Throughput-Oriented Scheduling On Chip Multithreading Systems

Alexandra Fedorova,
Harvard University, Sun Microsystems
Margo Seltzer,
Harvard University
Christopher Small and Daniel Nussbaum
Sun Microsystems

Abstract

The unpredictable nature of modern workloads, characterized by frequent branches and control transfers, can result in processor pipeline utilization as low as 19%. Chip multithreading (CMT), a processor architecture combining chip multiprocessing and hardware multithreading, is designed to address this issue. Hardware vendors plan to ship CMT systems within the next year, so now is the time for us to understand how to get the most performance out of these systems.

In this work we demonstrate how to leverage operating system scheduling to improve performance on CMT systems, and keep them performing well even when contention for shared resources is high. We have studied several of the most-contended shared resources in CMT systems in order to understand how contention for these resources affects overall system performance. Having analyzed the processor pipeline, L1 data cache and L2 cache, we have found that contention for the processor pipeline is not likely to be an issue for typical workloads, poor hit rate in the L1 does not have a signification impact on overall throughput, but sub-optimal performance in the L2 does have a potential to noticeably hurt performance. To produce good L2 performance, we have used a scheduling policy based on the balance-set principle. With this approach, we are able to reduce miss rates in the L2 by 20-40% and improve system throughput by 16-32%. To achieve a similar improvement in hardware the size of the L2 cache would have to be quadrupled.

1. Introduction

Modern applications, such as application servers, web services, and on-line transaction processing systems, are notorious for poor utilization of the CPU pipeline. Such applications are usually comprised of multiple threads of control executing short stretches of integer operations, with frequent dynamic branches. This structure decreases cache locality and branch prediction accuracy and causes frequent processor stalls [11, 30, 31, 32]. Modern superscalar processors, using speculative and out-of-order execution, can wring instruction-level parallelism (ILP) from scientific workloads, but can do little for branch-heavy transaction processing-style workloads. Even some SPEC CPU benchmarks yield processor pipeline utilizations as low as 19% for some configurations [14].

Chip multiprocessing (CMP) and hardware multithreading (MT) techniques were designed to improve processor utilization for transaction-processing-style workloads by offering better support for thread-level parallelism (TLP). A CMP processor includes multiple processor cores on a single chip, which allows more than one thread to be active at a time and improves utilization of chip resources. An MT processor has multiple sets of registers and other thread state and interleaves execution of instructions from different threads, either by switching between threads (e.g. on every cycle) or by executing instructions from multiple threads simultaneously (if the instructions use different functional units) [4,5,6,7]. As a result, if one thread is blocked on a memory access or some other long-latency operation, other threads can make forward progress. IBM's Power4 and Sun's UltraSPARC IV® are CMP processors. Intel's hyper-threaded Pentium IV and IBM's RS64 IV are MT processors.

Numerous studies have demonstrated the performance benefits of CMP and MT [3-7, 13, 23, 29]. Driven by improvements in chip densities, hardware vendors are proposing architectures that combine CMP and MT. We refer to such processors as chip multithreading (CMT) processors. Major manufacturers such as Sun Microsystems, IBM, and Intel have announced plans to design such processors, and some of them plan to ship the first commercial CMT processors in 2005 [8, 9, 18].

It is critical for us to understand how CMT performs and how to best take advantage of it. Because CMT processors may be equipped with dozens of simultaneously active thread contexts, competition for shared resources is more intense than on conventional processors. It is important to understand the conditions under which these shared resources can become performance bottlenecks, as well as how to prevent performance degradation in such situations. We have investigated how to leverage the operating system scheduler to squeeze the most performance out of CMT processors by determining when shared resources—the pipeline and L1 and L2 caches—can become performance bottlenecks.

We have found that, given the nature of modern integer workloads, the processor pipeline is not likely to become a major performance bottleneck, even though it is shared among several simultaneously active threads. Due to the memory-intensive nature of these applications, the performance of the memory hierarchy is far more important. Having studied the performance of the L1 and L2 caches, we have concluded that poor L1 data cache hit rates do not have a significant effect on system throughput, but a poor hit rate in the L2 cache can severely hurt performance and leave the processor pipeline under-utilized.

To improve performance of the L2 cache, we have developed a scheduling technique based on the balance-set principle [21]. The objective of the balance-set principle is to avoid thrashing: it suggests that the scheduler should schedule a group of threads to run simultaneously as long as the combined working set of this workload fits in the cache. We have found that a scheduling policy based on this principle can reduce the L2 cache miss rates by 20-40%, yielding a performance improvement of 16-32%. A similar improvement could have been achieved by increasing the size of the L2 cache, but the increase would have to be a factor of four. We believe that fixing the scheduler is preferable.

We discovered that the balance-set method in its classical incarnation does not work well for modern workloads, because it involves modeling cache miss rates using working set size [20, 21, 22]. We explain why the classical model fails to work, and demonstrate our success using a new model [19] that is based on reference locality.

Using software mechanisms to best exploit existing hardware not only allows us to get more value for our money, but also helps us build systems that can continue to provide good performance as applications evolve. While hardware designers do their best to make processors work well with the workloads that they expect customers to run on their systems, accurately predicting what these workloads will look like in the future is difficult, if not impossible. Since hardware cannot adapt as quickly as applications change, it is important to design software solutions that will keep systems running well when applications evolve.

The rest of this paper is structured as follows: in Section 2 we provide background on CMT systems, describe the CMT model that we are using, and discuss our simulation technology. In Sections 3, 4, and 5 we analyze shared resources on a CMT processor as potential performance bottlenecks: the processor pipeline, the L1 data cache, and the L2 cache, respectively. In Section 6 we describe how we used balance-set scheduling to improve performance of the L2 cache. We discuss related work in Section 7 and conclude in Section 8.

2. Background and system model

The systems addressed in this study have multiple processor cores on a chip (chip multiprocessing [24]) and multiple hardware contexts on each processor (hardware multithreading).

There are several ways to implement hardware multithreading, and they can be broadly categorized as *coarse-grained*, *fine-grained,* and *simultaneous*. The main difference between these categories is how the hardware switches among software threads.

Coarse-grained multithreading switches to a new thread when a thread occupying the processor blocks on a memory request or other long-latency operation [5]. While performance benefits of this architecture have been demonstrated for multithreaded workloads [23], it has also been shown that performance is limited on such architectures by the high cost of context switching [6].

This problem is addressed on *fine-grained multithreaded,* or interleaved, architectures, which switch threads on every cycle [6].

Simultaneous multithreading (SMT) architectures add multi-context support to multiple-issue, out-of-order processors. Unlike conventional multiple-issue processors, they can issue instructions from different instruction streams on each cycle, leveraging thread-level parallelism for improved instruction-level parallelism [7, 14].

Our model of a multithreaded processor core is based on the concept of fine-grained multithreading (interleaving), proposed by Laudon et al. [6]. An MT core has a small number of hardware contexts (two, four or eight), where each context consists of a set of registers and other thread state. Such a processor interleaves execution of instructions from the threads, switching between contexts on each cycle. A thread may become blocked when it encounters a long-latency operation, such as a cache miss. When one or more threads are blocked, the system continues to switch among the remaining available threads. For multithreaded workloads this improves processor utilization and hides the latency of long operations.

