the following benchmarks: 300.twolf, 179.art and Berkeley DB. To create a multithreaded workload we run six copies of each benchmark. The SPEC benchmarks run with reference sets. Before execution we read the file set accessed by the benchmarks into the file system buffer cache to ensure that

the workload is CPU-bound. Each Berkeley DB benchmark runs on its own copy of a private database, so there is no lock contention.

5.3 Experiments

5.3.1 Performance improvement with CASC

To evaluate performance improvement with CASC we ran the workloads described in Section 5.2 on our simulated CMT system. We start the measurement after CASC computes PCMRs of all threads in the system. Therefore, we measure the performance of the system in stable state. The measured simulation lasts for 2 billion simulation steps, which roughly corresponds to 10-16 billion application instructions. During this period we measure L2 miss rates and processor throughput in billions of application instructions per second.

We kept statistics on how much time each thread spent executing on a processor. This allowed us to validate that the system was indeed executing the code of our

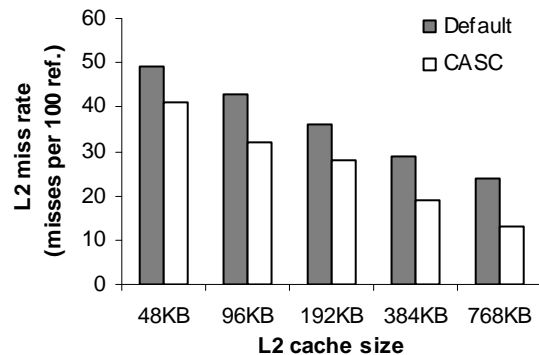


Figure 5. L2 miss rates with CASC and the default scheduler. CASC decreases L2 miss rates by 15-46%.

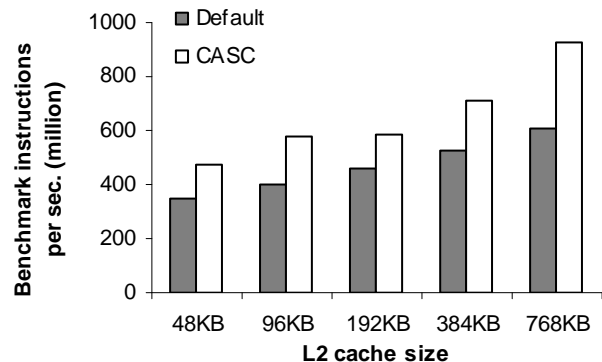


Figure 6. Combined instructions executed by all benchmarks per second with CASC and default scheduler. CASC improves throughput by 28-50%. Since all benchmarks were CPU intensive, this means that the benchmarks collectively accomplished more work with CASC than with the default scheduler.

workload the entire time during the experiment and that it did not achieve lower miss rates simply by running the idle loop due to excessive restriction of threads' CPU preferences. Since we started the measurement after all threads' PCMRs became known, there was no overhead due to the estimation of PCMRs in the scheduler. (We discuss the overhead incurred during the periods when PCMRs are estimated in Section 5.3.4).

Figures 5 and 6 present the results. CASC reduces L2 miss rates by 15-46%, which translates into throughput increase of 28-50%. Since the workload is CPU-bound and there is no scheduler overhead during the measured part of the experiment, this means that the benchmarks collectively completed more work per unit of time with CASC than with the default scheduler.

Note the trend in Figure 5. In all cases CASC achieves the L2 hit rate that the default scheduler achieves with an L2 cache that is at least twice as large. This means that for the workload we ran, CASC creates the illusion of having twice the L2 cache than the system really has.

The performance improvement with CASC is due to the fact that CASC is able to identify cache-greedy threads and restrict those threads' CPU preferences such that these threads are only scheduled in parallel with cache-frugal threads, and are never scheduled together. Although CASC does not identify cache-greedy threads with perfect accuracy (as we discuss in the next section), it is able to do so most of the time, and this is why it succeeds in achieving performance improvement. CASC does its best when it is able to pinpoint all or most of the cache-greedy threads. For example, in the experiment with the cache size of 768KB, CASC was able to categorize all six threads running 179.art as cache-greedy, while in other experiments it miscategorized several threads running 179.art. As a result, the experiment with the 768KB cache produced the best results: the 46% reduction in L2 miss rate and the 50% improvement in throughput.