Since, CMT systems do not yet exist, for the purposes of our study we have built a CMT system simulator toolkit [16] as a set of extensions to the Simics simulation toolkit [15]. Our toolkit models systems with multiple multithreaded CPU cores. The number of CPU cores per chip and the degree of hardware multithreading are configurable.

Our simulated CPU core has a simple RISC pipeline with one set of functional units (arithmetic unit, load/store unit, etc.). Each core has a single TLB and L1 data and instruction caches, shared by multiple threads. The (integrated) L2 cache is shared by all of the CPU cores on the chip.

We have decided to simulate a simple, classical RISC core, as opposed to a complex out-of-order processor. The advantage of a simple processor core is that it is smaller and allows space for more cores on each chip. For thread-rich workloads, studies have shown that chip multiprocessors with simple cores, such as ours, outperform chip multiprocessors with fewer complex cores [24]. Moreover, we believe that the results of our study that concern the memory hierarchy are applicable to a wide range of multithreaded architectures, since the architecture of the memory hierarchy does not depend on the architecture of the pipeline.

A schematic view of a possible configuration of a CMT processor is shown in Figure 1.

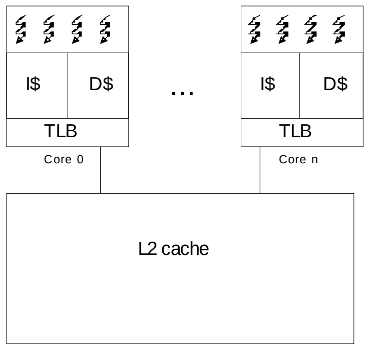

Figure 1. A schematic view of a CMT processor

Our simulator accurately simulates pipeline contention, the L1 cache, bandwidth limits on crossbar connections between the L1 and L2 caches, the L2 cache, and bandwidth limits on the path between the L2 cache and memory. We do not simulate a shared TLB; our measurements have shown that the TLB is not a highly contended resource for the benchmarks we ran. Our simulator is typically configured with one to four processor cores. Each core includes four hardware contexts, an 8KB L1 data cache and a 16KB L1 instruction cache (both 4-way set-associative). We also simulate a unified 12-way set-associative L2 cache, whose size we vary depending on the experiment. We chose cache sizes to be similar to those used in the hyper-threaded Pentium IV, a one-core multithreaded processor that is commercially available at the time of this writing [13].

3. Processor pipeline

In this section we discuss the effect of contention for the processor pipeline on overall system performance. We show that, for workloads composed of threads with significantly different pipeline requirements, an intelligent scheduler can increase performance by as much as a factor of two. Unfortunately, we also find that real workloads tend to be relatively homogeneous and that the potential performance gains using this technique are small.

Threads differ in how they use the processor pipeline. Compute-intensive threads with predictable branch targets, such as scientific workloads, issue instructions frequently and utilize the pipeline intensively. Memory-intensive threads frequently stall while waiting for a response from the memory hierarchy. Figure 2 illustrates how these threads utilize the pipeline.

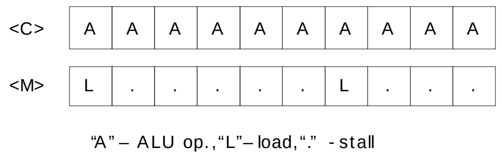

Figure 2. Compute-intensive vs. memory-intensive thread.

This picture suggests a scheduling policy that should improve pipeline utilization. Imagine what happens when compute-intensive and memory-intensive threads are assigned to multithreaded processors. When several compute-intensive threads are running on the same core, contention for pipeline resources is severe. On the other hand, memory-intensive threads often leave the pipeline under-utilized. Figures 3a and 3b illustrate this.

The answer is to co-schedule compute-intensive and memory-intensive threads for the best resource utilization (Figure 3c). This idea is similar to *paired gang scheduling* [33], a technique for scheduling on parallel supercomputers that matches compute-intensive and I/O intensive jobs for optimal resource utilization.

A scheduler can estimate pipeline requirements of a workload in order to identify compute-intensive and memory-intensive threads by measuring the workload's *single-threaded CPI*, a cycles-per-instruction metric that would result if the workload were scheduled to run on its own processor. Workloads with low CPIs (those nearing 1 on our simple pipeline) have high pipeline utilization, and vice versa.

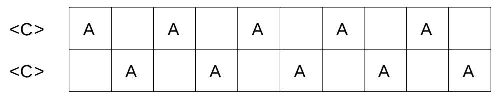

Figure 3a. Co-scheduling compute-intensive threads: While each thread is able to issue an instruction on every cycle, it cannot do so, because the processor has to switch among all the threads.

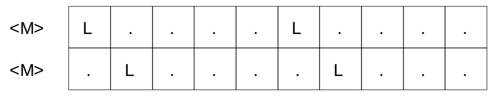

Figure 3b. Co-scheduling memory-intensive threads

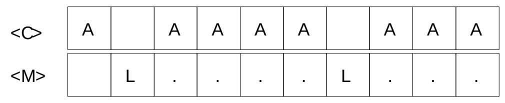

Figure 3c. Co-scheduling compute-intensive and memory-intensive threads.

We have looked into potential performance gains from such a scheduling policy, by running a simple experiment with microbenchmarks. We constructed microbenchmarks with the following single-threaded CPI's: 1, 6, 11, and 16, to simulate compute-intensive and progressively more memory-intensive workloads. Then, on a machine with four processor cores and four thread contexts on each processor, we tried several ways to schedule 16 threads: four each with CPIs of 1, 6, 11 and 16.

Table 1 illustrates four different schedules that we have tried. Schedules (a) and (b) match compute-intensive threads with memory-intensive threads, and are expected to perform better than schedules (c) and (d) that place compute-intensive threads on the same core.

Figure 5 shows the throughput achieved by each schedule in ops/sec, where an op is an iteration of the inner loop of the microbenchmark (1024 instructions). As expected, schedules (a) and (b) achieve the best performance[1]. The performance difference between schedule (d) and schedules (a) and (b) is a factor of two.

	Core 0	Core 1	Core 2	Core 3
(a)	1, 6, 11, 16	1, 6, 11, 16	1, 6, 11, 16	1, 6, 11, 16
(b)	1, 6, 6, 6	1, 6, 11, 11	1, 11, 11, 16	1, 16, 16, 16
(c)	1, 1, 6, 6	1, 1, 6, 6	11, 11, 11, 11	16, 16, 16, 16
(d)	1, 1, 1, 1	6, 6, 6, 6	11, 11, 11, 11	16, 16, 16, 16

Table 1: Assignment of threads to cores for schedules (a)-(d). Numbers in cells show the single-threaded CPIs of threads assigned to this core.

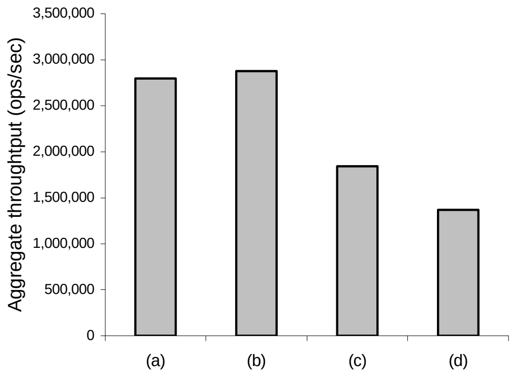

Figure 5. Aggregate throughput (in ops/sec) achieved by each schedule.

However it is necessary to have threads with widely varying single-threaded CPIs to achieve such a substantial performance improvement. We looked at SPEC benchmarks, measuring single-threaded CPI (Table 2) and the dynamic instruction mix (Figure 6). Based on these data, we expect to see little variation in how real workloads use the processor pipeline, and that there therefore would be little potential for performance gains from exploiting this variation. Our experiments with the SPEC benchmarks have confirmed that performance gains from scheduling based on pipeline usage are negligible for integer workloads.

While the benefits of CPI-based scheduling for SPEC workloads are small with our simple pipeline, they may be greater on an SMT system. Recall that SMT systems have multiple functional units and a multiple-issue pipeline. The range of single-threaded CPIs for the integer workloads presented in Table 2 may be greater on such machines. Since our simulator does not have capabilities for simulating a super-scalar

[1] Schedule (b) performs slightly better than schedule (a) because workloads with CPI 16 that dominate core 3 in schedule (b) have a smaller I-cache footprint and achieve a better I-cache hit rate.

pipeline, we could not confirm whether this would be the case. But we think that this would be an interesting area to explore.

	Average CPI	St. dev.
SPEC JVM		
compress	2.87	0.62
db	3.62	0.61
jack	3.78	0.71
javac	3.58	0.51
jess	3.65	0.49
mpeg	4.51	1.61
mtrt	3.31	0.72
SPEC CPU		
bzip2	2.50	0.80
crafty	2.98	0.25
gap	3.70	1.48
gcc	3.23	0.53
gzip	2.37	0.37
mcf	5.11	4.15
parser	3.55	0.30
perl	3.70	0.63
vortex	3.35	0.71
vpr	4.22	0.50
wolf	3.93	0.21
Server benchmarks		
SPEC Web Apache	4.33	0.31
SPEC JBB	3.93	0.17

Table 2. Average CPI of some integer benchmarks.

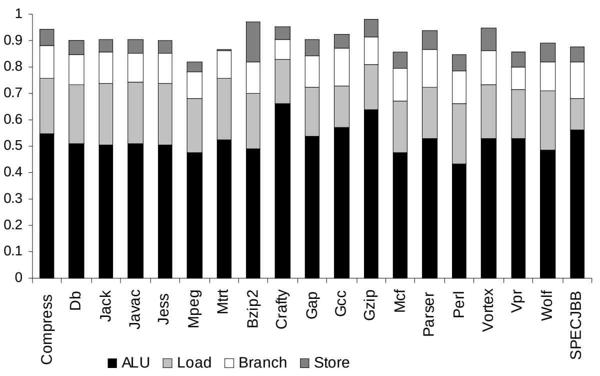

Figure 6. Fractional instruction mixes of integer workloads from SPEC JVM suite, SPEC CPU (int), and SPEC JBB.

4. L1 data cache

Multithreaded processors are typically configured with small L1 data caches, e.g., the L1 data cache on Intel's hyper-threaded Pentium IV is 8 KB [13]. We were concerned that this would turn out to be a performance problem when multiple applications share the cache. As it turned out, multithreaded processors tend to be immune to poor L1 utilization, and, therefore, performance optimizations that target L1 utilization do not have much of an effect on the overall system throughput.

We began by measuring the L1 data cache hit rates for some SPEC benchmarks on (simulated) machines with 8KB L1 data caches. We ran on a single-threaded machine to obtain a baseline measurement, and again on a multithreaded machine with four hardware contexts. When running on the multithreaded machine, we ran four copies of the same workload. The results are summarized in Table 3. We show the results for these benchmarks because they represent a reasonable mix of different cache behaviors, crafty and SPECJBB being very poor, gzip and Berkeley DB being relatively good, and vpr-place in between.

Application	baseline	four-way MT
crafty	89%	68%
gzip	93%	86%
vpr-place	89%	71%
Berkeley DB	97%	78%
SPEC JBB	84%	81%

Table 3. L1 data cache hit rates for baseline single-threaded and four-way multithreaded (MT) configurations. Data cache size is 8 KB.

Except for Berkeley DB, the cache hit rates for the four-way MT machine are well below what is considered to be “good” for conventional single-threaded processors. However, even with such poor cache hit rates, pipeline utilization for the four-way MT configuration is fairly high, between 0.7 and 0.9. Figure 7 shows the pipeline utilization, or IPC[2], with larger cache sizes.

Simulating larger cache sizes and examining how the IPC changes as cache hit rates become higher has allowed us to get a feel of how much of a performance improvement we could expect if we had implemented a scheduler that could dramatically increase hit rates. The results show that the potential for performance improvement is not great.

Increasing the size of the L1 data cache to 128KB, a factor of 16, resulted in hit rates increasing to above 90% for all workloads but only a 7% increase in

[2] Our simple pipeline issues at most one instruction each cycle. The highest IPC it is able to obtain is one. Therefore, the metrics for pipeline utilization and the IPC are equivalent in our configuration.

throughput on average. This tells us that even if we were to develop a technique that reduced cache miss rates by 10-20% the payoff in terms of improved performance would be small.

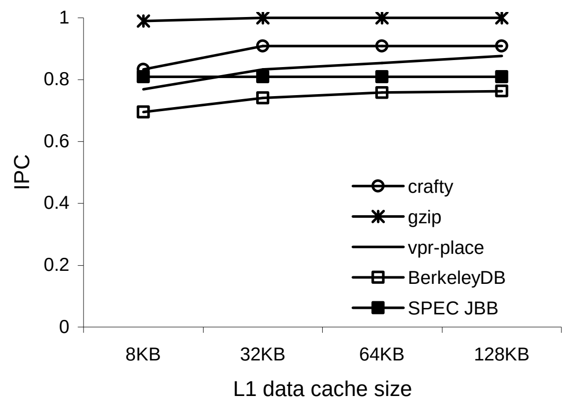

Figure 7. Pipeline utilization as function of cache size.

Similar results, showing the relationship between L1 cache hit rates and performance, have been described in a study on effects of affinity-based scheduling [25]. This work showed that increasing cache hit rates by 7-36% (13% on average) has resulted in maximum performance improvement of 10% (6% on average).

In fact, the theoretical basis for this was established a long time ago, in Denning's work on the causes of thrashing [21]. Denning showed that poor cache hit rates will severely degrade performance only if the cost of a cache miss is several orders of magnitude greater than the cost of a cache hit. This is not the case for modern machines, where the difference between hitting in the L1 and hitting in the L2 is typically within an order of magnitude. For example, hyper-threaded Pentium IV has the L1 latency of two cycles, and L2 latency of 18 cycles [13]. In our model, these latencies are two and 22 cycles respectively.

Although the main value of this result for us was that it helped us understand that it is not worthwhile to optimize for performance of the L1, it also demonstrates the key benefit of multithreaded processors: even with poor cache hit rates, it is possible to hide the resulting memory latency and keep the pipeline utilization high.