In the next section we discuss how accurately CASC is able to distinguish between cache-greedy and cache-frugal threads. Then we discuss how the fact that CASC restricts cache-greedy threads to a limited subset of CPUs affects fairness.

5.3.2 CASC accuracy

We found that CASC is rather successful at distinguishing between cache-frugal and cache-greedy threads. We executed each benchmark in single-threaded mode and recorded its L2 miss rates over time. This data allowed us to tag threads with respect to their locality (179.art – poor locality, Berkeley DB and 300.twolf – good locality). CASC was successful at assigning high PCMRs to threads with poor locality and low PCMRs to threads with good locality. Out of the 18 threads in our workload, CASC correctly categorized localities of 12-16 threads in each experiment.

We found that for improved accuracy it is helpful to delay the beginning of initial or secondary measurement for each thread in order to allow the thread get past its initialization phase. We set this delay to 0.5 milliseconds. We also discovered that the L2 miss rates measured during the measurement intervals that were particularly short (under 200 μ s) were consistently larger than those measured during longer intervals. We concluded that this was due to cold start effects (bringing the data into cache on start-up). To compensate, we discard measurement intervals that are shorter than 200 μ s.

We noticed that most of the errors in categorizing threads with respect to their locality were made during the secondary measurement, as opposed to the initial measurement. The reason could be that while in the initial measurement we measure the L2 miss rate of a particular group of threads (i.e. the threads belonging to each row), in the secondary measurement a thread with unknown PCMR might get to run with different sets of threads during the measurement. We are currently investigating approaches to increase the accuracy of the secondary measurement.

5.3.3 Fairness

Fairness is frequently traded-off for performance in throughput-maximizing schedulers [27,34,35], and CASC is no exception. With CASC, cache-frugal threads get to run more often than cache-greedy threads. The important thing, however, is that cache-greedy threads are never entirely starved.

To gauge how CASC affects fairness, we kept track inside the kernel of how much time each thread executed with CASC and with the default scheduler. Figure 7 shows the respective distributions of threads’ running times over a period of two seconds with CASC and with the default scheduler. Each of the stacked boxes corresponds to a thread, and shows the fraction of total runtime that this thread received. It is evident that the distribution of thread runtimes (i.e. box heights) is more even with the default scheduler than with CASC. For CASC, we show the data from the experiment where the unfairness was the worst. Note the six short boxes in the middle of the diagram for CASC: they correspond to threads running 179.art – the cache-greedy benchmark. In the 768KB experiment, CASC categorized all six threads running 179.art as cache-greedy, and this is why these threads got scheduled less often than other threads and the performance improvement in this case was more significant than in other cases.

With CASC, the standard deviation from the average running time was 90% of the mean in the worst case; with the default scheduler it was 43%. (Recall that the default scheduler implements the multi-level feedback queue scheduling policy, so one cannot expect that threads’ running times will be perfectly evenly distributed.)

The impact on fairness often makes people uncomfortable using algorithms like ours for the purpose of maximizing throughput. It is often believed that such algorithms accomplish performance improvements by unfairly penalizing resource-intensive jobs in order to help less demanding jobs. Furthermore, some believe that it is not possible to improve performance of resource-frugal jobs without hurting resource-intensive jobs. Previous theoretical work investigated a class of scheduling

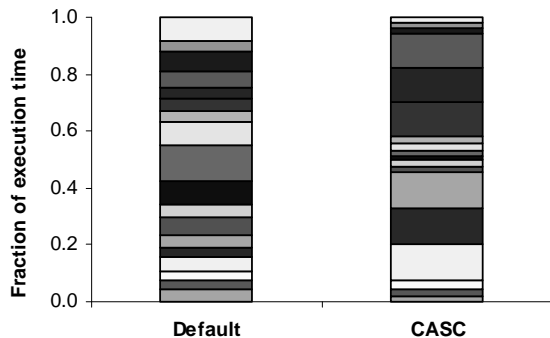


Figure 7. Fraction of total execution time occupied by each thread. The distribution is more even with the default scheduler (st.dev. is 43% of the mean) than with CASC (st.dev. is 90% of the mean). The data for CASC is for the experiment where the negative impact on fairness was the strongest.