This section demonstrated that small size of the L1 data caches is not an issue for performance on MT processors. L1 instruction caches on MT processors also tend to be small (12 KB on the hyper-threaded Pentium IV, 16 KB in our model). However, we have observed that all our workloads achieved good instruction cache hit rates (above 97%) when running in the multithreaded mode, and, therefore, did not consider performance optimizations targeted at the instruction cache.

5. The L2 cache

The L2 cache has a greater potential for becoming a performance bottleneck when its hit rate is poor, because the latency between the L2 and main memory is significantly greater than that between L1 and L2. For example, on the hyper-threaded Pentium IV, a trip from L1 to L2 takes 18 processor cycles, while a trip from L2 to main memory takes 360 cycles [13]. These latencies are set at 22 and 120 cycles in our model. A secondary effect of poor hit rate in the cache is that it may result in saturating the bandwidth between the cache and the next level of the memory hierarchy. The bandwidth between the L1 and L2 caches is typically greater than that between the L2 cache and the main memory. This also contributes to making a cost of missing in the L2 higher than that of missing in the L1. We have confirmed that poor hit rate in the L2 may pose a significant performance problem with the experiment presented in this section.

Once again, we ran this experiment on a single-core MT machine with four thread contexts, an 8KB L1 data cache and L2 caches of varying sizes.

In our L2 cache experiments we use a workload consisting of eight applications. Four are taken from the SPEC CPU benchmark suite (crafty, gzip, vpr-place and vpr-route), and the other four are synthetic benchmarks that we constructed to represent a wide range of cache behaviors (from very poor to very good). We chose these workloads because we understood their cache behaviors well, and we were sure that this workload mix included programs with both good and bad cache behavior. We needed to have such a workload mix to be sure that our scheduling algorithm (discussed in Section 6) does not starve workloads with bad cache behavior. We think that this workload mix is acceptable to use for a proof-of-concept experiment. (In future work we plan to evaluate our techniques using commercial workloads.)

Our simulator boots the Solaris™ operating system, and so our eight-program workload is run on top of Solaris™. To the operating system, each hardware context of a multithreaded processor looks like a regular processor in a multiprocessor system. The OS scheduler assigns threads to hardware contexts as it would assign them to processors on an SMP, picking four threads at a time to schedule on the four hardware contexts. We vary the size of the L2 cache and show how this affects L2 hit rates and performance. The hyper-threaded Pentium IV has a 256 KB L2 cache. We simulated both larger and smaller caches to explore effects of light and heavy cache contention on performance, without having to vary the workload. Figure 8 shows the processor IPC resulting from running the eight-program workload.

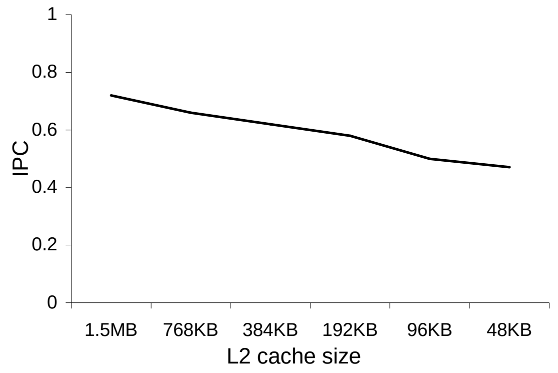

Figure 8. Processor IPC as function of L2 size.

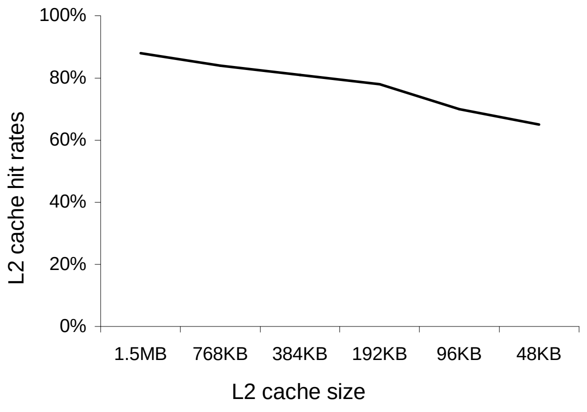

Figure 9. L2 hit rates as function of L2 size.

As Figure 8 shows, performance degradation is evident as the L2 cache becomes smaller. Figure 9 shows the corresponding decrease in the L2 hit rates.

In the following section we describe how we leveraged scheduling to alleviate pressure on the L2 cache and improve system throughput.

6. Scheduling

Having seen that the L2 cache has the most potential for becoming a performance bottleneck, we developed a scheduling technique that would improve the L2 hit rate. We were able to successfully leverage the balance-set approach suggested by Denning [20, 21], achieving a performance improvement of 16-32% over the default scheduler, and decreasing the L2 miss rates by 20-40%.

The basic idea behind balance-set scheduling is simple: to avoid thrashing, only schedule a group of threads whose working set fits in the cache. Making sure that the working set fits will result in good cache hit rates and prevent thrashing. However, we found that a workload's working set size is not always a good indicator of the cache hit rate that the workload will produce. While working-set-based models have been shown to work well for workloads typical of computer systems 20 years ago [22], they do not work well for modern workloads. Therefore, in order to successfully use the balance-set principle we needed to find a better way to model the cache behavior of a workload. In the next section we explain why the classical working set model did not work well for us and describe the cache miss rate model that we use.

6.1. Problems with the working-set model

A classical way to model cache miss rate based on the working set size of the workload relies on the premise that the size of the working set is a good indicator of a program's locality [21]. In his original work describing the working set model, Denning makes the assumption that the working set is accessed uniformly. Under this assumption, a large working set size will have poor locality, and a small working set will have good locality.

We originally attempted to use the Agarwal-Horowitz-Hennessy (AHH) model, developed in the 1980s, which was based on this principle, and has been used to produce accurate predictions in the past [22]. We were hoping that knowing the working set sizes of workloads would help us predict the cache hit rate they produce when scheduled together. In order to use a working-set based model, one must observe the size of the data set that a program touches over a period of time. Then, the size of this data set (the working set) is used to predict the cache miss rate that this workload will produce on a given cache architecture.

To our disappointment we saw unacceptably high errors in our miss rate estimates, which were off by as much as a factor of two in some cases. One parameter of the model is the period of time over which the working set is measured. Experimenting with different time intervals did not help.

A puzzling series of results led us to suspect that the working set model was not appropriate for our applications. We began by measuring the cache footprint for each application (the smallest cache size sufficient for the application to maintain its best achievable cache hit rate—the hit rate the application exhibits when run with an infinitely large cache). The footprint for gzip was about 200KB; for crafty it was about 40KB. According to the working set model, crafty should have experienced better cache hit-rates than gzip. However, this was not the case (mystery #1). With an 8KB L1 data cache, run on a baseline single-threaded configuration, crafty achieved a hit rate of 89%, while gzip had a hit rate of 93%. When we ran crafty and gzip on MT machines with four contexts, the 4-crafty workload produced a hit rate of 68%, and the 4-gzip workload produced a hit rate of 86% (mystery #2). And when we simulated larger caches, gzip's cache hit rate, relative to crafty, began to decline (mystery #3). The results are summarized in Table 4.