algorithms similar to ours and concluded that this was not the case [27]. In particular, this study investigated shortest-remaining-processing-time-first (SRPT) – a class of throughput-maximizing algorithms for scheduling the processing of web requests. It concluded that a) these algorithms optimize throughput and mean response time; b) for job mixes where job sizes have a heavy-tail distribution performance improvement does not come at the expense of resource-intensive jobs, and in most cases all jobs, including the resource-intensive ones, perform better with SRPT than with traditional round-robin scheduling. The intuition is that since resource-intensive jobs achieve poor performance anyway, slowing them down by a little does not significantly change their total execution time.

Investigating the applicability of the SRPT study for CASC and the workloads that are expected to run on CMT systems is the subject of future work.

5.3.4 CASC overhead

Because CASC’s way of performing initial and secondary measurements is non-intrusive and the bucketization algorithm is run infrequently, the measurement phases in CASC do not add any measurable runtime overhead. We measured the overhead of CASC on a non-CMT UltraSPARC® II server with four processors, each running at 450 MHz. We opted not to run on a simulator in order to be able to collect more data points. Since there is no shared L2 cache on this machine, there would be no performance benefits from CASC, so we set the L2 cache miss rate threshold to 100% - this way it would always be satisfiable, and threads’ CPU preferences would not be restricted.

We ran a workload of twelve SPEC CPU2000 benchmarks with reference input sets. Our scheduler, as expected, performed the initial measurement and then a set of secondary measurements. After that, to simulate arrival of new threads in the system and re-estimation of PCMRs of old threads, we triggered periodic secondary measurement of PCMRs by marking the PCMR of a randomly selected thread unknown. We performed PCMR re-measurement as frequently as every 250 milliseconds, but observed no performance degradation: with the default scheduler it took 63 minutes to complete the entire workload, the same time that it took with CASC.

This result is not surprising: During the initial measurement, the threads run as they would with the default scheduler. CASC simply observes when a particular group goes on and off the processor and reads the hardware

counters. At the end of the initial measurement, CASC solves the system of linear equations. This operation takes about 400 μ s on our test system, but since it has to be done only once, it does not add much overhead. The secondary measurement is non-intrusive: the scheduler simply observes the running threads, reads the hardware counters and performs a simple computation at the end of the measurement. The bucketization algorithm takes about 800 μ s on our test system. However, it is also performed infrequently: it is first performed at the end of the initial measurement and then it is repeated whenever a PCMR computed during the secondary measurement is larger than the maximum PCMR present in the system or when the estimated L2 miss rate is above the threshold.

5.4 Discussion

5.4.1 The workload

Performance gains from CASC depend on the workload: the greater the variance among threads' individual cache miss rates the greater the benefit one can see from using the algorithm. If all threads are the same, there is no room for improvement. However, as we showed in the previous section, CASC would induce no performance penalty either.

Cain *et al.* have studied microarchitectural characteristics of several popular server benchmarks: SPECJBB, SPECWeb and TPC-W [8]. They found that there was significant variation in L2 miss rates among the benchmarks. Furthermore, different components of TPC-W produced different L2 miss rates. If each of these components is implemented in a separate thread, then the resulting workload will have threads with diverse L2 behaviors. Although not explicitly reported in the paper, we estimated that the standard deviation from the average L2 miss rate for these benchmarks was approximately 42%. The workload presented in this study had a standard deviation of 56%, supporting our hypothesis that such workloads will derive a benefit from CASC.

Obtaining a heterogeneous server workload can be done in the following ways: either by running a server that has variability among its threads (like TPC-W) or by running several server applications with different behaviors on the same machine. The latter is a common practice for the purpose of server consolidation. To streamline IT operations and simplify management, many companies consolidate their server applications on a single piece of hardware. VMWare [13], virtual machine software often used for server consolidation, permits running multiple

operating systems and servers on a single machine. With the advent of CMT systems server consolidation may become more popular, because CMT processors require a high degree of thread-level parallelism, which is easier to furnish if multiple server applications are run on the same machine.

5.4.2 Implications for scaling

Having an algorithm that improves management of the L2 cache enables aggressive scaling of CMT processors. As application demands grow and hardware technology advances, CMT processors will need to scale. The way to scale traditional processors is to increase clock frequency. The way to scale CMT processors is to increase the number of thread contexts per chip. Given that the L2 cache is a critical shared resource, each additional hardware context has to be matched with additional L2 cache space [1,2,7]. While placing additional thread contexts on a chip does not require many transistors [39], substantially increasing the L2 cache does. As silicon technology approaches the 65nm mark, fitting more components on a chip becomes increasingly more difficult [6]. This may be an impediment for scaling of CMT processors. Since CMT is the *de facto* platform for future servers, scaling CMT processors is crucial for our applications to run well in the future. Having an algorithm that improves L2 cache management facilitates aggressive scaling of CMT processors in spite of technological limitations.

6. RELATED WORK

Previous work proposing scheduling algorithms for single-core SMT processors has been shown to improve system response time by 10-17% [4,5]. These algorithms co-schedule threads on a multithreaded processor with the objective of minimizing contention for shared pipeline resources. They work by sampling the space of possible schedules and then using the ones that perform the best. Our method is different in that it uses a model to predict the best schedules, without having to run many potentially suboptimal schedules.

Kihm *et al.* proposed a cache monitoring technique for multithreaded processors [31]. Using inexpensive specialized hardware, they gain insight into how different groups of threads share cache lines. This information could be used to co-schedule threads in subsets that produce the least interference in the cache. The authors do not describe a scheduling algorithm that takes advantage of this cache-monitoring technique.

Although CASC has been inspired by balance-set scheduling [36] it differs substantially: in contrast with balance-set scheduling, which uses working set size as an indicator of performance in the cache, CASC uses individual threads' locality scores and predicts overall performance in the cache using these locality scores. Furthermore, we are not aware of a practical implementation of the balance-set scheduling algorithm, while CASC has been implemented in a running system and has been shown to produce substantial performance improvements.

7. CONCLUSIONS

In this paper we described CASC, a cache-aware scheduling algorithm, and discussed our implementation of CASC for Solaris 10. CASC improves performance by managing the L2 cache, a critical shared resource on CMT processors, a popular emerging platform for servers and high-end workstations.

CASC reduces L2 cache miss rates by 15-46% and increases processor throughput by 28-50%. For the workload we ran such an improvement is equivalent to doubling the size of the L2 cache. CASC achieves these performance improvements using a novel technique for estimating L2 miss rates of groups of threads at runtime, with no significant performance penalty. In addition to performance improvement, these results allow for greater scalability of CMT processors.