A model based on cache working set could not predict such behavior in principle, because according to it, a workload with a small working set should consistently produce higher cache hit rates than a workload with a large working set, regardless of the choice of interval over which the working set is measured and other parameters. This prompted us to search for a different way to predict a program's cache behavior.

	baseline	**four-way MT**			
cache	***8KB***	***8KB***	***32KB***	***64KB***	***128KB***
crafty	89%	68%	86%	92%	95%
gzip	93%	86%	89%	90%	92%

Table 4. L1 data cache hit rates for crafty and gzip. **baseline** denotes a configuration when an application is running on its own single-threaded processor, **four-way MT** denotes a configuration when four copies of the same application are run on a single multithreaded core with four contexts.

6.2. A better metric of locality

The difficult-to-explain cache behavior of crafty and gzip led us to believe that there was something inherently different about the pattern in which these two programs accessed their working sets. We hypothesized that even though gzip has a larger overall working set, it must access it in small chunks, and get good cache hit rates as a result. On the other hand, crafty, although it has a smaller overall working set, must have poor locality of reference. To test our hypothesis we measured the degree of locality exhibited by gzip and crafty. Degree of locality can be represented as the distribution of *reuse distances*—reuse distance is the amount of time[3] that passes between references to a memory location. A distribution of reuse distances allows us to evaluate a program's locality: if reuse distances are small, the degree of locality is high and vice-versa. Taking a look at such distributions for crafty and gzip explained the strange cache behavior that they were exhibiting. Figures 10 and 11 illustrate this.

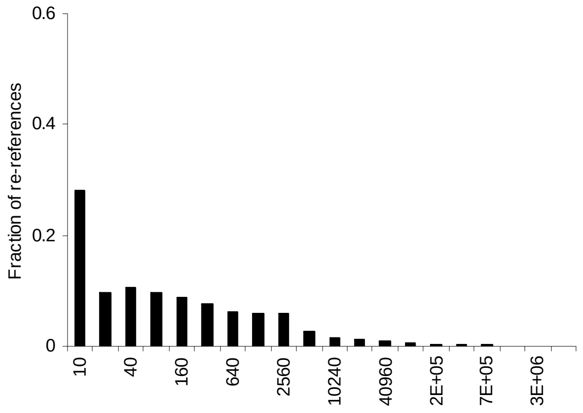

Figure 10. Reuse distance distribution for crafty.

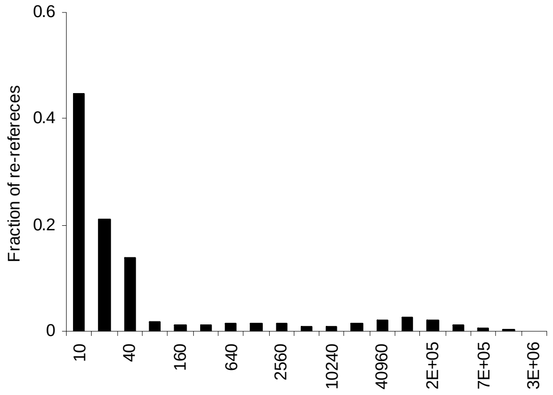

Figure 11. Reuse distance distribution for gzip.

It can be seen from the graphs that for gzip a large fraction of its memory re-references fall within a short time interval. For crafty, reuse distances are widely scattered.

These results led us to conclude that working set size is not a good way to measure program locality, because the assumption that the program accesses its working set uniformly does not hold for modern workloads. Based on the characteristics of workloads *circa* 1980, this assumption was probably quite reasonable. It explains why the AHH cache model produced accurate cache miss rate predictions at that time [22].

One might argue that we could have computed the working set over a very short time interval. If we looked at the working set over a tiny time interval, we might have seen that gzip's working set is as small or smaller than crafty's. But just changing the interval for

[3] Time is measured in terms of memory references.

measuring the working set would not solve the problem: as we saw, for larger caches gzip's cache behavior is worse than crafty's, due to gzip's working set being larger than crafty's if measured over a larger time interval. To detect such behavior, one would need to consider not just a single measurement for the working set, but perhaps a series of measurements performed over several different time intervals. And this is quite similar to using the reuse distance distribution.

6.3. The reuse-distance model

Recent work of Berg and Hagersten describes a cache model based on reuse distance distributions [19]. We will refer to this model as the *reuse-distance model*. We found the reuse-distance model to be extremely accurate: using it, our error rates for predicted cache miss rates were within a factor of 1.06 on average.

The model bases the probability of a cache hit on the time (number of memory references) since that location was last referenced (its *reuse distance*). The smaller the reuse distance, the greater the probability that the reference will result in a hit.

This model assumes a fully associative cache with a random replacement policy. The authors, however, found that predictions were accurate for the LRU replacement policy as well. We found that it is also accurate for set-associative caches.

The input necessary for this model is the *reuse distance histogram*. A reuse distance histogram is just a non-normalized version a reuse distance distribution. Instead of counting the fraction of re-references that fell within a certain distance, the histogram counts the total number of re-references that fell within this distance. In Section 6.6 we discuss how the data necessary to build a reuse distance histogram could be collected at runtime by the scheduler and what the associated costs might be.

For our purposes, we needed to adapt the reuse-distance model that predicts miss rates for individual threads to predict cache miss rates for multiple threads using the cache simultaneously. We developed two methods: COMB, which combines the reuse distance histograms for several threads, and AVG, which averages predicted miss rates for smaller caches.

We describe the methods in further detail below, but first we demonstrate the accuracy of predictions for multi-threaded workloads. Figure 12 shows actual and predicted miss rates obtained using the two methods. The actual miss rates were measured during the scheduling experiments with different L2 cache sizes that are discussed more in the next section. The predicted miss rates are the miss rates that the scheduler generated for each schedule before running this schedule (also discussed in the next section). As you can see, both methods are quite accurate. We will now describe these methods in detail.

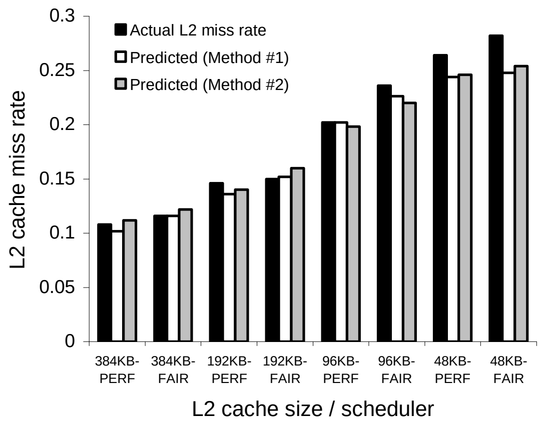

Figure 12. Actual vs. predicted miss rates.

6.3.1. Method #1: COMB

COMB combines the reuse distance histograms for multiple workloads that were collected for each workload in isolation and predicts the miss rate for the group of workloads, using the resulting combined histogram. To combine several histograms into one, we have designed a simple two-step algorithm:

1. For each reuse distance bucket, sum the number of references falling into this bucket for all histograms.
2. Multiply the value of each reuse distance appearing in the histogram by N or N-1, where N is the number of combined histograms (i.e. the number of threads).