8. REFERENCES

- [1] S. Hily, A. Sez nec. Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading. *MTEAC'98*.
- [2] A. Fedorova, M. Seltzer, C. Small and D. Nussbaum, Performance Of Multithreaded Chip Multiprocessors And Implications For Operating System Design, *USENIX 2005*.
- [3] L. Spracklen, Y. Chou, and S. G. Abraham. Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications, *HPCA 2005*.
- [4] A. Snavely, D. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Machine. *ASPLOS IX*.
- [5] S. Parekh, S. Eggers, H. Levy, J. Lo, Thread-sensitive Scheduling for SMT Processors, <http://www.cs.washington.edu/research/smt/>
- [6] Gordon E. Moore. No Exponential is Forever...but We Can Delay 'Forever'. *Presentation at International Solid State Circuits Conference (ISSCC)*, February 10, 2003
- [7] L. Spracklen and S. G. Abraham. Chip Multithreading: Opportunities and Challenges, *HPCA 2005*.
- [8] Harold W. Cain, Ravi Rajwar, Morris Marden, Mikko H. Lipasti. An architectural evaluation of Java TPC-W, *HPCA'01*.
- [9] W. Wulf and S. McKee. Hitting the Memory Wall: Implications Of the Obvious. *ACM SIGARCH Computer Architecture News*, V.23, Issue 1, pages 20-24, 1995
- [10] X. Shen, Y. Zhong and C. Ding. Locality Phase Prediction. *ASPLOS 2004*.
- [11] R. Balasubramonian, S. Dwarkadas, D.H. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. *ISCA 2003*.
- [12] J. Mauro and R. McDougall. Solaris™ Internals. *Prentice Hall*, 2001.
- [13] VMWare, <http://www.vmware.com>
- [14] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. *ISCA'98*.
- [15] A. Ailamaki, D. DeWitt, M. Hill, D. Wood. DBMSs on modern processors: Where does time go? *VLDB '99*.
- [16] A. Barroso, K. Gharachorloo, E. Bugnion. Memory System Characterization of Commercial Workloads. *ISCA'98*.
- [17] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, Walter E. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP Workloads. *ISCA'98*.
- [18] D. Tullsen, S. Eggers, H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *ISCA'95*.
- [19] P. Kongetira. A 32-way Multithreaded SPARC(R) Processor. <http://www.hotchips.org/archives/hc16/>, *HOTCHIPS 16*, 2004.
- [20] <http://www.intel.com/employee/retiree/circuit/righthandturn.htm>
- [21] R. Kalla, B. Sinharoy, and J. Tendler. IBM POWER5 chip: a dual-core multithreaded processor. *IEEE Micro*, Vol. 24, No. 2, pp. 40-47, 2004.
- [22] Advanced Micro Devices. AMD Demonstrates Dual Core Leadership, <http://www.amd.com/>, 2004.
- [23] T. Maruyama. SPARC64 VI: Fujitsu's Next Generation Processor, *Microprocessor Forum 2003*.
- [24] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. *USENIX'99, FREENIX Track*.
- [25] B. Gamsa, O. Krieger, J. Appavoo, M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. *OSDI 1999*, pp. 87-100.
- [26] J. Laudon, A. Gupta, M. Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. *ASPLOS VI*.
- [27] N. Bansal and M. Harchol-Balter. Analysis of SRPT Scheduling: Investigating Unfairness, *SIGMETRICS 2001*.

- [28] D. Nussbaum, A. Fedorova, C. Small, The Sam CMT Simulator Kit, *Sun Microsystems TR 2004-133*, March 2004.
- [29] T. Krazit. IBM's Meyerson: Chip Industry Needs A Plan 'B'. *Computerweekly.com*, 7 October, 2004.
- [30] P. Magnusson et al. SimICS/sun4m: A Virtual Workstation, *USENIX Tech. Conf.*, June 1998.
- [31] J.L. Kihm and D.A. Connors. Implementation of Fine-Grained Cache Monitoring for Improved SMT Scheduling. *ICCD'04*.
- [32] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson and K. Chang. The Case for a Single-Chip Multiprocessor, *ASPLOS 1996*.
- [33] R. Eickenmeyer R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of multithreaded uniprocessors for commercial application environments, *ISCA'96*.
- [34] M. Seltzer, P. Chen, J. Ousterhout. Disk Scheduling Revisited, Proceedings of the USENIX Winter 1990 Technical Conference.
- [35] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps, *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2003.
- [36] P. Denning. Thrashing: Its causes and prevention. *Proc. AFIPS 1968 Fall Joint Computer Conference*, 33, pp. 915-922, 1968.
- [37] A. Fedorova, M. Seltzer, C. Small and D. Nussbaum, Throughput-Oriented Scheduling On Chip Multithreading Systems, *Technical Report TR-17-04*, Harvard University, August 2004.
- [38] E. Berg, E. Hagersten. Efficient Data-Locality Analysis of Long-Running Applications. TR 2004-021, University of Uppsala, May 2004.
- [39] Deborah T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, M. Upton, Hyper-threading technology architecture and microarchitecture. *Intel Technical Journal*, pp. 4-15, February 2002.