Step 2 is necessary because when several threads share the cache the reuse distance for each memory re-reference will be increased. For example, if a thread running on its own re-referenced a memory location within one time step, when it runs with three other threads, before it re-references that memory location, the other threads may intervene with their own memory references. As a result, the original reuse distance of 1 could become anywhere between 1 and 4. We have experimented with multiplying the reuse distance by coefficients from 1 to N, where N is the number of histograms (number of threads) being combined. We found that using a coefficient of N-1 works better for larger caches, and a coefficient of N works better for smaller caches. But even if the coefficient of N-1 is used independently of cache size, predictions are still quite accurate (within a factor of 1.1).

The drawback of this approach is that it is probably too computationally expensive to implement on a real system. Imagine a machine with 32 hardware contexts and 100 threads. Every time a scheduler has to make a decision, it has to potentially combine (100 choose 32) histograms and compute the predicted miss rates. Our second approach is much less expensive but works just as well.

6.3.2. Method #2: AVG

AVG, the second method, uses the reuse distance histograms for individual workloads and predicts the miss rate that would result if each workload were run with a fraction of a cache equal to (TOTAL_CACHE / NUMBER_OF_THREADS). Then, individual predicted miss rates are averaged to get the projected miss rate for the combination of workloads. The intuition here is to pretend that each thread runs with its own dedicated partition of a larger cache. We found that predictions based on this method are just as accurate as those based on the previous method, as Figure 12 shows.

This method of predicting miss rates is preferable for a real implementation, because it is less computationally expensive. The prediction has to be made only once for each workload, and then, to project miss rates for multiple workloads, the scheduler needs only to average several values. Furthermore, the scheduler can remember averages for small groups of threads, and then, to predict the miss rate for a larger group, the scheduler can simply merge the known averages.

6.4. Balance-set scheduling

This section describes how we were able to leverage the reuse-distance model and balance-set scheduling to improve system performance and decrease the L2 miss rates. The experiments that we run here use the machine configuration and the workload described in Section 5.

Our scheduling algorithm works as follows. Instead of following the classic balance-set principle, which suggests scheduling a workload only if its total working set fits in the cache, we adapted this principle to work with reuse-distance model as follows. Each time we make a scheduling decision, consider the predicted miss rates for all possible groups of threads, and consider for scheduling those whose predicted miss rates are below a given threshold.

In a programmatic environment the scheduler could choose the miss rate threshold by observing the L2 cache miss rates and the corresponding pipeline utilization, and identifying the miss rates that keep the pipeline well utilized (i.e., at or above 60-70%).

For the purposes of this study, we chose miss rates thresholds to be as low as possible, but that they were not too low so as to exclude cache-greedy workloads from a set of schedulable groups. For example, if the miss rate threshold is chosen to be too low, it is possible that thread groups that satisfy this threshold include only workloads with cache-frugal behavior. As a result, our scheduler would starve the cache-greedy threads.

Table 5 summarizes the miss rate thresholds we used for all cache sizes, and the miss rates that were obtained with the default scheduler (recall the experiment in Section 5).

L2 cache size	Miss rate under default scheduler	Miss rate threshold used
384 KB	19%	13%
192 KB	22%	19%
96 KB	30%	25%
48 KB	35%	27%

Table 5. L2 miss rates with the default scheduler and miss rate thresholds used for balance-set scheduler.

The miss rate thresholds are lower than the miss rates obtained with the default scheduler, yet they do not exclude any workload from the set of schedulable four-tuples. If many threads on the system happen to be cache-greedy, it may not be possible to schedule a thread on every hardware context and still satisfy the miss rate threshold. In this situation, it may be preferable to leave some hardware contexts unused – thrashing would prevent them from being fully utilized anyway. Tradeoffs involved in making this decision should be investigated further.

We emulated the actions of balance-set scheduling in the following way: for a given cache size, we predicted cache miss rates for all groups of four threads from the eight-thread workload we described in Section 5. Then, from all the groups whose predicted miss rate satisfied the threshold (see Table 5), we chose the groups to schedule. We simulated each scheduled group for about 400 million cycles. We report the throughput and miss rates achieved over about two billion cycles.

We tried two policies when selecting the groups of threads to schedule: performance-oriented (PERF), and fairness-oriented (FAIR). With PERF, we selected the groups whose predicted miss rate was the lowest, without any regard for fairness. However, we also made sure that no workload was starved. With FAIR, we attempted to make sure that each workload receives an equal share of the processor. As we selected groups of threads we kept track of how many times each of the workloads had been selected. Each time a selection was made, we would favor groups that contained the least frequently selected workloads. Although we do not claim to have thoroughly addressed fairness with this policy, we wanted to see how well we could do with this simple approach. We will discuss how these two policies compare in terms of fairness in the next sub-section.

Figure 13 presents the throughput (processor IPC) achieved with the default Solaris™ scheduler, and the emulated balance-set schedulers: PERF (the one that used performance-oriented policy), and FAIR (the one that used fairness-oriented policy).

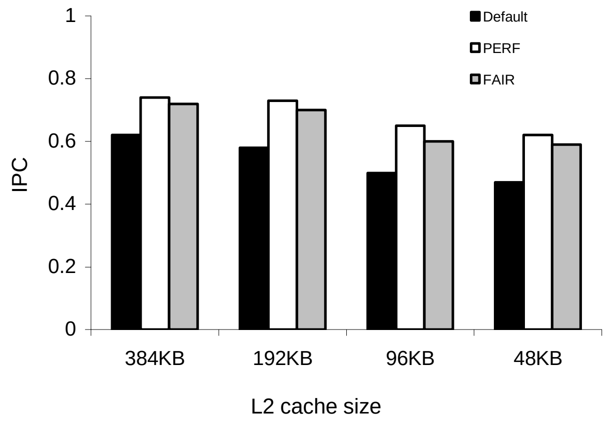

Figure 13. IPC achieved with the default scheduler, PERF, and FAIR.

In all cases, the balance-set scheduler outperforms the default scheduler: the lowest performance gain is 16% with the FAIR scheduler when the L2 size is equal to 384KB, and the highest is 32% with the PERF scheduler when the L2 size is equal to 48KB. As cache pressure becomes more severe, there is more benefit to be reaped from the balance-set scheduling approach. As the cache size becomes larger and contention becomes lower, we expect to see less benefit from the balance-set scheduler: if the performance achieved with the default scheduler is already good, there is less room for improvement.

Figure 14 shows the corresponding L2 miss rates. With balance-set scheduling we were able to reduce the L2 miss rates by 20-40%. It is interesting to place this result into perspective by evaluating the means that would be necessary to achieve similar improvements in hardware. With a 384KB L2 cache, the balance-set FAIR scheduler achieves a miss rate of 12%. To achieve a similar miss rate with the default scheduler, the L2 cache size would have to be 1.5 MB – four times larger. The same analysis applies to the rest of the cache sizes.

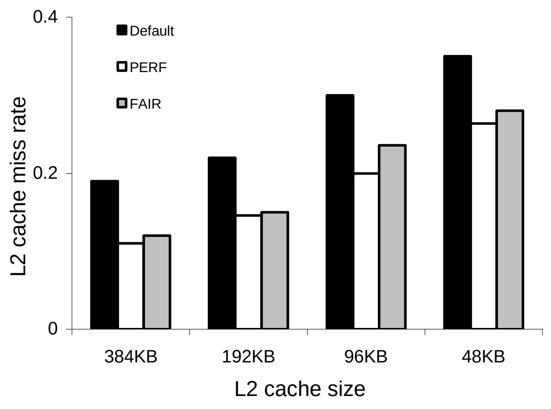

Figure 14. L2 cache miss rates achieved with the default scheduler, PERF, and FAIR.

6.5. Fairness

Although addressing fairness of the balance-set scheduling was not the objective of this study, fairness merits some discussion. The inevitable trade-off one has to pay with balance-set scheduling is fairness. A scheduler that is trying to optimize for cache hit rate will inevitably favor the well-behaving workloads. In the previous section we described the two thread-selection policies that we have used: performance-oriented and fairness-oriented.

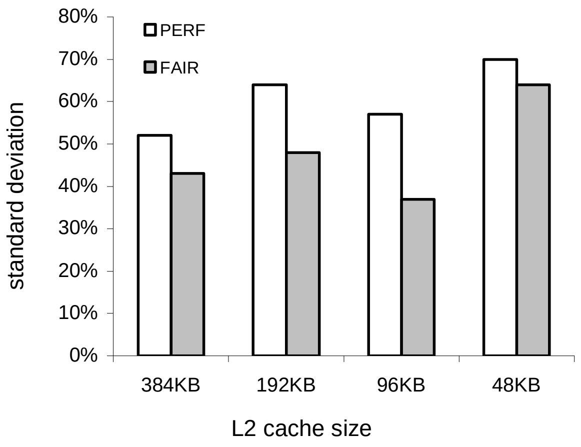

Figure 15. Evaluation of fairness for the two scheduling policies.

To estimate the degree of fairness of the balance-set scheduler, we used the following metric: standard deviation from the average CPU share. Under a fair-share scheduler, each workload would get an equal share of CPU, and the standard deviation will be zero. Figure 15 displays standard deviations for the balance-set schedulers PERF and FAIR.

Although the FAIR scheduler is somewhat fairer than PERF, its standard deviation from the average is still rather high. To give the reader an idea of how these standard deviations translate into the actual distribution of time fractions that each thread received, we show the diagrams of such distributions in Figure 16. For clarity, we show the numbers for the L2 size of 192 KB (best fairness), and for the L2 size of 48 KB (worst fairness). The diagrams show that the distribution of time slices among threads is not equal, but that each thread is represented, so none are starved.

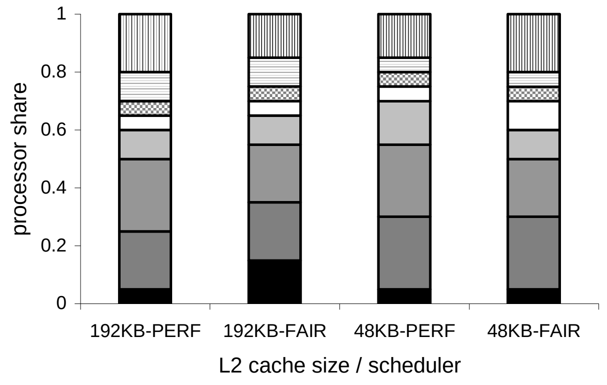

Figure 16. Share of processor time received by each workload with PERF and FAIR schedulers for L2 cache sizes of 192 KB (best overall fairness) and 48 KB (worst overall fairness). Each segment of the graph indicates a share of processor cycles for each of the eight workloads.

Achieving complete fairness and satisfying the miss rate thresholds are contradictory objectives. In its attempts to satisfy miss rate thresholds, the scheduler would inevitably favor cache-frugal threads and would schedule such threads more frequently than cache-greedy threads. Then, in order to compensate for unfairness, the scheduler would need to schedule only cache-greedy threads, without even considering cache-frugal threads for scheduling. However, this can result in a situation when some hardware contexts are left unused, whereas scheduling cache-frugal threads on those contexts would not have hurt the cache hit rate, but would increase overall system throughput, because more contexts would be kept busy. Therefore, we believe that we should redefine fairness in the context of CMT systems. Previous work on SMT scheduling reformulated the definition of fairness used on conventional systems making it more appropriate for multithreaded processors [2]. This work has also demonstrated that it is possible to incorporate thread priorities into an SMT-aware scheduler without sacrificing performance. We are hopeful that we can use ideas from this study to improve the fairness of our scheduler.

6.6. Implementation costs

In the previous section we have shown the potential benefits of balance-set scheduling for maximizing throughput on CMT systems. In order for this approach to be useful in practice, it must be practical to implement. While evaluating implementation costs precisely is the subject of ongoing work, we offer some discussion here.

In Section 6.3 we briefly discussed the scheduler costs involved in predicting the miss rates based on reuse distance histograms.

Another aspect of the implementation has to do with collecting the data required for building reuse distance histograms. To collect such data it would be necessary to monitor memory locations and record their reuse distances. The authors of the reuse-distance model have implemented a user-level tool that collects such data using the hardware watchpoint mechanism [25]. Their implementation works with optimized code, and is based on sampling, which can be performed at very low rates without loss of accuracy. The overhead of their implementation is less than 20% for long-running applications, with watchpoints being the most significant source of overhead. The authors are convinced that on systems where more than one watchpoint is available (such as UltraSPARC III®) the overhead can be significantly reduced. The overhead would also be lower if the data collection were done in the kernel (as would be the case with the scheduler), as opposed to on user level.

The amount of data that needs to be stored for every runnable thread in the system is small. The size of the reuse distance histogram can be made fixed. Even though the range of possible reuse distances can be very large, reuse distance histograms can be compressed. In fact, in our implementation of the reuse-distance model, we used compressed histograms, and the reported error rates were achieved using these compressed histograms. To compress a histogram, we aggregated reuse distances in buckets. Our predictions remained accurate even though we used fewer than 20 buckets.

7. Related work

The area that is most closely related to ours is on symbiotic scheduling for SMT processors [1, 2, 12]. The scheduler described in this study sampled the space of possible schedules, identified the groups of threads that perform best when scheduled together, and then attempted to co-schedule threads that were identified to perform well together. This scheduler was shown to improve system response time by 17% over a naïve scheduler. This method can be implemented with virtually no overhead and requires trivial hardware support. Our approach is different in that it uses modeling to predict the best schedule. Modeling may be preferable to sampling when the sample space becomes very large, such as on a system with a dozen of hardware contexts. It would be interesting to compare the effectiveness and costs of the method proposed in the SMT study to ours on a large CMT configuration. We are also hoping to apply ideas from the follow-up study on incorporating priorities into the SMT-aware scheduler [2] to improve fairness of our scheduler.

The authors of the SMT scheduling study argue that when evaluating performance of a scheduler on a multithreaded system, it is fairer to use a *weighted speedup* as a metric for throughput [1,2], rather than IPC. Weighted speedup measures the increase in throughput that is achieved when threads are co-scheduled on an MT processor over running on its own processor. They argue that this is a better metric, because it addresses the fact that threads do not get equal share of the multithreaded processor when they are co-scheduled. Since addressing fairness was not the primary objective of our study, we believe that using IPC as a measure of throughput was more appropriate as it has made our presentation clearer to a reader unfamiliar with the concept of weighted speedup. As we address fairness in our scheduler, we may resort to using weighted speedup as a metric for throughput.

To model cache miss rates for a group of threads we have changed a model that predicted cache miss rates for an individual thread to work with multiple threads. An alternative method has been proposed before [27]. This method is as accurate as our AVG method, but more computationally expensive.

In this study we have considered co-scheduling threads based on their cache behavior. The workload that we used contained independent threads that did not share data. A natural performance optimization to consider for a workload of threads that share data is to co-schedule threads based on their data sharing patterns. Thekkath and Eggers showed that such a scheduling policy does not yield significant performance benefits [28].

Cohort scheduling [10] is a scheduling infrastructure for server applications. This infrastructure batches execution of similar operations from different server requests together, and by doing so improves data locality, increasing processor IPC by 30% and reducing L2 cache misses by 50%. The applicability of this technique is limited to a specific class of applications (albeit an important one), and requires significant changes to application code. Our scheduler can make scheduling decisions by doing passive monitoring of running threads.

The Capriccio thread package [17] implements resource-aware scheduling by monitoring each thread's behavior, identifying the points at which a thread blocks, and measuring threads' resource requirements as they transition between blocking points. This knowledge about threads' resource requirements is then used in making scheduling decisions to optimize resource utilization. This method is more general than ours in that it can optimize for usage of different types of resources, but it requires very detailed monitoring of the program's state.

8. Conclusions

In this study we have identified the shared resources in a CMT system that are likely to become performance bottlenecks. We have determined that contention for the L2 cache has the greatest effect on system performance.

We have investigated how to leverage the operating system scheduler to reduce the pressure on the L2 cache. We have successfully applied the balance-set scheduling approach: our experiment demonstrated that this approach can reduce L2 cache miss rates by 20-40% and increase performance by 16-32%. To achieve similar reduction of miss rates by means of hardware, the L2 cache size would need to be increased by a factor of four.

In addition, we have adapted the *reuse-distance* cache miss rate model to work with multithreaded and CMP processors where concurrently active threads share a cache.

As the next step of our work we plan to repeat our experiments on a larger machine configuration and use commercial workloads. We also plan to investigate the nature of commercial workloads that we expect will be run on CMT systems, to see how much these workloads can benefit from the balance-set scheduling. Another important goal is to evaluate the implementation costs of this scheduling algorithm on a real system.

9. Acknowledgements

We would like to thank Matt Welsh, Keith Smith, James Laudon, John Davis, Nicolai Kosche, Erik Berg, and Anton Okmianski for their helpful comments and suggestions.

10. References

[1] A. Snavely, D. Tullsen, "Symbiotic Job Scheduling for a Simultaneous Multithreading Machine," *ASPLOS IX*, 2000.

[2] A. Snavely, D. Tullsen, G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor," *SIGMETRICS*, 2002.

[3] D. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture", *Intel Technology Journal Q1*, 2002.

[4] R. Alverson et al., "The Tera Computer System", *Proc. 1990 Intl. Conf. on Supercomputing*.

[5] A. Agrawal, B-H. Lim, D. Kranz, J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing", *ISCA*, June 1990.

[6] J. Laudon, A. Gupta, M. Horowitz, "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations", *ASPLOS VI*, October 1994.

[7] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, D. Tullsen, "Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading", *ACM TOCS 15, 2*, August 1997.

[8] Sun Microsystems web site, http://www.sun.com/ processors/throughput/datasheet.html

[9] Intel web site, http://www.intel.com/pressroom/ archive/speeches/otellini20030916.htm

[10] J. Larus, M. Parkes, "Using Cohort Scheduling to Enhance Server Performance", *USENIX Tech. Conf.*, June 2002.

[11] J. Lo et al., "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors", *ISCA*, June 1998.

[12] S. Parekh, S. Eggers, H. Levy, J. Lo, "Thread-sensitive Scheduling for SMT Processors", http://www.cs.washington.edu/research/smt/

[13] N. Tuck, D. Tullsen, "Initial Observations of the Simultaneous Multithreading Pentium 4 Processor", *PACT*, September 2003.

[14] D. Tullsen, S. Eggers, H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *ISCA*, June 1995.

[15] P. Magnusson et al., "SimICS/sun4m: A Virtual Workstation", *USENIX Tech. Conf.*, June 1998.

[16] D. Nussbaum, A. Fedorova, C. Small, "The Sam CMT Simulator Kit." Sun Labs TR. *In preparation; fedorova@eecs.harvard.edu.*

[17] R. von Behren, J. Condit, F. Zhou, G. C. Necula, E. Brewer, "Capriccio: Scalable Threads for Internet Services," *SOSP*, October 2003.

[18] "IBM Readies Power5 Microprocessor", http://www.supercomputingonline.com/nl.php?sid =4308

[19] E. Berg, E. Hagersten, "StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis", *ISPASS-2004*, March 2004

[20] P. Denning, "The working set model for program behavior", *CACM 1l*, 5 (May 1968), 323-333.

[21] P. Denning, "Thrashing: Its causes and prevention", *Proc. AFIPS 1968 Fall Joint Computer Conference, 33*, pp. 915-922, 1968.

[22] A. Agarwal, M. Horowitz, J. Hennessy, "An Analytical Cache Model," *ACM TOCS 7*, pp. 184--215, 1989.

[23] R. Eickenmeyer et al., "Evaluation of multithreaded uniprocessors for commercial application environments", *ISCA'96*.

[24] L. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing", *ISCA'00*.

[25] J. Torrellas, A. Tucker, A. Gupta, "Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors", *Journal of Parallel and Distributed Computing 24*, pp. 139—151, Feb. 1995.

[26] E. Berg, E. Hagersten, "Efficient Data-Locality Analysis of Long-Running Applications," TR 2004-021, University of Uppsala, May 2004

[27] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with application to cache partitioning," *15th International Conference on Supercomputing*, 2001.

[28] R. Thekkath, S. Eggers, "Impact of Sharing-Based Thread Placement on Multithreaded Architectures", *ISCA'94.*

[29] J. M. Borkenhagen, R. J Eickemeyer, R. N. Kalla, S.R. Kunkel, "A multithreaded PowerPC processor for commercial servers", *IBM Journal of Research and Development 44*, 6, pp. 885.

[30] A. Ailamaki, D. DeWitt, M. Hill, D. Wood. "DBMSs on modern processors: Where does time go?" *VLDB `99*, September 1999.

[31] A. Barroso, K. Gharachorloo, E. Bugnion, "Memory System Characterization of Commercial Workloads", *ISCA'98*.

[32] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker, "Performance characterization of a Quad Pentium Pro SMP using OLTP Workloads", *ISCA'98.*

[33] Y. Wiseman, D. Feitelson, "Paired Gang Scheduling", *IEEE Transactions on Parallel and Distributed Systems*, *14*, 6, pp. 581-592, 2003.